AgEcon SEARCH
RESEARCH IN AGRICULTURAL & APPLIED ECONOMICS

The World's Largest Open Access Agricultural & Applied Economics Digital Library

This document is discoverable and free to researchers across the globe due to the work of AgEcon Search.

Help ensure our sustainability.

Give to AgEcon Search

AgEcon Search
http://ageconsearch.umn.edu
aesearch@umn.edu

*Papers downloaded from **AgEcon Search** may be used for non-commercial purposes and personal study only. No other use, including posting to another Internet site, is permitted without permission from the copyright owner (not AgEcon Search), or as allowed under the provisions of Fair Use, U.S. Copyright Act, Title 17 U.S.C.*

The Stata Journal (2002)
2, Number 2, pp. 202–222

Speaking Stata: How to face lists with fortitude

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

Abstract. 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 these commands may be used interactively, and none is restricted to use in Stata programs. They are explained and compared in some detail with a variety of examples. In addition, a self-contained exposition is given on local macros, understanding of which is needed for use of `foreach` and `forvalues`.

Keywords: pr0005, foreach, forvalues, for, lists, local macros, substitution first

1 Working through lists

Have you ever typed a series of very similar Stata commands one after the other and wondered whether there was a quicker way to do what you just did? Or have you ever heard that Stata has a cluster of commands for cycling through lists, but put off learning about them until some future day? Or have you struggled to get any of the `foreach`, `forvalues` or `for` commands to do what you want? If you answered "Yes" to any of these questions, then this column should be of some interest to you. What it attempts is an overview of those three commands with sufficient explanation and examples to help you see your way to solving some of your own problems. Working your way through a list is a very common experience in data management and analysis. Being able to face that list with `for`titude cuts your typing, and thus imparts extra speed. Perhaps even more importantly, it also helps you work systematically and logically on your projects.

In all this, I am not going to assume that you know about Stata programming, or indeed about programming in any computer language. (If you do, you may well see family resemblances between Stata's syntax and that of other languages you know already.) Readers who have been following this series (previous columns in Cox (2001, 2002)) will recall a recurrent theme: I am trying to explain ways of using Stata which I call "programming without programming". The air of paradox is only superficial: a program in Stata, after all, is defined strictly by the fact that you start by typing `program define`, then type a series of Stata commands, and end by typing `end`, and we are not going to do that. But programming in a broader sense includes all ways of making the computer do almost all of the work, which you achieve by defining your problems in a language that the computer can understand. And Stata has structures for defining problems in which you cycle through lists, which are our concern here. They are extremely useful within Stata programs, but they can also be used interactively, and that is the emphasis in this column. Do not read too much into the fact that `foreach`

© 2002 Stata Corporation

and `forvalues`, introduced in Stata 7, are documented in the Programming Manual[1].

There is just one piece of Stata arcana which you need first: the idea of a local macro. The next section goes over that ground. As it happens, this is also one of the key ideas in understanding Stata programs.

2 Local macros

2.1 A character string given a special name

A macro in Stata is just a character string given a special name, so that you can then use that name, and Stata can understand that name, to refer to the contents of the character string. It is a way of setting up a mutually understood code.

If I type in Stata

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

then what I have done is assign the name `rhsvars` to the character string

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

The quotes here, `" "`, delimit the string. In fact, `" "` often are not necessary, although I do recommend that you use them, at least until you know when you can omit them without ensuing difficulty. The `" "` do make clear exactly where the string starts and ends, but they are not part of the string themselves.

But what if you did want to include one or more `"` characters in a character string? We will examine this point later.

The name of the string is usually much shorter than the string to which it refers. Put differently, the most obvious reason for defining a macro is so that you can refer to it later, often repeatedly, and save yourself some typing. In this example, you will probably recognize that the string includes names of variables from the auto data bundled with Stata. We can imagine these variables as appearing on the right-hand side of some model equations, hence our name `rhsvars`. We refer to it by using single quotes `‘ ’`, say

```
. regress mpg ‘rhsvars’
```

Note carefully that the opening or left quote or backtick `‘` differs from the closing or right quote or apostrophe `’`. On my keyboard, and on many others, the corresponding keys occur some distance apart: the `‘` key towards the top left, near `Esc`, and the `’` key towards the right of center, nearer `Return`. (Some Unix users may find that the backtick key has been mapped to something else. If so, using this stuff in Stata will be unnecessarily difficult until you find out how to undo that mapping.) Whatever your

[1] References to Stata manual entries clearly presuppose that you have access to a copy. If not, the advice is to use `search` and `help` within Stata to find help online.

keyboard, using a pair of ’ ’ (or for that matter ‘ ‘) will not work in Stata to indicate macro names. Here it is possible to be misled by several of the fonts which may be used within Stata: many represent closing quotes by a character more like ', which looks as if it might legitimately occur as one of a pair.

2.2 How Stata processes macro names: substitution first

We need to understand a little of how Stata interprets a command line like that just given. Essentially, all macro names are substituted by their contents *before* Stata attempts to execute any command. This fact is crucial to avoiding, or to debugging, some of the more subtle mistakes in using these commands. It is so important that I am going to single it out as the *substitution first* rule.

