The Stata Journal (2002)
2, Number 1, pp. 86–102

Speaking Stata: How to move step by: step

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

Abstract. The `by` *varlist*`:` construct is reviewed, showing how it can be used to tackle a variety of problems with group structure. These range from simply doing some calculations for each of several groups of observations to doing more advanced manipulations making use of the fact that with this construct, subscripts and the built-ins `_n` and `_N` are all interpreted within groups. A fairly complete tutorial on numerical evaluation of true and false conditions is included.

Keywords: pr0004, by, sorting, subscripts, true and false, _n, _N

1 Divide and conquer

This column focuses on the `by` *varlist*`:` construct in Stata. `by:`, as I will call it for brevity, is one of Stata's most distinctive and powerful features, but also one that many users find hard to grasp. It is not so much that `by:` is especially difficult to understand. Rather, the challenge is mainly in absorbing some of the important details, in recognizing when a problem calls for `by:`, and in thinking your way towards the few lines of Stata that then solve that problem. In addition, the Stata documentation for `by:` is in several short sections scattered around the manuals, at [U] **14.1.2 by varlist:**, [U] **14.5 by varlist: construct**, [U] **16.7 Explicit subscripting**, [U] **31.2 The by construct**, and [R] **by**. Here I attempt to bring all the main details together and to provide a good range of illustrations. As an extra, I will also give a fairly complete run-down on how Stata handles true and false conditions. This is often very useful with `by:`, and elsewhere in Stata.

The big idea behind `by:` is just this: do something separately for each group of observations defined by a specified variable list (*varlist*). For example, imagine that you have read in Stata's `auto.dta`. On your machine, that dataset will be in whatever directory or folder the `sysdir` command tells you pertains to `STATA`. If you typed at the Stata prompt something of the form

```
. by foreign: command
```

that *command* would be executed separately for each group of observations defined by the variable `foreign`. With these data, typing `tabulate foreign` tells you that there are two categories of that variable. As value labels have been attached to `foreign`, those categories are shown as `Domestic` and `Foreign`, indicating the origin of the 74 cars described in the `auto` dataset, whether domestic, made in the United States, or foreign, made elsewhere. If *command* were (say) `summarize mpg`, then the values of `mpg` would be `summarize`d, first for observations for which `foreign` has the value label `Domestic`, and then for those with the value label `Foreign`.

In short, `by:` subdivides a composite task—to do something for each of several groups—into its component tasks. It is a matter of divide and conquer. Note that the command introduced with `by` *varlist*`:` both defines the problem to Stata and instructs Stata to carry out the command for each group. We are here avoiding the need to set up, for ourselves, any kind of loop over the different groups defined by *varlist*. Users who come to Stata with some programming background, or with experience of other statistical software, often are predisposed to think about writing a miniature program to cycle through groups, but `by:` allows, as it were, one kind of "programming without programming". I said more about related ideas in my column in the previous issue; see Cox (2001).

Before proceeding to the details, let us note that `by:` will not work with all Stata commands. Look at the help on any command to see whether `by:` will work with that command. Even if it does not, on occasion the same functionality may be implemented as a `by()` option.

2 Sort first

Before you can issue a command prefixed with `by` *varlist*`:`, you must, however, first `sort` the data in memory according to *varlist*. If *varlist* were a single numeric variable, then sorting would be by numeric order: lowest first, and highest last. Note that numeric missing is always treated as higher than any nonmissing numeric value. If *varlist* were a single string variable, then sorting would be by alphanumeric order, more precisely ASCII order; for example, missing (empty) strings go before numeric strings such as `"1"`, which go before lowercase letters such as `"a"`, and which in turn go before uppercase letters such as `"A"`. See [R] **sort** if desired.

If the data are already sorted by *varlist*, and Stata knows this to be the case, then the `sort` is unnecessary. The first part of that condition should be clear: if the data are already sorted, then the `sort` should indeed be unnecessary. But what is meant by the second part? Suppose that you type in a dataset ordered by some variable: to you they are now sorted by that variable. That's not enough to Stata: it needs to do a `sort` by itself, and then it knows that data are indeed sorted, even if that `sort` doesn't change the order of the observations. The easiest way to read Stata's mind on this point is to look at the bottom of the output issued by the `describe` command, which tells you whenever Stata thinks that the data are sorted by a particular *varlist*. In practice, however, if data need to be sorted in a particular way, it is often fastest just to do that explicitly. It is rare indeed that the machine time thus spent exceeds the time checking manually whether data are sorted and by what *varlist*.

