The Stata Journal (2003)
3, Number 2, pp. 185–202

Speaking Stata: Problems with lists

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

Abstract.

Various problems in working through lists are discussed in view of changes in Stata 8. `for` is now undocumented, which provokes a detailed examination of ways of processing lists in parallel with `foreach`, `forvalues`, and other devices, including new, concise ways of incrementing and decrementing macros and evaluating other expressions to do with macros in place. New features for manipulating lists held in macros and the new `levels` command are also reviewed.

Keywords: pr0009, lists, for, foreach, forvalues, levels, macros, tokenize

1 Introduction

Writing a regular column is a challenging exercise and, in many ways, a surprising one. The obligation to file copy regularly, even if only once a quarter, has had the unexpected side-effect of increasing my respect for journalists. More seriously, the challenge of writing a series of contributions raises the issue of how long a column can be sustained before all the author's hobbyhorses have been ridden into the ground, especially when my self-defined territory deliberately excludes many large chunks of Stata that are supremely well-documented, or just not part of my expertise. Stata itself goes a long way toward relieving any doubts on that score. A new release brings so many features that I will not have time to comment on more than a few before the next release appears in turn. As the British say, the task is like painting the Forth Bridge, a bridge supposedly long enough that, by the time the painters have reached the end, they must go back and start again.

The news in 2003, which is not news to almost all readers, is that Stata 8 is out. The big news within that for most users includes new graphics and dialogs, but—as promised in my last column—the main topic in this column is once again dealing with lists, which in one way or another has been one of my most persistent themes. You know that in many Stata problems you want to do (almost) the same thing again and again. You want to work your way through a list. Often Stata does this for you. Automatic looping over observations for many operations comes as a surprise to users who have become accustomed to spelling out such loops in other software. But often you have to ask Stata to do it for you, and our concern is with the tools provided for that purpose.

I have, inevitably, a list of things I want to go through on this general topic of lists. Here is the menu:

- `for` is on its way out (a news item, with some journalistic comment).
- How can you best work your way through parallel lists without `for` (a detailed discussion, with explanation of codes)?
- Have you checked out [P] **macro lists** yet (another news item, rather drier)?
- `levels` is a simple new command to help with some list problems (something easy to digest to end with).

2 Farewell to for?

First give the bad news, it is said. If you used Stata 7 or a previous version, then quite possibly the demise of `for` will have been a small piece of bad news for you.

Strictly, demise is, as yet, too strong a word. More precisely, `for` is no longer on display as part of the official Stata repertoire. The code is still there, and if you used `for` in your do-files or programs, they should still work in Stata 8. If your do-files do not work, it will be for some other reason. The online help and the manual entry have gone, so for either or both of those you must depend on whatever is accessible to you from previous versions. This is not prohibition, but it certainly amounts to discouragement. `for` is to be assigned to history, and you are best advised to make greater use of the alternatives, particularly if you imagine using Stata for some time to come.

Why has this been done, particularly in view of Stata's longstanding commitment to maximizing continuity from release to release? `for` has long served many users well. Some regard it as one of their favorite commands. But it has severe limitations, several of which were rehearsed in a previous column (Cox 2002b). I will mention here just three.

First, `for` was, frankly, idiosyncratic and did not mesh well with the rest of Stata. Particularly limiting was the fact that it could not make use of local or global macros in the ways that might be expected. There was no deep-seated bug here, and `for` was not behaving differently from other Stata commands in this respect; merely, its behavior often clashed with users' preconceptions of how it should behave.

Second, once `for` had been learned by many users, they often tried to build in multiple loops and multiple commands. For all sorts of reasons, this often did not work. `for` was just not strong enough to bear the weight that some users tried to put on it. This was not because the code was incompetently written; it is just too difficult a task to handle what were in one sense complete programs codified as just one command line. Even when `for` did work, the resulting code was often difficult to read, understand, or revise. Users were thus being tempted into bad habits because of the features that `for` appeared to offer.

Third, `for` was slow. Frequently, that was not obvious to users because of the speed of Stata and of their computers, but `for` was implemented as interpreted code, thus imposing an overhead. One of the main characteristics of the new commands `foreach` and `forvalues` introduced in Stata 7 is that they are implemented as compiled code and thus run much faster.