Let us go through that more slowly. Whenever you type a Stata command, Stata has two main things to do. The first task is to receive your command and to translate it into its own private terms. The second task is to try to execute your command. What is crucial here is that the first task includes substitution (sometimes called expansion) of any macro names which you used in the command line. In the example just given, Stata sees the left quote symbol ‘, a name, and the right quote symbol, ’, and thinks “Aha! a macro name!”. It then looks to see whether that macro name has been defined. In this case, `rhsvars` has previously been defined as `"trunk weight length turn displacement"`. Hence Stata substitutes (expands) the macro name, so that it now sees the line

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

Trying to execute this command is then the second task which it has to do. With the auto data loaded in memory, it should be straightforward.

But what could go wrong in handling a macro? The main pitfall is just getting the name slightly wrong. With local macro names, as with other names, Stata is case sensitive, so that upper and lower case letters are different: `‘RHSVARS’` and `‘rhsvars’` will mean different things, unless—exceptionally—you deliberately defined two macros with different names and the same contents. In addition, as is true almost everywhere else in Stata, Stata makes no attempt to correct your spelling.[2]

Suppose that by mistake you typed the name of some macro which does not exist. For example, suppose you typed `‘rhsvar’` when you should have typed `‘rhsvars’`, and you have not defined `‘rhsvar’`. In general, Stata does not mind you doing this. It may complain about the consequences, which are indeed unlikely to be what you want, but referring to a nonexistent macro is not, in itself, an error. In fact, Stata programmers often do this, and properly used, it turns out to be a very useful feature. Moreover, as this example implies, Stata does not allow you to abbreviate macro names.

[2]There is at least one exception to this, but it is as much by way of a little joke as anything else. When the `separate` command was being written, one objection to its name was that even some people very good at spelling in English have difficulties spelling this word correctly. Thus, as you may have found, there is also an undocumented `seperate` command, which issues a peremptory message about your spelling, but then passes the instructions to `separate` anyway. Stata is not usually so forgiving.

Stata's attitudes on macro names may be a surprise to you. After all, Stata complains vigorously whenever it expects names of existing variables, and you refer to a variable name which is not included in your data, a mistake which, once again, is most likely to be just a typo, as in typing `summarize npg` for `summarize mpg`. But it is the way Stata works. The rule is that a macro name which refers to nothing is substituted by nothing, that is, the empty string `""`. So

```
. regress mpg ‘rhsvar’
```

would be seen by Stata as

```
. regress mpg
```

which happens to make perfect sense as a way of getting the mean of `mpg` and a t-based confidence interval, but is unlikely to be what you wanted.

The macros you have defined are not set in stone. They will fade away at the end of a session; `exit` Stata, and all your defined local macros vanish. Less existentially, you can redefine macros at will. Suppose you wanted to add the variable name `gear_ratio` to `rhsvars`. Here is one way to do it, just overwriting the old definition:

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

and here is a better way of doing it, amending the old definition by adding to it:

```
. local rhsvars "‘rhsvars’ gear_ratio"
```

How does that work? It is another example of the substitution first rule; all macro names are substituted by their contents *before* Stata attempts to execute any command. So, Stata first substitutes `‘rhsvars’` by its definition. It now sees

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

and it then executes the command which that specifies, which happens to redefine the macro. This looks exactly the same as the first way of doing it, so why did I say that the second was better, as we obliged Stata to do the extra work? One reason is that Stata will be far faster than you are at substituting the contents for the name, and less error prone. An even more compelling reason is that it is a natural and helpful way of writing any operation in which we accumulate results step by step.

What often happens somehow—and more will be explained later—is that we loop around a statement like

```
. local results "‘results’‘new’ "
```

where `‘new’` refers to some new result. If the macro `results` is undefined when Stata comes to this statement, that is not a problem. The local macro `results` would then be created with the value of local macro `new`, followed by a space. Note the detail of the space, usually needed to separate elements of the list of `results`. This is useful even

for accumulating simple lists like `1 2 3 4 5`, even though there are other ways of doing that, principally through `numlist` (see [P] **numlist**).

If the macro `results` is already defined when Stata comes to this statement, then whatever is in `new` will just be appended to the existing value of `results`. Either way, we are seeing a method of accumulating results one by one.

Not only can macros be redefined readily, a key special case of that is redefining them as empty. To reverse a previous notion, a macro redefined as empty by

```
. local whatever
```

or by

```
. local whatever ""
```

ceases to exist.

2.3 What does 'local' mean?

We have yet to explain the jargon `local`. As its geographical counterpart implies, macros are visible within some space, a name space: that is, local macros are visible only within the Stata program in which they are defined. An interactive session counts as a Stata program, as does any other program, whether one that is defined as such (given its inclusion between `program define` and `end`), or one that is so by courtesy (that is, a do-file, or a set of commands in Stata's do-file editor).