Whatever the variable(s) being sorted, the most important result is that observations with identical values end up next to each other. That is guaranteed, as a matter of definition, by `sort`. Thus, `sort foreign` implies that all the observations with value 0 will be next to each other at the start of the dataset, and all those with value 1 will be next to each other at the end of the dataset. To know that `foreign` takes on values 0 and 1, we need to look underneath the value labels to see the actual numeric values.

In the auto data, we can do this by `tabulate foreign, nolabel`, or by seeing from `describe` that `foreign` has value labels `origin` attached and then looking at the label definition with `label list origin`.

Consider this experiment with the auto data:

```
. sort foreign
. sort mpg
. sort foreign
```

Will the data be in exactly the same order after the third `sort` as after the first `sort`? The answer is that it is possible, but that you should never assume that this will be the case. In general, it is highly unlikely. Once the data have been sorted by `mpg`, Stata forgets the particular order of observations that it obtained previously. As far as it is concerned, any order of observations in which all the values of 0 for `foreign` precede all the values of 1 is an acceptable solution, and naturally there are many such solutions. (To be precise, there are 52! 22! solutions given 52 values of 0 and 22 values of 1, and that's a lot.) This detail can be crucial in a few problems concerning `by:`, and on occasion you must make sure that it does not bite you. To epitomize the problem in two sentences: Only if all observations are unique with respect to *varlist* does `sort` *varlist* produce exactly the same order of observations every time. If there are any observations identical with respect to *varlist*, then the exact order of observations after `sort` *varlist* may well differ.

Before Stata 7, the need to `sort` first meant that every such problem was at least a two-step

```
. sort foreign
. by foreign: command
```

but in Stata 7 there is now a way to do this as a single step,

```
. bysort foreign: command
```

which can be thought of as a telescoping of the two commands above. `bys` is an acceptable abbreviation for `bysort`. `by foreign, sort:` *command* is also legal syntax. The choice is a matter of taste.

The next step in complexity is to a very common situation. Concretely, consider panel or longitudinal data in which observations are arranged first by some kind of panel identifier and second by time of recording. In biostatistics, observations on visits of patients will normally carry patient identifier and date of visit as key variables. Many people have data like these, even if they don't use the term "panel". Data management and data analysis problems often then require some work done separately for each patient, but respecting time order (say, working with the last visit for each patient). Before Stata 7, the preparatory sorting would have been

```
. sort patientid date
. by patientid: command
```

but in Stata 7 there is now a way to do this in a single step:

```
. bysort patientid (date): command
```

Note the syntactic nuance of `by` *varname1* `(`*varname2*`)`: this means that *command* will be executed separately for groups of *varname1*, but that each such group will be sorted first according to *varname2*.

From now on in this column, the syntax used will be that of Stata 7. Readers using earlier versions are thus requested to translate `bysort` commands into the older and more long-winded syntax.

3 varlist can have one or more variables

All the examples so far have been of `by:` with a single variable, but if you didn't know, then the use of the general jargon *varlist* and the last point made about panel data would probably lead you to suspect that what can be done with one variable can as easily be done with two or more variables. Indeed, that is one of the cardinal virtues of `by:`. It is essentially no more difficult to tackle problems in which groups of observations are defined by several variables than it is to tackle those in which they are defined by one variable. Programmers in any language will especially appreciate this, as in most other circumstances there is a prospect of setting up multiple loops, which is at least tedious and error-prone.

Let's imagine that in the auto data we want to issue

```
. by foreign rep78:  command
```

That is, groups of observations are defined jointly by two variables `foreign` and `rep78`. The situation here is perhaps best shown, a little indirectly, by looking at the results of

```
. contract foreign rep78
```

`contract` (see [R] **contract**) produces by default a condensed dataset consisting of all those combinations of its *varlist* that exist in the data, together with a new variable `_freq` showing the number of observations in each combination. Let us list the results:

```
. list foreign rep78 _freq
     foreign     rep78         _freq
  1. Domestic         1             2
  2. Domestic         2             8
  3. Domestic         3            27
  4. Domestic         4             9
  5. Domestic         5             2
  6. Domestic         .             4
  7.  Foreign         3             3
  8.  Foreign         4             9
  9.  Foreign         5             9
 10.  Foreign         .             1
```

We see that in practice there are two categories of `foreign` and six of `rep78` (the missing category can easily be overlooked). However, of the twelve possible combinations, only ten actually occur. When we `sort` by `foreign rep78`, first all observations are sorted according to the value of `foreign`, just as previously discussed. Then all observations

with each value of `foreign` are sorted according to their values of `rep78`. Given three or more variables, such a process would be repeated, breaking ties where possible by referring to variables mentioned later in the *varlist*. Note that the order in which variables are named in the *varlist* is crucial for determining the precise sort order. Thus, the results of `sort foreign rep78` differ from those of `sort rep78 foreign`. The same combinations of categories will appear, but in a different order.