`foreach` and `forvalues`, new in Stata 7, are now the main workhorses for looping through lists. If these are new to you, apart from the online help, see [P] **foreach** and [P] **forvalues** or my earlier tutorial (Cox 2002b). The assumption here, naturally, is that you have access to a full set of manuals. Failing that, the online help should always provide some assistance. You may need to supplement those manual entries with some reading about local macros, particularly within [U] **21 Programming Stata**. These references to the *Programming Reference Manual* do not mean that you have to write Stata programs to be able to use these constructs. On the contrary, they are often used interactively, and you can incorporate them in do-files or code you run from the do-file editor.

(Once again, what is a Stata program? Simply, whatever is defined by a Stata `program` command. Note that this definition is not circular, as a question in one language (discussion of Stata) is answered by an answer in another language (Stata). However, we will not be writing any Stata programs in this column.)

For Stata programming before Stata 7, `while` was the main workhorse in previous versions, and it is still useful for some problems. For interactive use, `for` was the main workhorse in previous versions, but as said, its use is now officially discouraged.

3 Problems in parallel

All this leaves open one large question. `for` offered machinery for handling problems involving two or more lists in parallel. Indeed, to a large extent, this is precisely what `for` was designed to do. How are these parallel problems best handled now in Stata 8?

Here to start with is a small but genuine example. I have been bitten by it with climatological data, and a questioner on Statalist was bitten by it with employment data. You have monthly data, but they come as columns, so that your Stata dataset contains variables clearly and helpfully named `jan` through `dec`. You realize that you would be much better off with the data as one long column, and the smart way to arrange that is by a `reshape`. However, `reshape` expects such variables to have a common stub, so that they are, say, `m1-m12`, the stub being `m` and the suffixes `1` through `12`. You need to `rename` before you can `reshape`. You could grind your way through the twelve commands

```
. rename jan m1
. rename feb m2
```

and so forth, but it is not long before you want a better way of doing this, especially when the next version of the problem to bite you involves hundreds of variables. Setting aside the fact that a special tool exists for this task (Cox and Weesie 2001), how can this be done from first principles? `for` adepts will know a solution:

```
. for var jan-dec \ new m1-m12 : rename X Y
```

Some users might prefer not to rely on the identifiers `X` and `Y` provided by default to refer to the current members of the first (`X`) and second (`Y`) lists, and they might want to specify their own:

```
. for OLD in var jan-dec \ NEW in new m1-m12 : rename OLD NEW
```

Feel free to prefer that as more explicit style, at least as a matter of historical taste. The only biting constraint on choice is that the identifiers should not occur literally within the command. The rationale is simple: each instance of these identifiers will be substituted in turn by the successive members of the lists, so we must ensure that happens only when desired.

Standing back from the detail of how it might be written, admire the concise one-command solution, on which we cannot improve with Stata 8 tools without writing a program to do the work off-stage. Indeed, for those who feel that nostalgia is not what it used to be, here is an example of `for` at its best.

Let us take it apart, step by step. `for` is being presented with two lists. We are assuming here that the variable list `jan-dec` refers to the 12 variables you want, which are adjacent in memory, and in the right order. In practice, that is probably true. If necessary, you should verify this using `describe` or `ds` or the Variables window. The list of new variable names will be used in the successive `rename`s. More generally, what `for` does is to check that the lists are what is claimed—that (a) `jan-dec` really is a list of variables in memory and that (b) `m1-m12` does define new variable names that are legal and all not in use—and that the lists do have the same length (12 elements in our case). That is a lot of useful checking, just in case our assumptions do not match the dataset in Stata's memory. Once it is done, `for` cycles through what follows—in this case, just a `rename` command with different pairs of old and new variable names.

More generally, what `for` offered, for two or more lists of a certain length, was to go through them in parallel. Member $i = 1, \dots, I$ of a first list was linked with the corresponding members of any other lists, so, whatever the complexity of the code, `for` would cycle through just I runs of the commands specified.

3.1 Using foreach and forvalues

How can this be done with either `foreach` or `forvalues`? At first sight, there are two basic difficulties. First, each of those commands takes just one list as argument. Second, if those commands are nested, the result is to run not through lists in parallel but through all cross-combinations of lists, producing in effect an array of results. This is worth understanding, irrespective of our current example, so let us pause to explain in detail. Consider this simple example:

```
. foreach letter in a b c {
.         forvalues i = 1 / 3 {
.                 display "`letter'`i'"
.         }
. }
```

What happens here is that we get 9 results, not 3, as can be seen by following through the loops. The outer `foreach` loop starts with `a` from its list. Then the inner `forvalues` loop starts with 1 from its list. As a result, the combination `a1` will be printed. The innermost loop always cycles fastest, so continuing, we get `a2` and `a3`. Then we jump to the outer loop once more, and so on. In succession, therefore, we will get `a1`, `a2`, `a3`, `b1`, `b2`, `b3`, `c1`, `c2`, and `c3`.