What does this mean? Suppose again that you have defined local macro `rhsvars`. What would happen if you run a Stata program written by Stata Corp programmers, or by a Stata user, or—if you are a programmer yourself—by you on some previous occasion, and that program also uses some local macro `rhsvars`? This is more likely than you might think, even if you do not program yourself, and you never use anything not in official Stata, because the majority of commands in official Stata are defined by Stata programs. But even if, by sheer coincidence, another program includes the same name, it is not a problem at all, as the two macros are totally distinct, and never confused by Stata. And this is precisely what `local` means: visible only locally, within a program. While somebody else's program is running, Stata sees and works with their `rhsvars`. Even if it is changed during the running of a program, that has no effect on your `rhsvars`. That principle is vital for using local macros, because otherwise you would need to look inside every program you used and check for possible incompatibilities.

2.4 Global macros

There are, in contrast to local macros, macros in Stata which are visible everywhere, irrespective of what program is running, Stata itself (an interactive session), or a program so defined, or a do-file. 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, for example,

```
. regress mpg $rhsvars
```

The single `$` flags the start of a macro name. Very occasionally, you may need to avoid possible confusion by using braces as well, as in `${rhsvars}`. Although global macros have important uses, and quite possibly you may have used some already, I will say very little more about them in this column. What we are discussing needs only local macros, and even though global macros could be used for some of the same purposes, there is a risk that your global macros and somebody else's global macros could get confused. Indeed, the whole point about global macros is that multiple definitions cannot coexist.

2.5 Macros can contain numeric characters

Although we have defined macros as character strings, those strings in Stata are very often numeric characters, and so we frequently think of such macros as having numeric values. That piece of news may have been another surprise. You may be accustomed to the sharp division between string and numeric variables, as shown by the special methods you need to cross it, such as the `encode`, `decode` and `destring` commands. In fact, this double aspect of macros is largely a useful illusion, one which turns out to be yet another example of the substitution first rule. In other words, 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 the definitions

```
. local i 1
```

or

```
. local i "1"
```

Stata does the same thing: sets the local macro `i` to the character `"1"`. As mentioned previously, the delimiters `" "` are usually optional. The character concerned happens to be a numeric character, but that causes no problem. Now suppose we want to redefine the macro by incrementing by 1. If we are thinking numerically, what is most natural to us, and perfectly acceptable to Stata, is to write this as an evaluation:

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

To be precise, the equals sign flags to Stata that the expression which follows should be evaluated and it should feed the result of that evaluation to what precedes it. This is a new syntax compared with any examples we have seen so far, but a syntax that is very useful. Following the substitution first rule, what Stata sees is

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

It then evaluates the expression 1 + 1 and redefines the macro `i` as the result of the expression, namely the number 2, which it is happy to treat as the numeric character `"2"`. (That is a little more of a leap than you might think: remember that computers do their calculations in binary, not decimal.) The evaluation is nothing to do with the macro: it is part of the rest of Stata. That rest of Stata from context takes you to want `+` to be, as usual, the numeric operation addition. But suppose the expression was

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

Here the quotes `" "` insist to Stata that the macros are to be treated as strings, for which purpose those quotes are essential. The rest of Stata from context takes you to want `+` to be the string operation concatenation, that is, adding the strings by juxtaposition. As a result, `i` will now contain the string `"11"`. Again, they happen to be numeric characters, but `"April" + " " + "2002"` or `"Stata " + "Corp"` would have worked just as well.

Evaluations are also often used to produce the results of string expressions, as in

```
. local lcstring = lower("‘string’")
```

If you want to know more about string functions like `lower()`, see [U] **16.3.5 String functions**.

Note finally that

```
. local i "‘i’ + 1"
```

is different again. Given `i` of `"1"`, what ends up as the new local `i` is just the string `"1 + 1"`. Without the equals sign, no evaluation will take place, just substitution of the macro name by its existing contents. The same result would be produced by

```
. local i = "‘i’ + 1"
```

as, given the surrounding quotes, the `+` character is treated as just another character and not an operator. Here there would be evaluation, but it makes no difference to the contents of the macro.

2.6 Compound double quotes

We left one point hanging in the air, namely, how to get the `"` character into a macro. The answer is to use so-called compound double quotes, `‘"` and `"’`, as delimiters. The first marks the start and the second marks the end of a macro. What is more, they themselves can be nested within a macro definition as in

```
. local saying ‘"She said "Quotes can be tricky""’
. local metasaying ‘"He said ‘"She said "Quotes can be tricky""’"’
```

For more on this, see [U] **21.3.5 Double quotes**.

3 The three commands foreach, forvalues, and for

After such a long preamble, we can now move directly to the heart of the matter. Three Stata commands, `foreach`, `forvalues`, and `for`, make up a trio for cycling through lists. `for` has the longest history, going back to Stata 3.1 in 1993, with redesigns in Stata 5.0 (1997) and then in Stata 6.0 (1999). That bare summary hides a difficult search for a good compromise between simplicity and power. `for` was made more powerful (but more complicated) in 5.0, and then simplified again in 6.0, and even now it is arguable that `for` remains a command very useful when you understand it, but too easy to get wrong. For that reason alone, I would advise anyone new to the topic to start with `foreach` and `forvalues`. Note that while `forvalues` can be abbreviated all the way down to `forv`, `forval` appears to be the most widely used abbreviation, presumably because it is more nearly pronounceable, and I will adopt that in examples.