Assuming that the auto data are read in once more, now we can do something for all the combinations of the two variables:

```
. bysort foreign rep78: command
```

It is really the same idea as before, but one new detail is worth emphasis. Stata just silently ignores cross-combinations of categories not represented in the data. (Remember that only ten out of twelve cross-combinations of `foreign` and `rep78` are actually present in the auto dataset.) Usually, that is a feature. If you use `by:` and results appear to be incomplete, then this will usually be the explanation. Of course, other limitations can bite, such as when some minimum number of observations is needed on mathematical or statistical grounds for a calculation. Thus, for example, two observations distinct on both variables are the minimum for a correlation coefficient to be calculated, and even then the result necessarily has magnitude 1. On occasion with `by:`, the option `rc0` can be useful for ensuring that Stata continues regardless of failure for any one group of observations. `rc0` stands for `r`eturn `c`ode `0`, which Stata habitually emits when a command has executed without problems.

4 The built-ins _n and _N

You already know enough to use `by:` effectively. Most uses of `by:` are no more than sorting the data into groups and then doing something separately for each group, and with `bysort` that can be done in one command. What follows are the slightly more advanced tricks possible with this construct. They are the means for solving a surprisingly large number of problems in Stata whenever observations have group structure.

The key to many such tricks lies in Stata's built-in variables, `_n` and `_N`. If no `by:` is in sight, then `_n` is just the observation number: 1 for the first observation, 2 for the second, and so on. `_n` depends on the current sort order of the data. This alone can be useful: suppose you read in some data that do not possess an identifier. `generate id = _n` assigns identifiers 1 up, and given that, `sort id` will bring the data back to the original order.

Similarly, if no `by:` is in sight, then `_N` is the total number of observations in memory. At a given time this is one number: for the auto data as supplied with Stata, it is 74. But `_N` is also variable, so if observations are added or dropped, it will necessarily change.

Now comes the twist that gives `by:` extra spin. Under the aegis of `by` *varlist*`:`, `_n` and `_N` are interpreted as observation number (1 up) and total number of observations *within each group of observations defined by varlist*. That is, they are always interpreted with reference to the current group of observations, rather than the entire dataset.

To see this, here is a simple example. Suppose that with those panel data on patient visits we want to `generate` a variable that records the number of visits for each patient.

```
. bysort patientid: generate int nvisits = _N
```

The main idea is that because `generate` is executed under `by patientid:`, `_N` is the number of observations in each group, namely, the number of visits for each patient. It is clear that we do not need to `sort` each patient's observations by date to count the number of visits. Even if they are not in date order, we will get the same counts. However, `bysort patientid (date):` would do no harm and would produce the same results. In other problems, not surprisingly, getting each group of observations in date order will be crucial for getting the correct results.

By the way, `int` indicating variable type is just a flourish here, as a reminder that the variable will have modest integer values, so that there is a small advantage in storing them appropriately. Its use also presupposes, as we would certainly hope, that the number of visits does not exceed 32,766 for any patient.

Within each group there are two special observations, the first and the last. In calling these observations "special", I am not thinking primarily of substantive meaning, even though first and last state may be of particular scientific or practical interest, but rather of two details that can be useful in writing code.

First, each group of observations evidently has at least one member. (Recall that Stata unobtrusively skips, on our behalf, those logically possible cross-combinations of two or more variables that do not occur in the dataset in memory; that is, those combinations for which the number of members is zero.) Thus, if we wish to do something once (and/or systematically) for each group, then we could do it *either* for the first observation in the group *or* for the last observation in the group. With singleton groups, those for which `_N` is 1, the first and the last will clearly be the same observation. To put this another way, it always makes sense to refer to the first observation in each group (Stata idiom will be `_n == 1`) or to the last observation in each group (Stata idiom `_n == _N`). The double equal sign `==` is used in Stata whenever you wish to test for equality: compare the use of the single equal sign `=` for assignment.

Second, we will often want to count something, or more generally to calculate some sum, over all the observations in each group. The natural Stata way to do this will be to produce a cumulative or running `sum()` within each group and then to pick up its last value as the result we want.