More generally, what nested loops offer, using any combination of `foreach` and `forvalues` for two or more lists, is to go all through the combinations, the Cartesian product if you prefer. For lists with I, J, K, $\ldots$ members, we will get $I \times J \times K \ldots$ results, hence the idea of an array of results. For many problems, this is exactly what you want. Populating a matrix is one class of examples. What you need to do is to initialize a matrix of the required size and then cycle over rows and columns, overwriting each element with the required number from some calculation. There is a case study of getting a matrix of Kendall's tau in Cox (2002b).

This is all extremely useful, but it would be disappointing if it now appeared that the side-effect of the syntax of these commands is that getting results from parallel lists is no longer easy. Fortunately, it is not too difficult. The key point is that, although `foreach` and `forvalues` do not offer machinery for doing this themselves, there are several ways of ensuring that you do it in harmony.

Returning to our `rename` problem, last seen as something like

```
. for var jan-dec \ new m1-m12 : rename X Y
```

here are various ways to do it. Each will be discussed in some detail; we will summarize the main ideas afterwards.

3.2 Automated foreach with arranged stepping through an integer sequence

We cycle through the variables with `foreach` and arrange that we step through an integer sequence in harmony. So, we have the idea of a loop

```
. foreach v of var jan-dec {
.         ...
. }
```

and we will step, one by one, through the integers 1 through 12. Here is the most common way to do that, in Stata 7 and later:

```
. local i = 1
. foreach v of var jan-dec {
.         ...
.         local i = `i' + 1
. }
```

and we will fill in the whole block

```
. local i = 1
. foreach v of var jan-dec {
.         rename `v' m`i'
.         local i = `i' + 1
. }
```

The logic is simple. Outside the loop, we initialize the local macro `i` to 1. Once inside the loop, the first statement becomes

```
. rename jan m1
```

Here are two steps: The local macro `` `v' `` is replaced by its contents, which the first time around the loop is the name of the first variable in `jan-dec`, namely `jan`. Just as with `for`, `foreach` acts on the keyword `var` to treat `jan-dec` as a *varlist* and to unpack it. Even before that, there is a check that it really is a valid *varlist*.

Similarly, the local macro `` `i' `` is replaced by its contents, which the first time around the loop is 1.

Putting the original and substituted text together, we have the command just given. Then we increment the local macro (increase it by 1) so that it now takes on the value 2. Note that incrementing, without qualification, always means adding 1. Similarly, decrementing, unqualified, always means subtracting 1.

In a sense, what is happening here is that we are trading on ambiguity. When first initialized, the macro is set to 1 by a numeric operation, as `local i = 1` forces an evaluation of the expression to the right of the equal sign `=`, and what is there looks to Stata like something numeric to be evaluated. In contrast, the command `local i = "1"` would have insisted to Stata that the thing put into the macro is a one-character string. We did not say that, although we could have. When you look at it, you will not detect much evaluation needed, but remember that you are smarter than Stata. Later, when we substitute the contents of `i` within ``"`letter'`i'"``, the current value is substituted as a character within a string. Later still, when `i` is being incremented, the absence of `" "` again signals a numeric rather than a string operation. In other words, we are flipping back and forth between treating macro contents (for example, `1`) as numbers (1) and single-character strings (`"1"`). Most of the time, we can be cavalier about this, but occasionally you will need to think harder about what Stata is understanding by what you are asking it to do.

To summarize our progress so far: We have now one way to do it. It may not appeal very much, but it works. You could have done exactly that in Stata 7, but here now is a new twist, introduced in Stata 8, which is worth mastering. It will be familiar to you if you have experience in programming C or almost any language modeled on C. Let's show this in practice first and then explain:

```
. local i = 1
. foreach v of var jan-dec {
.         rename `v' m`i++'
. }
```

We have lost a line and gained a reference to `` `i++' ``, which is one of a set of quadruplets. Seen all at once, with our example macro name `i`, they are `` `i++' ``, `` `++i' ``, `` `i--' ``, and `` `--i' ``. In context, Stata sees `` `i' `` and knows that this means to "evaluate the local macro `i` and then put its contents *here*" as part of parsing a command that is then executed (if legal). What we have now with `` `i++' `` is threefold:

1. Evaluate the local macro `i`.
2. Put its contents *here*.
3. Add 1 to the value of the macro.

So if before Stata reads this, `i` evaluated to 1, then after Stata reads this, it will evaluate to 2. The incrementation is performed in place.