Let us start with a simple problem. Suppose we wish to `generate` a series of powers of a variable `y`. The slow but sure way of doing it

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

cries out for a simpler structure, even if you go no higher than y^4 *and* you are familiar with Stata's use of the `PageUp` key.

4 foreach

4.1 A first example

One way to do it is with `foreach`, and provides us with our first example:

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

Now that we know that `` `i' `` refers to a local macro name, the structure should seem less mysterious. More precisely, this is an example of one possible syntax with `foreach`, so we will back up and explain that more generally.

4.2 First syntax

Using more ordinary words, yet also a little abstraction,

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

The structure concisely encapsulates several features:

- A macro name must be given in the first part of the `foreach` structure.
- A list must be specified immediately after. In the syntax given here, `in` *list*, the keyword `in` specifies that you are going to spell out all the *in*dividual elements of the list. (That wording is a bit strained, but I find it helps me to remember the syntax.) There is another syntax, which we will see in a moment.
- One or more statements, at least one of which refers to the macro name, must be given within braces `{ }`. Quite how you space them out between those braces is up to you, so long as each statement occurs on a separate line. I like to indent commands by one tab, which in Stata is always set to a width of 8 characters, but that is partly a matter of taste. Stata does not care about indenting, but you should: poorly laid out code is more difficult to read, to understand, and to modify than well laid out code. (For one very helpful general statement on code style, see Chapter 1 of Kernighan and Pike (1999).)
- The `foreach` structure automatically defines the macro in turn as each of the members of the list and then substitutes the contents in the commands within the braces. In our case, the result is the three commands given earlier.
- The macro disappears at the end of the `foreach` structure. Thus you will lose any preexisting macro with the same name. If you are in the habit of using many local macros interactively, you may want to devise your own conventions for names to be used only with structures such as `foreach`.

Let us look at another problem: producing transformations of several variables, supposing say that positive skewness is our main concern, and we are considering square root, logarithmic and reciprocal transformations. Given three transformations and possibly many more variables than that, you should prefer to loop through a list of 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 (varlists in Stata jargon), especially wildcards and variable ranges, as detailed at [U] **14.4.1 Lists of existing variables**. Such features can be exploited with `foreach` by using the second syntax referred to just now.

4.3 Second syntax

```
. foreach macro_name 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. (That wording may also serve as a mnemonic.) In our last example, the *listtype* is

`varlist`, so, if we had just been inspired by reading Tukey (1977) to try transforming everything numeric, we could try

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

The `capture` here is a device to catch the occasions when a command will not work. It has many uses in this territory. The notion is easy to grasp: if the following command does not work, then just digest the output (including the error) and carry on regardless. The alternative, without `capture`, is that `foreach` will stop at the first statement which does not work, leaving you very often with a partly completed task. See [P] **capture** for more examples.

Why might the `generate` commands not work? Suppose in our case that the variables in memory included some string variables, so that any attempt to take logarithms or square roots or reciprocals of those variables would fail. The solution is to say, by means of `capture`, "unless this will not work".

Another reason they might not work is whenever you have variable names very near the 32-character limit on length (in Stata 7), such that the new name implied by `log`x'` or `sqrt`x'` causes an error message. In this case, and in others, we might want to carry on regardless, but still get an informative message, to see if we need to sort out any problems afterwards. There are several ways of doing this, but here is one, for simplicity shown with only the logarithmic transformation:

```
. foreach x of varlist * {

.         capture gen log`x' = log(`x')
.         if _rc { di "`x': " _rc }
. }
```

As the documentation for `capture` explains, Stata commands which fail produce a so-called return code which is not zero. `capture` puts the return code in `_rc`, so that it is accessible. The second line of code in this example displays the name of the offending variable and the return code whenever that is not zero, using the command `if` (see [P] **if**) and also the fact that `if _rc` is always equivalent to `if _rc ~= 0`, as explained in my previous column (Cox 2002, Section 5).

We have seen a *listtype* which is a `varlist`. `foreach` also allows four other types of list: those given within a `local`, a `global`, a `newlist` (that is, a list of new variable names; see [U] **14.4.2 Lists of new variables**) or a `numlist` (that is, a list of numbers; see [U] **14.1.8 numlist** or [P] **numlist**). So an earlier example could be written in terms of a `numlist`:

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

with no real gain in this case, beyond showing the principle. In passing, note that three-letter abbreviations like `num` are permissible for each *listtype*.

In general, this second syntax is much more powerful and more useful than the first: if your list contains tens, hundreds, or thousands of elements, you clearly do not want to be obliged to specify all of them, and the indirection made possible by (say) *varlist* or *numlist* notation is extremely helpful. What is more, Stata is often faster at cycling through lists given in this second syntax.

One very common application of `foreach` is just to produce univariate results for each of several variables. Sometimes a Stata command producing univariate results permits a varlist containing several variable names, while sometimes such a command may only be used with a single varname. `foreach` can be used as a wrapper to cycle through a varlist whenever only a single varname is acceptable. For example, let us imagine that we are drawing normal probability plots (also known as normal quantile plots or probit plots) using `qnorm`. See [R] **diagplots** if more information is needed. The syntax diagram of `qnorm` shows that it allows only the varname syntax, so we wrap a generic call inside a `foreach`:

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

`more` here ensures that each graph remains visible for as long as we wish to scrutinize it (see [P] **more**). As before, the `capture` device is available should it be needed. In this case, we would need `capture noisily` because `capture` alone would swallow the graph as well as any error message (see [P] **quietly**). Naturally, an alternative is just to make the effort to be sure that the varlist supplied contains only appropriate variables, in this example, only numeric variables which we wish to check for normality.

4.4 Do not confuse the two 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 {
```