With panel-like data in particular, we may want to make reference to changes, say, in some measure like blood pressure `bp` since some previous state. For that it is essential that observations are sorted properly, not only between but also within groups, as by `sort patientid date`. In doing this, a nuance is worth noting. There is no observation before the first and none after the last observation. It may be that in computer memory the last observation for one patient is stored just before the first observation for another, but Stata automatically acts logically and ignores that.

What are the changes between successive values for each patient?

```
. bysort patient (date): generate chbp = bp - bp[_n-1]
```

`bp` is the current blood pressure (we could validly say `bp[_n]`, but there is no need to do so), while `bp[_n-1]`—for the current observation number `_n`, minus 1—is the previously measured blood pressure, so we just subtract that. As another expression of how `by:` works, note that the subscript, here `[_n-1]`, is always interpreted within groups. For more on subscripts, see [U] **16.7 Explicit subscripting**.

In problems such as these, a mental check on boundary conditions is often a good idea, at least until you know each idiom so well that you know what is and what is not problematic. First check what happens with the first in each group, here `bp[1]`. Its previous value, `bp[0]`, does not exist in the data. In fact, Stata acts as if `bp[0]` were missing. Those of a philosophical or theological persuasion can ponder the subtleties here, but the practical result is the same. As a consequence, `chbp[1]`, which is `bp[1] - bp[0]`, will be returned as missing. Then check what happens with the last in each group. In this case, the previous value will always be present, so long as there are at least two observations in each group.

What is the difference between the first value and the last value for each patient?

```
. by patient: generate totchbp = bp[_N] - bp[1]
```

As the data are already in the correct sort order as a result of the previous `bysort`, `by` is sufficient here. Once more, this example carries the message that under `by:` subscripts are always interpreted within each group, so that `bp[1]` is the first value for each group and `bp[_N]` is the last.

Here again, if we think about boundary conditions, one small problem can be identified with singleton groups, for each of which `_N` is 1. For such groups the first and the last are necessarily the same observation, and the difference is, thus, necessarily 0 (unless missing minus missing, which in Stata is missing). We might decide to exclude all such singletons from the calculation by

```
. by patient: generate totchbp = bp[_N] - bp[1] if _N > 1
```

Here now is a more involved but quite manageable problem: count the numbers of increases and the numbers of decreases between successive measurements in blood pressure for each patient. Depending partly on the resolution of measurement, some pairs of measurements will appear identical: let's arbitrarily lump the instances of no change with the increases. A first idea is that we count increases or no changes by counting how often

```
bp >= bp[_n-1]
```

and similarly that we count decreases by counting how often

```
bp < bp[_n-1]
```

As said, `bp[0]` is not present in the data, or what is equivalent in practice, we can also say that `bp[0]` is always evaluated as missing. So, we certainly do not want to count all

those occasions on which `bp[1] < bp[0]`—which evaluates to `bp[1] < .`—as if they were real (physiological) decreases. In fact, we might as well bundle that boundary problem with all the other cases in which one or other measurement is missing. So, we will put this to Stata as producing two count variables:

```
. by patient: generate int ninc = sum(bp>=bp[_n-1] & bp<. & bp[_n-1]<.)
. by patient: generate int ndec = sum(bp<bp[_n-1] & bp<. & bp[_n-1]<.)
. by patient: replace ninc = ninc[_N]
. by patient: replace ndec = ndec[_N]
```

Let's unpack that more slowly. In the first command, the event of interest is that blood pressure did not decrease—`bp>=bp[_n-1]`—*and* that the current blood pressure is not missing—`bp<.`—*and* that the previous blood pressure is not missing—`bp[_n-1]<.`. (Learn to save yourself keystrokes by noting that `< .` is equivalent to `~= .`, a fine discrimination signaled by Lachenbruch (1992)). If all these conditions are true, the numerical result of the Boolean expression

```
bp>=bp[_n-1] & bp<. & bp[_n-1]<.
```

for each observation is 1; if even one condition is false, the numerical result is 0. (In the next section, we will look in much more detail at the numerical evaluation of true and false conditions.) The second command has a similar flavor, except that we are counting decreases. In both commands we get the cumulative or running sum by using the `sum()` function. The last value produced by that function is the one we want, and the last value we know how to get, by specifying `_N` as subscript.