The other quadruplets are all similar. `` `++i' `` increments before use, `` `i--' `` decrements after use, and `` `--i' `` decrements before use. Double characters (`++` and `--`) represent single operators. The key to learning their meaning is just to take the left-to-right order of macro name and operator literally. What comes first is done first.

What is more, these operators apply only to local macros and nothing else. After `global i = 1`, `display $i++` is illegal. `display "$i++"` is perfectly legal but does nothing special: you just get shown `1++`, which might even be something you want to print. But, on reflection, you will see that this restriction is no restriction, as once you can do it with local macros, you can indirectly do it with other objects.

3.3 Automated forvalues with arranged stepping through a varlist

Our first way of tackling this problem was to cycle through the variables with `foreach` and to ensure a cycle in step through an integer sequence. The second way is the reverse of this. The loop then starts with the idea

```
. forvalues i = 1/12 {
.         ...
. }
```

We therefore need a way to select each variable in turn, and, in fact, there are several ways. The overriding desire in practice is to type as little as possible: after all, it is easy but tedious to do in 12 statements. At the same time, it helps to know lots of tricks, even if some of them do not really help much until we have to tackle more complicated problems.

```
. forvalues i = 1/12 {
.         local v : word `i' of jan feb mar apr may jun jul aug sep oct nov dec
.         rename `v' m`i'
. }
```

or, if you like,

```
. local vars "jan feb mar apr may jun jul aug sep oct nov dec"
. forvalues i = 1/12 {
.         local v : word `i' of `vars'
.         rename `v' m`i'
. }
```

The construct `: word` # `of` takes the specified word from the string to its right and puts it into what is named to its left, which in practice is usually a local macro. Words are delimited by spaces, except that `" "` bind tighter than spaces can separate. In the string

```
Stata statistical software
```

there are, as you would expect, 3 words, whereas in the string

```
Stata "data analysis" software
```

there are also 3 words, in this sense. In the string

```
1 2 3
```

there are also three words, as the word is being used in a programming sense, not a linguistic one. So, as you will guess, each time around the loop we pick the `i`th word from the string specified. Whether we specify the list of months outside or inside is not very important. I prefer the outside, largely because, if this is code you revisit some time later, it will be just a little clearer what is going on.

That is some improvement, but we would like to be more concise still. One specific trick is to use `ds` (see [R] **describe**) to expand the *varlist* for you:

```
. ds jan-dec
. forvalues i = 1/12 {
.         local v : word `i' of `r(varlist)'
.         rename `v' m`i'
. }
```

This use of `ds` is not completely general. What we are relying on here is a side-effect. `ds` will expand the *varlist* supplied but also leave behind that list in `r(varlist)`. `ds` is an example of an r-class command, which leaves behind results with names of the form `r(`*result*`)`. For an introduction, see [U] **21.8 Accessing results calculated by other programs**.

By the way, extra features were added to `ds` in June 2003. If your Stata 8 is earlier than this, you will need to `update` it to see what they are.

r-class commands leave results in memory until the next r-class command is executed, or sometimes earlier, so they should be regarded as ephemeral. So you need to be aware that, in some other block of code, you could lose `r(varlist)` to a command executed before you use it. Here is a more general structure:

```
. ds jan-dec
. local vars "‘r(varlist)’"
. forvalues i = 1/12 {
.         local v : word ‘i’ of ‘vars’
.         rename ‘v’ m‘i’
. }
```

The difference is that we copy the contents of `r(varlist)` somewhere safe before some other command either destroys or overwrites it. If your interest inclines more to programming, also check out `unab`, documented at [P] **unab**, as another way of unabbreviating a *varlist*.