although you will learn from the consequences of the first that it was not what you wanted. For this reason, any way of keeping the `in` and `of` syntaxes separate in your mind, such as the mnemonics suggested earlier, may be of help.

5 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_name = range_of_values {
.         one or more statements defined in terms of that macro name
. }
```

Here the *range_of_values* specifies a sequence of numbers and takes one of two main forms, exemplified by `1/10` and `10(10)100`. The first is short hand for `1 2 3 4 5 6 7 8 9 10`, that is, integers from 1 to 10. The second is short hand for `10 20 30 40 50 60 70 80 90 100`, that is, integers from 10 to 100 in steps of 10. Clearly `1(1)10` would be another way of specifying the first example. For decreasing sequences, use a form like `25(-1)1`.

There are yet more possible ways of specifying ranges. `10 20 : 100` and `10 20 to 100` are equivalents to `10(10)100`, which you may use if you prefer.

In short, whenever you want to cycle over a simple sequence of integers, `forvalues` is an alternative to `foreach`. `foreach` must store the integers, whereas `forvalues` calculates them one by one, giving it an edge in speed of execution.

Using `forvalues` gives a possible alternative to our first example with `foreach`, generating powers of a variable:

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

For a new example, let us again imagine that we are drawing normal probability plots using `qnorm`. One key issue in assessing the variability on such plots is how much variability would be expected even if the parent distribution really were exactly normal (Gaussian). Suppose our variable's normal plot were saved in a gph-file by a line such as

```
. qnorm ourvar, saving(ourvar)
```

It is helpful to be able to compare such a plot with a reference portfolio of plots for normal samples of the same size. Such a procedure is suggested, for example, by Wild and Seber (2000) in their excellent text (p. 413). This `qnorm` command would use every value of *ourvar* (so long as no values were missing; we will set that possible complication on one side). Knowing Stata's predilection for showing multiple graphs in a $k \times k$ array, we will go for 5×5 as a modest number of graphs which remain fairly legible individually. Given the existing plot already in *ourvar*`.gph`, we need 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 the local macro `i` goes from 1 to 24, at each step we get a new sample from a standard normal N(0, 1) by using first the `uniform()` function and then the `invnorm()` function. We then use `qnorm` to draw a normal probability plot, which will flash by, but we save the graph image in a file with the same name as the variable. As a further small but useful flourish, we accumulate the names of the variables (and thus the gph-files) in a local macro `G`.

Now all the ingredients are ready. We can just redraw our saved graphs:

```
. graph using ourvar ‘G’
```

To recap on the `forval` structure: the macro created, namely `i`, runs through the values 1 through 24. It is substituted in turn as a part of a new variable name (e.g., `v‘i’` becomes `v1`) and as part of a new filename (same example, different context). We see also an illustration of an important principle: such a structure can be as valuable for its side effects—in this example, the graph files left behind—as for its immediate and obvious effects. But most clearly of all, typing this is a large saving on repeating a `generate` command and a `qnorm` command for each of 24 variables. And the reference graphs remain of use if we want to look at other variables in this way.

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’ "
. }
```

which has at least one advantage: a side effect of declaring *listtype* to be `newlist` is that Stata would check before entering the loop that no existing variables had any of the variable names `v1-v24`.

6 Initializing before foreach or forvalues

In all the examples so far we have been able to set up our `foreach` or `forvalues` loop straight away. There are many problems, just a little more complicated, in which we need one step more, namely, to initialize one or more things before we enter the loop. This can arise in various ways, and we will look at two. We might want to create a new variable, but the recipe for creation is too complicated for a single command. (The limit is likely to be inherent not so much in Stata's syntax as in our own need to keep things simple, but therein is no shame and much benefit.) Or we might want to populate a matrix with entries from separate calculations.

6.1 Initializing a variable

Such tasks often arise in cleaning up fairly large datasets containing string variables, say names of countries or companies or diseases. If the dataset has been produced by, or from the responses of, several people, or even by one person, there may be various small inconsistencies. To be safe, we keep the original variable, but work first at producing a more consistent version. If we are happy with our work, we might be bold and drop the original.