Stata is operating here just as you might if given several apples and oranges and asked to count first the apples, and then the oranges. Given apple, apple, orange, orange, apple you could sum $1 + 1 + 0 + 0 + 1$ apples $= 3$ apples and $0 + 0 + 1 + 1 + 0$ oranges $= 2$ oranges. Think of this as producing running tallies of 1 2 2 2 3 apples and 0 0 1 2 2 oranges seen so far, and reading off just the last tally. You are thinking just as Stata does given the commands above. It is no surprise that counting problems can be reduced to summing 1s and 0s (or summing 1s, as the 0s make no difference).

It turns out that we could have done all this in one fell swoop with `egen`, as for example:

```
. by patient: egen int ninc = sum(bp>=bp[_n-1] & bp<. & bp[_n-1]<.)
```

However, we save only one command, and it is always worth knowing the underlying Stata thinking. We need to be prepared for all those problems for which no one has previously written a program offering a canned solution. In passing, note that, confusingly, `egen`'s `sum()` always gives final sums, not cumulative or running sums.

5 True and false in Stata

We will now pick up on an idea just mentioned, that true and false conditions have numerical equivalents. This material is useful in many applications of Stata, not only for `by:` but more broadly, so we will look at the area in some detail.

Most computer languages have some way of indicating and working with what is true and what is false, but not all languages choose exactly the same way. Stata follows two rules, the second of which may be considered as a generalization of the first. I will state the rules, and then we will look at each in turn.

- *Rule TF1: Logical or Boolean expressions evaluate to 0 if false, 1 if true.*
- *Rule TF2: Logical or Boolean arguments, such as the argument to* `if` *or* `while`*, may take on any value, not just 0 or 1. 0 is treated as false and any other numeric value as true.*

5.1 Rule TF1

First consider the results of logical or Boolean expressions. (George Boole, who lived from 1815 to 1864, worked on differential equations, finite difference calculus, algebra, logic, and probability; see, for example Boole (1854) or Heath and Seneta (2001).) In Stata, these expressions make use of one or more of various relational and logical operators. The operators `==`, `~=`, `!=`, `>`, `>=`, `<`, and `<=` are used to test equality or inequality. The operators `& | ~` and `!` are used to indicate "and", "or" and "not". Note that it is a matter of taste whether you use `~` or `!` to indicate "not". In this column, in deference to Stata Corporation house style, we use `~`. If you want to learn more about any of these, see help on operators at [U] **16.2 Operators**.

For example, in the auto data, the expression `foreign == 1` will be true for those observations for which the variable `foreign` is 1 and false otherwise. As stressed earlier, the double equal sign `==` is used whenever you wish to test for equality: compare the use of the single equal sign `=` for assignment. As a second example, the expression `2 == 2` is always true. That may not seem helpful or instructive, but there are occasional uses for expressions that are necessarily always true. More complicated expressions can readily be constructed: `foreign == 1 & rep78 == 4` will be true whenever `foreign == 1` *and* `rep78 == 4`. Typing

```
. count if foreign == 1 & rep78 == 4
```

shows that there are 9 such cars in the auto data. (Incidentally, although the `count` command may seem trivial, it is a very simple and useful way of getting answers to some basic questions about your data.)

What is also important, and what may be new to you, is that logical expressions have numerical values. This can be immensely useful. In Stata, the rule is that false logical expressions have value 0 and true logical expressions have value 1. Thus, logical expressions may be used to generate indicator variables (also often called binary, dichotomous, dummy, or indeed logical or Boolean, depending on tribal jargon), which have values 0 or 1. The command

```
. generate byte himpg = mpg > 30
```

will `generate` a new variable that is 1 whenever `mpg` is greater than 30, and 0 otherwise.

By the way, `byte` indicating variable type is just another flourish here, as a reminder that the variable will have small integer values, so that there is a small advantage in storing them appropriately.

Some wrinkles should now be mentioned. What if `mpg` were missing? As mentioned earlier, the rule is that Stata treats numeric missing values as higher than any other numeric value, so missing would certainly qualify as greater than 30, and any observation with `mpg` missing would be assigned 1 for this new variable. This leads to the next wrinkle:

```
. generate byte himpg = mpg > 30 if mpg < .
```

would assign 1 if `mpg` were greater than 30, but not missing; 0 if `mpg` were not greater than 30; and missing if `mpg` were missing. The logic is that we did not say what result was wanted if `mpg` were missing: in the absence of instructions, Stata will shrug its shoulders in the only way it knows, and assign a result of missing. The same logic would apply if we were only interested in domestic cars:

```
. generate byte himpg = mpg > 30 if foreign == 0
```

If `foreign` were not equal to 0, then the result would be missing. Otherwise, the result would be 1 or 0, according to whether `mpg` was or was not greater than 30.