Once more, you could have done that in Stata 7, but Stata 8 has introduced a way of doing it more concisely. Here is an example:

```
. ds jan-dec
. local vars "‘r(varlist)’"
. forvalues i = 1/12 {
.         rename ‘: word ‘i’ of ‘vars’’ m‘i’
. }
```

We have lost one line putting `word ‘i’ of ‘vars’` into `local i`. You will probably guess that what we have here is another feature whereby a macro reference is actually an in-line computation. When faced with nested macro delimiters, `‘ ’` within `‘ ’`, the rule for Stata is like that for parentheses in elementary algebra: evaluate the innermost expression first and work from the inside outwards (and work also from left to right). Let us dissect

```
‘: word ‘i’ of ‘vars’’
```

the way that Stata would. Strip the outermost delimiters

```
‘                     ’
```

and discover within the expression

```
: word ‘i’ of ‘vars’
```

Substitute the current values of local macros `i` and `vars`, getting, say,

```
: word 1 of jan feb mar apr may jun jul aug sep oct nov dec
```

Finally, instead of putting the results of that expression (which will be the single variable name `jan`) in a local macro, as would commonly be done, just place the result exactly where the instruction was found.

I want to interject here two small pieces of advice. It is natural to panic a little on meeting this feature and to see only a confusing little snowstorm of delimiters, but, as implied, it should be no more confusing than dealing with parentheses, brackets, or braces in algebra. (Remember, innermost first!) Also, choose a font within Stata, or within your text editor, that differentiates clearly between left single quotes and right single quotes. Stata's own fonts are, as you might guess, one set of possibilities.

There is another way to do this that we should cover before leaving this example.

```
. tokenize jan feb mar apr may jun jul aug sep oct nov dec
. forvalues i = 1/12 {
.         rename ``i'' m`i'
. }
```

`tokenize` is yet another command from Stata programming that can be used interactively. See [P] **tokenize**. It takes its argument, splits it into tokens, and assigns those tokens to local macros `1` and up, according to how many tokens there are. By default, tokens are just words in the Stata sense used earlier. (They are more general than words, as they can be delimited by something other than spaces, but that is a different story for now.) So, after this `tokenize`, local macro `1` will contain `"jan"`, local macro `2` will contain `"feb"`, and so on, all the way up to local macro `12`, which will contain `"dec"`. The effect of `tokenize` is here equivalent to 12 statements: `local 1 "jan"`, `local 2 "feb"`, all the way up to `local 12 "dec"`.

The body of the loop is just the single command `rename ``i'' m`i'`. The nested quotes around the first argument are no worry: the "innermost first" rule is the thread that lets you traverse the maze. The first time around the loop, the local macro `i`, which is controlled by the `forvalues` command, contains `1`, so we substitute for that

```
rename `1' m1
```

leaving just local macro `1` to substitute, giving

```
rename jan m1
```

As `forvalues` cycles through 1 through 12, this logic yields commands all the way up to `rename dec m12`.

We have squeezed almost all the juice out of this example. But before we leave it, note that we can put together the best features of the last two solutions:

```
. ds jan-dec
. tokenize `r(varlist)'
. forvalues i = 1/12 {
.         rename ``i'' m`i'
. }
```

3.4 Summary of tools and tricks

A summary was promised of the main tools and tricks available for this example. The example may have seemed trivial, but it has allowed us to introduce most of the devices that are useful in practice.

1. You can cycle through a *varlist* with `foreach` and arrange that you cycle through an integer sequence in harmony. You can increment in the long-winded way `local i = ‘i’ + 1` or in the new, short-winded way with a reference to `‘i++’`.

2. You can cycle through an integer sequence with `forvalues` and arrange that you cycle through a *varlist* in harmony. Knowing that `ds` leaves behind an unabbreviated *varlist* can save a lot of typing. For selecting a variable name, use `word` # `of` or ensure through `tokenize` that what you want lies within local macros from `1` up.

3.5 A second example

Let us turn now to a fresh example, in which we can use these ideas but in which some other elements are also needed.

In his interesting survey *Statistical Rules of Thumb*, van Belle (2002, 107–108) refers to (mean − median)/standard deviation as a measure of skewness, noting that it lies between −1 and 1 (for proof, see Hotelling and Solomons 1932). This measure was new to me, and I wondered how it compared with other measures, such as the moment-based skewness measure, as calculated by `summarize`, or

$$\frac{(\text{upper quartile } - \text{ median}) - (\text{median } - \text{ lower quartile})}{\text{upper quartile } - \text{ lower quartile}} \\ = \frac{\text{upper quartile } - 2\times\text{median} + \text{ lower quartile}}{\text{upper quartile } - \text{ lower quartile}}$$

which also lies between −1 and 1. To illustrate the approach, we need only show how to pick up results for the first two measures.