Imagine a survey with a question on vacation destinations. Respondents were asked to state countries they wished to visit. Initial inspection reveals, among many other details, a multitude of near synonyms for Britain: Britain, Great Britain, UK, United Kingdom, and so forth[3]. We decide to combine these all into Britain.

A constraint on this is that we can only use `generate` once: thereafter we can make any number of changes, but they must be done through `replace`. It is often advisable therefore to put a `generate` statement outside the loop as in

```
. generate str1 Destination = ""
. replace Destination = destination
. foreach c in "Great Britain" "UK" "United Kingdom" {
.         replace Destination = subinstr(Destination,"`c'","Britain",1)
. }
```

Three extra details arise here. First, in producing a copy of a string variable, we could specify the same variable type as the original: `describe` or `codebook`, say, will tell us what that is. But I usually `generate` an empty `str1` variable and follow with a `replace`, thus getting Stata to promote the variable automatically. This saves the time (person time, not computer time) of the `describe` and exploits the fact that Stata will be economical in choosing the string variable type needed. Moreover, on occasion you might recall or guess that a variable is (say) `str11` when it is really `str12`. Stata will take you quite literally if you specify a string variable type and will quietly ignore what it takes to be superfluous characters. You may be some distance further on before the damage is spotted.

Second, note how strings with embedded spaces such as `"Great Britain"` need delimiting quotes.

Third, the `subinstr()` function is explained at [U] **16.3.5 String functions**.

6.2 Populating a matrix

Many bivariate commands produce single-number statistics which we might want to display in the form of a two-way table. `correlate` does this automatically: given a varlist of numeric variables, it will produce a correlation matrix. However, other commands are not set up to do this automatically. `foreach` makes it possible to overcome that in a relatively straightforward way. This example is a little more complicated than any so

[3] Perhaps even England, despite facts revealed by any good atlas.

far. If you are trying to copy this in your own Stata, or to modify it to fit a related problem of your own, you will probably want to make use of a text editor, which could be Stata's own do-file editor[4].

Consider `ktau`, which takes only a pair of variables *varname1* and *varname2*, and calculates Kendall's tau. (See [R] **spearman**.) Having read Newson (2002) on the many virtues of τ_a in particular, we might like to look systematically at results for the auto data. We will set aside the string variable `make` and the binary variable `foreign` and look at those ten variables on at least an ordinal scale, `price-gear_ratio`.