Examples of the usefulness of the numerical value of logical expressions arise whenever we want to count something. We've seen one example earlier in the column: let's now look at another in light of how Stata treats true and false. Suppose we want to create a new variable in which we will put the frequencies of `mpg` being greater than 30, by categories of `rep78`:

```
. bysort rep78: generate byte nhimpg = sum(mpg > 30)
. by rep78: replace nhimpg = nhimpg[_N]
```

In the first of these two commands, the function `sum()` produces a cumulative or running sum of `mpg > 30`. If `mpg > 30`, the Boolean expression `mpg > 30` is evaluated as 1, and 1 is added to the sum; otherwise, the expression is evaluated as 0, and 0 is added. This yields a running count of the number of observations for which `mpg > 30`. In the second command, we replace the running count by its last value, the total count. This is all done within the framework of `by:`, for which data must be sorted on `rep78`, which is done as part of `bysort:`. Under `by:`, the `generate` is carried out separately for each group of `rep78`. Similarly, the `replace` is done separately for each group of `rep78`.

In this example, we have been using the fact that there are no missing values of `mpg` in the auto data. And whenever you know that this is true of a variable in your data, you too can ignore the possibility of missing values. But a more general recipe for counting observations greater than some threshold is to use

`sum(` *varname* `>` *threshold* `&` *varname* `< .)`

That is a trustworthy recipe for whenever you want to exclude missing values. Naturally, if you know that missing means in practice "too high to be measured", then you might want to include missing values.

5.2 Rule TF2

Recall the rule: *Logical or Boolean arguments, such as the argument to* `if` *or* `while`*, may take on any value, not just* 0 *or* 1. 0 *is treated as false and any other numeric value as true.*

Now consider what happens if you type something like

```
. list mpg if foreign == 1
```

Stata `lists` `mpg` for those observations for which `foreign` is equal to 1, and does not `list` them if this is not so. From the above, we can see that a more long-winded way of explaining this is that Stata `lists` `mpg` whenever the logical expression `foreign == 1` is true, or evaluates to 1.

So this looks like the same idea in a different form. It is, but there are extra twists. Consider now

```
. list mpg if foreign
```

There are no relational or logical operators in sight, but Stata is broad-minded here. It will still try its best to find a way of deciding on true or false; in fact, it will accept any argument that evaluates to a number not 0 as indicating true, and any argument that evaluates to a number 0 as indicating false. If the mathematical or computer jargon "argument" is new to you, think of it here as indicating whatever is fed to `if`.

Given a numeric variable such as `foreign`, Stata looks at the values of that variable, and any value not 0 is treated as true and any value 0 as false. In other words,

```
. whatever if foreign
```

and

```
. whatever if foreign ~= 0
```

are exactly equivalent. That is always true for any numeric variable. In practice, also, there is a very nice shortcut if, and only if, you have an indicator variable that takes only the values 0 or 1. The two commands

```
. list mpg if foreign == 1
. list mpg if foreign
```

are equivalent in practice in the auto data. In the first, Stata evaluates the expression `foreign == 1`, and then executes the action indicated (to `list`) if and only if the expression is true, or evaluates numerically to 1. In the second, Stata looks at the values of the variable `foreign`, and then executes the action if, and only if, the value is a number not 0. In the auto data, `foreign` is not 0 when, and only when, it is equal to 1, so the two conditions are satisfied by exactly the same observations. Over time this will save you many keystrokes when you are working with indicator variables, and it will let you type Stata syntax close to the way you are thinking, say, `if female` or

even `if ~female`. Here `~` reverses the choice: `~` flips any value not 0 to 0, and any value 0 to 1. But remember that numeric missings count as not 0.

You can always check, either interactively or in a program, that a variable has only the values 0 and 1. A good way to check is with `assert`, using `|` to indicate "or":

```
. assert  varname == 0 |  varname == 1
```

If *varname* were equal to any other value, Stata would deny the assertion. `assert` is a very useful command for checking that data do satisfy those conditions you think should be satisfied. Don't be misled by its placement in the Programming volume of the manuals at [P] **assert**: the many examples in that manual entry give a flavor of how it can be useful interactively.

Note also that if you typed, perhaps by accident,

```
. list mpg if rep78
```

you would get a `list` for all observations, because `rep78` is never 0. It is the same logic.

If the argument were just a number, the same logic still applies. This also can be useful with the `if` and `while` programming commands, so that, for example, `while 1` would begin an infinite loop. We won't go into that further now, but you can learn more from [P] **if** and [P] **while**.