The first step is to see that we can obtain the elements of this unfamiliar measure as results accessible after `summarize`, so that we could go

```
. foreach v of var varlist {
.         quietly summarize ‘v’, detail
.         di "‘v’ {col 33}" %6.3f ‘r(skewness)’ "{col 44}" ///
               %6.3f (‘r(mean)’ - ‘r(p50)’) / ‘r(sd)’
. }
```

All this does is to put out a list of variable names and results for familiar and unfamiliar measures of skewness. We have paid some attention to layout (using SMCL `col` directives, as explained at [P] **smcl**) and to format, but otherwise the results are not put anywhere except the monitor. Further examination of the results will usually be much easier if they are placed in new variables. At first sight this is a little awkward, as we have one pair of results for each variable, so how will they fit alongside the existing dataset? As long as we have more observations than variables, we just compile new variables by the side of, and unrelated to, the existing dataset. (If we have more variables than observations, we need to increase the number of observations or put the results elsewhere. Let us continue with the simpler optimistic assumption.)

This little problem must be solved backwards. We want each new skewness result to be put into a separate observation, which in turn implies that we need to loop over

```
replace ... = ...  in  observation
```

and that itself requires a `generate` before the loop is executed. (No `replace` can be undertaken without a previous `generate`.) Jumping to a solution,

```
. generate varname = ""
. generate vanbelle = .
. generate skewness = .
. local i = 1
. quietly foreach v of var varlist {
.         summarize `v', detail
.         replace varname = "`v'" in `i'
.         replace vanbelle = (`r(mean)' - `r(p50)') / `r(sd)' in `i'
.         replace skewness = `r(skewness)' in `i++'
. }
```

Some details here deserve spelling out. We need to initialize each variable we are going to use. The recording of variable names here is rather more than an adornment. As these values bear no relation to the structure of the existing dataset, the very first `sort` after this will destroy all links to the present order of the variables, so we should record variable names explicitly. We also need to initialize a local macro, here `i` to `1`, so that we can step through the observations. After we put all the results for a particular variable in a particular observation, this can be incremented. For the names of particular r-class results, you will need to refer to the manual or (what is sometimes quicker) work them out by examining a `return list`.

Once that is done, a graph can be produced

```
. scatter vanbelle skewness, mlabel(varname)
```

which, naturally, you may inspect for your favorite dataset.

This example should not be left before we grasp one awkward detail. Suppose that a variable were actually constant. In this situation, the standard deviation is necessarily zero, and the measure discussed by van Belle therefore is indeterminate. That may seem appropriate if you are not inclined to discuss the skewness of a constant. Another approach is to insist that if the variable is a constant, the mean and median are necessarily equal and the measure thus necessarily zero. Indeed, whenever the mean and median are equal, the measure is necessarily zero. In Stata terms, a more careful coding for that recipe would be

```
cond((`r(mean)' == `r(p50)') & (`r(mean)' < .), 0,
        (`r(mean)' - `r(p50)') / `r(sd)')
```

where we have trapped the case of missing means or medians.

The kind of approach described here may be compared to that in cookery or do-it-yourself programs. We all know that expert cooks sometimes make mistakes just like everybody else, except that they presumably do so less often than most others. What you see, however, lacks all the gaffes, small or large, which were edited out before publication. Similarly, this example was something I really wanted to do, but I did not get all the details right the first time. How you do it is up to you, but working your way up from the first beginnings to a more complete script in a text editor is an approach that appeals to many Stata users.

3.6 Three or more lists in parallel

What about processing three or more lists in parallel? In essence, there are no new ideas needed, and the story is just more of the same. `foreach` or `forvalues` will give you automated stepping through one list; any others are your own responsibility. In most cases, there will be some structure to the problem that you can and should exploit as fully as you can.

One word of caution: `tokenize` cannot be applied to yield two or more lists of numbered macros. Each `tokenize` overwrites the results of any previous `tokenize`, or for that matter, any other assignment to local macros with names 1 and up. This points up the value of using `word` # `of`.

4 Macro lists

One new feature in Stata 8 that you may have missed among the larger features is documented dryly under the rather cryptic heading [P] **macro lists**. The online help is accessible under `macrolists`. In essence, this is a set of new utilities designed for manipulating lists held in macros (either local or global). The loss of generality here is negligible, as it is not difficult to get other kinds of lists into macros. The utilities take as arguments either one or two macro names, work on the contents of the named macros and, as appropriate, place their results in a named macro or return a number (0 for false or 1 for true), a size, or a location. Examples are

```
. local sorted : list sort tobesorted
```

which, not surprisingly, sorts a list; and

```
. local included : list A in B
```

which returns 1 if the elements of A are all included in B, and 0 otherwise. (Often, there is just one element in A; that is a useful special case.)