To grind through all the possibilities, we need two `foreach` loops, one nested inside the other. Here is our first attempt:

```
. foreach v of var price-gear_ratio {
.         foreach w of var price-gear_ratio {
.                 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. It extracts `price` as the first variable of the `varlist` `price-gear_ratio` and sets that as the first instance of local macro `v`. Then it extracts `price` as the first variable of the (same) `varlist` `price-gear_ratio` and sets that as the first instance of the local macro `w`. Note that it is vital that we have two distinct macros for referring to the two variable names, as only occasionally will the two variables be identical. Even in those cases we should really do the calculation, as τ_a will only be equal to 1 if there are no tied values. Then Stata substitutes the macro names by their contents, and executes `ktau price price`. So at this point Stata has completed one step through the outer loop and one step through the inner loop. Here now is the crucial rule: the innermost loop is completed first. Stata therefore resets local macro `w` to the second variable of `price-gear_ratio`, which happens to be `mpg`, and executes `ktau price mpg`. It continues cycling through the `varlist` until the last variable, executing `ktau price gear_ratio`. At this point the inner loop, for macro `w`, is complete, so Stata returns to the outer loop, for macro `v`. That is reset to `mpg`. The inner loop then restarts all over again, with `w` reset to `price`, so we get `ktau mpg price`. The inner loop then continues until finished, the outer loop restarts, and so on and so forth, until for 10 variables in each list, all 100 pairwise correlations are calculated.

You may have noticed that we have been a little extravagant in our use of computer time. We made Stata calculate both `ktau` x y and `ktau` y x even though the results are identical. In general, given p variables, there are at most $p(p+1)/2$ values of τ_a of substantive interest, and obliging Stata to calculate all p^2 values is wasteful by a factor of almost 2. However, for a small dataset and a modest number of variables, the extra time is trivial, and less than it would take to modify the code. If we wanted

[4]Helpful advice to Stata users on text editors is made difficult by the very large number which exist and the diversity of platforms—Windows, Macintosh and Unix—on which Stata runs. Nevertheless, the FAQ at *http://fmwww.bc.edu/repec/docs/textEditors.html* gathers detailed notes from several Stata hands.

an efficient program to run repeatedly on a variety of problems, including those with greater numbers of observations, making savings would be important, but it is less so in interactive code.

But there is a more important detail we have glossed over, the treatment of missing values. Left to its own devices, Stata will use as many observations as possible in calculating τ_a. With the auto data, this will be 74, unless one of the variables is `rep78`, for which 5 values are missing, so that it will be 69. This behavior may be what you want, but it is more likely that you would prefer casewise deletion, meaning that the same observations are used in every calculation. One way to achieve this is to use `egen`, see [R] **egen**, to count missing values across variables:

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

and then to stipulate that the `ktau` is calculated `if nmiss == 0`, that is, only if all variables specified are nonmissing in each observation.

Results from these calculations are still produced in a long and awkward list. It would be much more practical to have them in a table. One of the easiest ways to do this is to populate a matrix with the correlations, and then use `matrix list` to take care of all display matters. A first step is to set up, before the `foreach` loops, a matrix of the right size, which we can fill arbitrarily. Each element of the matrix will then be replaced in turn within the `foreach` loops. The matrix function `J(`r`,`c`,`z`)` creates an $r \times c$ matrix, all of whose elements are z (see [P] **matrix define**). 10 variables give a 10×10 correlation matrix:

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

I have also used 10, an impossible result for τ_a, as the typical element in the initial matrix as a check that the code cycles through all the possibilities intended. Any 10 lurking in the final result would signify an error in my code.

Now what we need to do is pick up the result left behind in memory after each time `ktau` is executed. The manual entry on each command, in a *Saved Results* section, will document any results which are temporarily accessible in Stata's memory after a command is executed. "Temporarily" means until you `exit` Stata, or until the next command of similar type is executed, whichever happens sooner. If the manual entry is not accessible, typing either `return list` or `estimates list` immediately after a command will show what is saved where. The rule of thumb is to try `return list` first unless the command can be thought of as estimating a model, in which case try `estimates list` first. In our case, using either the manual or `return list`, we find that we need to pick up `r(tau_a)`.

Next we need to cycle through row and column indexes. One way is to initialize a row index `i` and a column index `j` to 0 just before the corresponding `foreach` loop starts, and then increment by 1 every time we set `v` or `w`, as appropriate. Then the replacement of each matrix element is to `tau[‘i’,‘j’]`.

Putting the code together:

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

I will flag again how systematic indenting is an aid to understanding the code. Notice how each right brace `}` marks the end of each `foreach` loop.

There is still one thing missing, intelligible labeling of the rows and columns of the `tau` matrix. It is worth knowing how to do this, as it makes the result so much easier to read. Another command usually billed as a programming command, `unab` (see [P] **unab**), saves your typing. It "unabbreviates" (ugly word, but useful action) a variable list into a local macro, after which you can attach the names to the `tau` matrix (see [P] **matrix rownames**):

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

7 The for command

`for` is the third of our trio of commands for cycling through lists. Treating it last inverts the historical order: as mentioned previously, `for` was introduced in Stata in version 3.1 in 1993 and thus long predates `foreach` and `forvalues`, introduced in version 7 in 2001. Explaining it this way reflects my experience that while very useful for easy tasks, `for` all too often becomes awkward or problematic for slightly more difficult tasks. That aside, let us first revisit some of our examples and see how they would be done with `for`.

7.1 Examples

For our powers of a variable example, we could use

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

or (better)

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

For our transformations of a variable example, we could use

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

For the normal probability plots,

```
. for var varlist : qnorm X \ more
```

or

```
. for var varlist, pause : qnorm X
```

7.2 Discussion

The pattern underlying these examples of `for` is, with more details to be explained shortly,

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

Looking at the examples just given in this light shows both similarities and differences compared with `foreach` and `forvalues`.

- `for` has a notion of *listtype*, just like `foreach`, except that the possible types are `varlist`, `newlist`, `numlist` and `anylist`. Arbitrary lists can—and indeed must—be specified as type `anylist`. Three letter abbreviations of each *listtype* are allowed.
- `for` does not use local macros to hold successive members of lists. Instead, you indicate by a placeholder (by default `X`) where the member of the list belongs in the command(s) to be executed. The principle is similar, but not identical.
- `for` does not use braces, nor does it allow the use of separate lines, in specifying the commands to be executed. The syntax is that commands follow a colon `:` and that multiple commands are separated by backslashes `\`.
- `for` has a few special options of its own. `pause` is one such option used in the previous set of examples.

This is not the whole of the story, and further options and features, especially the use of multiple lists to be processed in parallel, are explained at [R] **for**. But immediately you may be able to sense why `for` divides experienced Stata users into rival camps, say the "`for`" and "against" factions. On one side, many users appreciate its conciseness. With a little experience, you can often encapsulate a series of commands in a single and fairly short command line with `for`. On the other side, while the structure may include repetition of several commands for each member of the list, separation of those commands by backslashes means that such code can be difficult to read, and thus

difficult to understand and above all to debug. Naturally, readability depends also on other practices. It helps a great deal to have personal rules such as always surrounding backslashes with spaces.

One key difference is not obvious from these examples, or indeed otherwise. Structures using `foreach` and `forvalues` are best thought of as a series of commands under the control of a specified loop. A notable consequence is that any local macros will be substituted just before each command is executed. Therefore, their contents may vary through the loop. We have seen this principle in operation: it is what makes possible the accumulation of results one by one into local macros, as in the normal probability plot example. 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.

There is no contradiction here, nor any special behavior. Everything in the last paragraph follows from the substitution first rule. But what this rules out for `for` is any use of local macros whose contents vary through the loop[5]. Attempting to try to manipulate local macros within `for` is thus almost always problematic. That does not rule out use of local macros to specify part of the command line, which from the substitution first rule (yet again) is no part of `for`'s concern. So, to give a simple example, if the local macro `myvars` contained a varlist, then