Finally, if you were to supply, most probably by accident, just the name of a string variable or a text string as an argument to `if` or `while`, there would be an error message, as Stata cannot interpret either as a numeric argument. This does not apply, however, to expressions in which strings are tested for equality or inequality or processed to yield a number. The expression `"frog" == "frog"` is true, and evaluated as 1, while `index("Stata", "S")` yields a number (in this case also 1).

6 Applying the ideas to some specific problems

Now let us apply these ideas to various problems.

6.1 Has a doubly entered dataset been entered properly?

One way of checking data entry is to type in the same dataset twice. If this has been done correctly, then every observation should have an identical twin. This is a little different from the more common problem in which duplicate observations may signal some kind of accidental repetition, but the way to detect it in Stata is much the same. We can check for such duplicate observations in a single line:

```
. bysort *: assert _N == 2
```

Here `*` is a wildcard for all variables. `_all` would also work here. In each distinct group, the number of observations `_N` should be 2. If this is so, then Stata says nothing, a case of no news being good news. If it is not so, we might find it best to generate an indicator variable:

```
. by *: gen isbad = _N ~= 2
```

and then to `edit if isbad` or `list if isbad`. The "bad" observations so identified will be unique (`_N` is 1) or occur in threes or larger groups (`_N` > 2).

6.2 Observations in a group differing on a variable

Suppose you have data on various individuals with genotypes ascertained from samples taken at different times. You want to `list` only those samples with differing genotypes for each individual.

Suppose the data look like this small subset:

```
. list id genotype in 1/7
1. 0       vv
2. 0       vv
3. 1       vv
4. 1       ww
5. 2       ww
6. 2       vv
7. 2       ww
```

Note that `genotype` could be either a string variable or a numeric variable with labels. The solution here applies to both, and also to numeric variables without labels. Consider the effects of sorting the data on `id` and then on `genotype`:

```
. sort id genotype
```

If all the values of `genotype` are the same for each `id`, then after sorting the first value within each `id` will equal the last. If there is any variation within `id`, that will not be true. This will work regardless of the number of observations for each `id`, the number of `genotype`s, and the type of variable used. Thus, in the example data for `id` 0, the first value `vv` will equal the last, but for `id` 1 and 2, the first and last values will differ. The example of `id` 2 also shows why sorting first is essential, as at present the first and third value are both `ww`, but the middle value is `vv`.

Accordingly, we work out which groups have different values and then `list` those groups only:

```
. bysort id (genotype): generate byte diff = genotype[1] ~= genotype[_N]
. list id genotype if diff
```

As always, with `by:` subscripts apply to observations within each group. `[1]` thus denotes the first observation and `[_N]` the last observation within each group. The expression is logical or Boolean and will be evaluated numerically as 1 or 0. If the corresponding values differ, then `diff` will be 1, and if they are the same, `diff` will be 0. Then the `list` is restricted to values which are different: `if diff` is a reliable shortcut for `if diff == 1` as all those values which are not 0 will in fact be 1.

How could this be extended to identifying groups that differ on at least one of two or more variables? One way would be first to use `egen, group()` to group values according to one or more variables, and then to use the same method on the resulting variable.

6.3 Do any or all group members possess some characteristic?

This next problem is a generalization of the previous one. The characteristic might be having exactly the same value within groups, but it could be something else.

In the simplest case, you might have a binary variable recording whether (say) persons are male or female, unemployed or employed, or whatever, and some group variable, say a variable recording a family identifier. For example,

```
. l family person female in 1/12
      family     person     female
1.         1          1          1
2.         1          2          1
3.         1          3          1
4.         2          1          0
5.         2          2          0
6.         2          3          0
7.         3          1          0
8.         3          2          0
9.         3          3          0
10.        3          4          1
11.        3          5          1
12.        3          6          1
```

As you can guess, `female` is recorded as 1 for female and 0 for male: it is a good convention to use names like `female`, signifying directly which sex is denoted by 1, rather than names like `gender` that require explanation of the coding used.

Consider various families in the listing just given:

1. Family 1 contains 3 females, and so values of `female` are 1, 1, 1.
2. Family 2 contains 3 males, and so values of `female` are 0, 0, 0.
3. Family 3 contains 3 males and 3 females, and so values of `female` are 0, 0, 0, 1, 1, 1.