The introduction of these extra utilities and other syntax elements in Stata 8 greatly enriches the facilities for list manipulation. Even when some operation is not provided as a language primitive, it is typically a step or two away.

Want a list not of successive integers, but of powers of 10?

```
. forvalues i = 1/6 {
.         local j = 10^`i'
.         local newlist "`newlist'`j' "
. }
```

or

```
. forvalues i = 1/6 {
.         local newlist "`newlist'`=10^`i'' "
. }
```

Note the trailing space. Want to reverse a list, say the one in local macro `mylist`?

```
. forvalues i = `: list sizeof mylist'(-1)1 {
.         local newlist "`newlist'`: word `i' of `mylist'' "
. }
```

or

```
. tokenize `mylist'
. forvalues i = `: list sizeof mylist'(-1)1 {
.         local newlist "`newlist'``i'' "
. }
```

Want to replicate a list, as a block, three times?

```
. forvalues i = 1/3 {
.         local newlist "`newlist'`mylist' "
. }
```

In fairness, note that

```
. local newlist : display _dup(3) "`mylist' "
```

has been possible for some time in Stata. Finally, want to replicate a list, repeated elements adjacent, three times?

```
. tokenize `mylist'
. forvalues i = 1/`: list sizeof mylist' {
.         local `i' : display _dup(3) "``i'' "
.         local newlist "`newlist'``i'' "
. }
```

Naturally, if the list is very short, you could just type out the solutions.

5 levels and the problems it solves

A new command, `levels`, was added to Stata 8 on 16 April 2003. `levels` displays a sorted list of the distinct values of a variable taking on integer or string values. To put it another way, `levels` displays the levels of a categorical variable, very broadly so defined. Its name echoes the terminology of levels of a factor long standard in experimental design (see, for example, Cochran and Cox 1957, 148; Fisher 1942; or

Yates 1937, 5) and, incidentally, the name of commands with broadly similar intent in some other statistical software. As the date implies, you may need to `update` your copy of Stata 8 to use `levels`. In addition, as its release follows the original release of Stata 8, `levels` is not documented in the manuals.

Let us explain precisely what is meant here by a variable taking on integer values. What is crucial is the content of a variable, not its storage type; thus, if a `float` variable contained the distinct values 1, 2, 3, 4, and 5, those are its levels. But a `float` variable containing 3.14159 is beyond the reach of `levels`.

`levels` serves two different functions. Occasionally, it serves to give a compact display of distinct values. More commonly, it can be useful when you desire to cycle through the distinct values of a variable, particularly with `foreach`. What makes this easy is that `levels` leaves behind a list in `r(levels)`, which may be used in a subsequent command. Alternatively, the list may be assigned to a local macro. One key advantage of that, to repeat a point made earlier, is that r-class results are ephemeral. Putting a copy of those results in a local macro of your choice is good practice; at least, if you overwrite it, then that will be your fault. Either way, the key point is that `levels` automates the making of a list, which means that you never have to specify a list of values directly.

A common pattern is repeating some analysis for each distinct group of some classifying variable, such as a categorical variable in the strict sense or the different members of a panel. Sometimes such repetition is possible directly through `by:` (Cox 2002a) or a `by()` option, but often it is not. Even when `by:` is supported, stepping through group by group may be preferred. For example, many graphs done `by()` are subdivided into several small panels; a portfolio of graphs produced one at a time is sometimes more convenient.

A first template might look something like

```
. levels factor, local(levels)
. foreach l of local levels {
.         di "-> factor = `l'"
.         whatever if factor == `l'
. }
```

This is, as said, a template, and several commands may be executed for each distinct level. Indeed, once the basic structure of the `foreach` loop has been worked out, it is usually easy to elaborate. In contrast, multiple commands executed under the aegis of `for` often become unreadable and unmanageable.

Let us focus on the role of `levels` in facilitating this loop. First, `levels` examines the variable *factor* and puts a list of its distinct levels in the local macro `levels`. The name is yours to choose; this is merely one possible idiom. You will at the same time get a display of the levels, and you might want to see if that fits your idea of what is going on. Second, `foreach` looks for its argument inside the local macro we just created. Clearly, you need not work out for yourself what the distinct levels are, let alone type them out.