```
. for var `myvars' : qnorm X \ more
```

is perfectly acceptable. Stata will substitute the contents of `myvars`, and only then does it try to execute the `for` code.

In addition, although we will not explore the point in detail, it can be difficult or even impossible to nest `for` commands, such that one `for` calls another `for`. On the whole, do not even try, as the equivalent code with `foreach` and/or `forvalues` will be much easier to write.

One less subtle pitfall with `for` is that the placeholder (by default `X`) might occur as part of the command line. As you will be aware, Stata command syntax is based almost uniformly on names composed of lower case letters. There are several exceptions, such as `xi`, see [R] **xi**, which produces variable names including characters such as `I` and `X`, and indeed it uses them partly because Stata users are exhorted, by precept and by example, to use lower case for the most part. In addition, it might be that `X` occurs as part of some other text. Not surprisingly, Stata has no way of distinguishing occurrences of the placeholder from occurrences of the same symbol(s) which are intended literally, and all occurrences of those symbols would be treated as placeholders, producing typically either an error message or bizarre results.

Very simply, when this happens, or if you want to do it anyway, you need to specify another placeholder, as in

[5] There is at least one trickier way of doing it, in which the series of commands is bundled into a do-file, and the `local` manipulations kept out of sight (and thus out of mind) of `for`. In my view, that way has no advantages over a more direct approach using `foreach` or `forvalues`.

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

or

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

The second example should make it plain that a placeholder need not be a single symbol.

The `in` syntax here is the reverse of that of `foreach`. That is just the way it is. It is of course equally true that the alternative placeholder should not appear as itself within the command line. Use of upper case letters is not mandatory, but rather a strongly recommended convention. An alternative is to use a character otherwise unlikely to occur, the leading candidate being `@`.

A similar detail is that you will need to protect text including backslashes, say Windows filenames, by inclusion within quotes against the inclination of `for` to treat those backslashes as command separators.

There is a further point of interest to anybody who wishes to embed structures like this into do-files (and, especially, Stata programs) to be used many times over. `for` is implemented as a Stata program defined in an ado-file, which is at present a few hundred lines long. In contrast, `foreach` and `forvalues` are implemented in C code as part of the Stata executable. That implementation as a Stata program poses an overhead of interpretation. As it executes the program defining `for`, Stata needs to interpret each successive command line and thus will be much slower than would be the case with `foreach` or `forvalues`. Often you will find this difficult to detect, but in principle the latter structures are much to be preferred in code to be reused frequently.

8 Summary

What is my "take home" message from this column? The effort needed to master these commands is rewarded by the payoff in efficiency and system in going beyond what is easy to do directly in Stata. Even the apparently esoteric notion of local macros is not as bizarre as it may first seem: their name is perhaps their strangest feature. Grasping that notion is like the first steepish climb out of the valley on many a mountain walk. Once upon the first ridge, the view justifies the puff and the perspiration, and you can walk more easily. If you know `for` and like it, then stick with it. Whether or not you know `for`, do check out `foreach` and `forvalues` and add them to your repertoire.

9 What's next?

In my next column I will look at the contrast between string and numeric variables. As briefly mentioned in section 2.5 above, the difference between them marks a great divide between kinds of data in Stata. However, many datasets are in some sense problematic in this respect, leading to difficulties in input, management and analysis. I will focus on the logic and uses of the several ways which now exist for crossing that divide.

10 Correction to previous column

In my previous column Cox (2002), the description of ASCII order on page 87 contains a mistake. In ASCII order, lowercase letters such as `"a"` go *after* uppercase letters such as `"A"`.

11 Acknowledgments

For various helpful comments, I am grateful to Kit Baum, David Kantor, Scott Long, Roger Newson, and Joe Newton.

12 References

Cox, N. J. 2001. Speaking Stata: How to repeat yourself without going mad. *Stata Journal* 1: 86–97.

—. 2002. Speaking Stata: How to move step by: step. *Stata Journal* 2: 86–102.

Kernighan, B. W. and R. Pike. 1999. *The Practice of Programming*. Reading, MA: Addison–Wesley.

Newson, R. 2002. Parameters behind "nonparametric" statistics: Kendall's tau, Somers' D and median differences. *Stata Journal* 2: 45–64.

Tukey, J. W. 1977. *Exploratory Data Analysis*. Reading, MA: Addison–Wesley.

Wild, C. J. and G. Seber. 2000. *Chance Encounters: A First Course in Data Analysis and Inference*. New York: John Wiley & Sons.

About the Author

Nicholas Cox is a statistically-minded geographer at the University of Durham. He contributes talks, postings, FAQs, and programs to the Stata user community. He has also co-authored eight commands in official Stata. He was an author of several inserts in the *Stata Technical Bulletin* and is Executive Editor of *The Stata Journal*.