From these examples, we can see a correspondence between two ways of thinking about such families:

1. If all members of a family are female, the minimum value of `female` is 1 in that family, and vice versa.
2. If no members of a family are female, the maximum value of `female` is 0 in that family, and vice versa.
3. If any member of a family is female, the maximum value of `female` is 1 in that family, and vice versa.

Thus, let us `sort` by `family female` and then within each `family` look at the minimum (first) and maximum (last) value of `female`.

```
. sort family female
. by family: generate byte anyfemale = female[_N]
. by family: generate byte allfemale = female[1]
```

In the simplest case, with no missing values, `anyfemale` or `allfemale` will be 1 or 0 according to whether it is true (1) or false (0) that any or all in a family are female. However, real examples could easily contain missing values. Missing numeric values (.), counted as higher than any other numeric values, will be sorted to the end of any group of numeric values. Thus, if any values of `female` were missing, so also would be `female[_N]`.

We will use indicator variables instead, defined such that the first or the last value in each group holds what is needed to answer the question.

The condition that *any* person be female can be put also in this way: at least one person is female.

```
. generate byte isfemale = female == 1
. bysort family (isfemale): generate byte anyfemale = isfemale[_N]
```

Note the subtle difference between `female` and `isfemale`. If `female` is 0 or 1, so also is `isfemale`. But if `female` is missing, `isfemale` is 0, because the expression `female == 1` is then false.

What is meant precisely by *all* persons being female? Does this mean "absolutely all persons are female", or it is enough that "all persons with known (not missing) sex are female"? You must decide this nuance on scientific or practical grounds, but let us show both solutions. Whether absolutely all persons are female is given by

```
. generate byte isfemale = female == 1
. bysort family (isfemale): generate byte allfemale = isfemale[1]
```

because if there are any examples of `isfemale` of 0—corresponding to anyone male or with missing value for sex—then `isfemale[1]` will be 0 after sorting. Whether all non-missing values are female is given by

```
. bysort family (female): generate byte allfemale = female[1]
```

which is what we had before. Note that if all values of `female` are missing, then `allfemale[1]` too is missing, which seems right—except that you should be sure not to type thereafter `if allfemale` when you really mean `if allfemale == 1`.

More generally, we can use any expression that is true or false to create an indicator variable. Sorting on that within groups (here families) ensures that minimum and maximum are first and last within each group, and then we can use those minimum and maximum values to determine group characteristics. Hence, if the characteristic of interest is not coded as a 0–1 variable, we just need to create such a variable:

```
. generate byte isDemo = party == "D"
. bysort family (isDemo): generate byte anyDemo = isDemo[_N]
. by family: generate byte allDemo = isDemo[1]
```

Once more, it turns out that we could save a command here and there by using `egen, max()` or `egen, min()` as appropriate, but the approach from first principles brings its own rewards.

7 Summary: The rules for by:

Let us now summarize the rules for `by` *varlist*`:`.

- *Rule BY0: Before running any command* `by` *varlist*`:`*, the observations in memory must first be* `sort`*ed according to varlist.* In Stata 7, `bysort` allows you to telescope `sort` *varlist* and the subsequent `by` *varlist*`:` command into a single command line: that is very often convenient, but it is not compulsory.
- *Rule BY1: Any command issued* `by` *varlist*`:` *is carried out separately for each group of observations with identical values on varlist. varlist may contain one or more variables, which may be numeric and/or string.*
- *Rule BY2: Under* `by` *varlist*`:`*, the Stata built-in variables* `_n` *and* `_N` *and subscripts are interpreted with respect to the current group of observations, not the whole dataset.*

8 What's next?

I hope that it is now clear that `by` *varlist*`:` is a very powerful construct. One major theme in this column—of working through lists—will be revisited in the next issue, when I shall be discussing how to set up your own loops over lists using commands like `foreach` and `forvalues`.

9 Acknowledgments

William Gould provided very helpful comments on an earlier version of the section on true and false. Some material stems directly or indirectly from questions asked on Statalist.

10 References

Boole, G. 1854. *An Investigation of the Laws of Thought.* London: Walton and Maberley.

Cox, N. J. 2001. Speaking Stata: how to repeat yourself without going mad. *Stata Journal* 1: 84–95.

Heath, P. and E. Seneta. 2001. George Boole. In *Statisticians of the Centuries*, eds. C. C. Heyde and E. Seneta, 167–170. New York: Springer-Verlag.

Lachenbruch, P. A. 1992. ip2: A keyboard shortcut. *Stata Technical Bulletin* 9: 9. In *Stata Technical Bulletin Reprints*, vol. 2, 46. College Station, TX: Stata Press.

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*.