A more general point is that `levels` just records what is present in the data, which is trite but can be useful. Consider two kinds of little problems. One is that the possible levels of some variable are known, more or less, but there remains a possibility of gaps. If a key variable is number in household, we know we can cycle from 1 upwards, but what about the high end? Even in a very large dataset, we might get some households of 11 and some of 13, say, but none of 12. At worst, cases that do not exist might crash some code. There are other ways to tackle this problem, but `levels` is an easy solution.

The other kind of little problem is that your levels may be essentially unstructured. Many organizations assign multi-digit identifiers on some kind of logic of their own, but users see in their datasets only irregular sequences. Typing out hundreds or thousands of these identifiers is an extremely unattractive prospect.

Arranging a display that informs the user which group is being analyzed seems an obvious detail, but it is very easy to overlook. Unless the number of groups is very small and the patterns are known to be distinct, it is all too common to lose track of which results you are examining; at worst, there can be weeping, wailing, and gnashing of teeth later when groups of results are found to be useless because all are unlabeled. In the case of printed results, `display` can be used, as above, and indeed, bold and attractive titles can be quickly produced with the aid of a few SMCL directives. In the case of a graph, one natural possibility is to make use of a small but distinct title option, such as `subtitle()`, `caption()`, or `source()`.

A questioner on Statalist wanted separate line graphs for the temperatures recorded for each patient in a panel. This is as easy as it should be:

```
. levels id, local(levels)
. foreach l of local levels {
.         line temp time if id == ‘l’, subtitle(id is ‘l’)
.         more
. }
```

What `more` does is to hold the graph until the user has finished with it. On occasion, especially when the graphs are being saved to files, you might not want even to look at them while they are being produced; `more` can then be suppressed. Another small detail is that this example assumes numeric identifiers. If the identifiers were string, the appropriate test would clearly be `if id == "‘l’"`.

Having said all that, there is another solution to this problem and one that is in fact more general, based on the function `egen, group()`. See [R] **egen**. This function maps the distinct groups of the *varlist* supplied to integers 1 and up. Not only can it be used for groups defined by several variables, but it can be used with any kind of variable, including those with numeric values with fractional parts. Thus, once you have found out the maximum value of the group variable generated, you can use `forvalues` to cycle over the groups so defined. Given this, why introduce `levels`? The rationale is perhaps twofold. First, although `egen, group()` has been in Stata for some time, users have often overlooked its use for problems for this kind. Second, `levels` has a certain directness as a tool designed for its purpose.

6 Summary

This column started with what will have been a piece of bad news to those who are fond of `for` (although my argument would be that most such fondness was misplaced). But I hope that bad news has been more than balanced by useful information about the newer or more esoteric devices surveyed, especially those for handling lists in parallel or for easily going through all the members of a list. As in previous columns, the tools needed for interactive solutions of various list problems have typically been borrowed from the repertoire of Stata programming. Familiarizing yourself with them is a good way to improve your Stata skills and cut down on tedious and error-prone typing of a long list of nearly identical commands.

7 What's next

Tables are one of the staple products of data analysis, and Stata provides many general-purpose and special-purpose tabulation commands. We will look at some problems that appear to fall between the gaps and what can be done about them.

8 Acknowledgments

Kit Baum and Nicholas Winter contributed to the development of `levels`.

9 References

Cochran, W. G. and G. M. Cox. 1957. *Experimental Design*. New York: John Wiley & Sons.

Cox, N. J. 2002a. Speaking Stata: How to move step by: step. *Stata Journal* 2(1): 86–102.

—. 2002b. Speaking Stata: How to face lists with fortitude. *Stata Journal* 2(2): 202–222.

Cox, N. J. and J. Weesie. 2001. dm88: Renaming variables, multiply and systematically. *Stata Technical Bulletin* 60: 4–6. In *Stata Technical Bulletin Reprints*, vol. 10, 41–44. College Station, TX: Stata Press.

Fisher, R. A. 1942. The theory of confounding in factorial experiments in relation to the theory of groups. *Annals of Eugenics* 11: 341–353.

Hotelling, H. and L. M. Solomons. 1932. The limits of a measure of skewness. *Annals of Mathematical Statistics* 3: 141–142.

van Belle, G. 2002. *Statistical Rules of Thumb*. New York: John Wiley & Sons.

Yates, F. 1937. *The Design and Analysis of Factorial Experiments*. Harpenden: Imperial Bureau of Soil Science Technical Communication 35.

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 twelve commands in official Stata. He was an author of several inserts in the *Stata Technical Bulletin* and is executive editor of *The Stata Journal*.