The Stata Journal (2016)
16, Number 1, pp. 229–236

Speaking Stata: Truth, falsity, indication, and negation

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

Abstract. Many problems in Stata call for selection of observations according to true or false conditions, indicator variables flagging the same, groupwise calculations, or a prearranged sort order. The example of finding the first (earliest) and last nonmissing value in panel or longitudinal data is used to explain and explore these devices and how they may be used together. Negating an indicator variable has the special virtue that selected observations may be sorted easily to the top of the dataset.

Keywords: dm0087, true or false, logical, Boolean, indicator variable, dummy variable, sort, by, panel data, longitudinal data, programming, data management

1 Introduction: True and false, in Stata and otherwise

The title may hint at a miniature philosophical treatise, but the topic is eminently practical. We start with these fundamentals:

1. When fed an argument, Stata takes nonzero values as true and zero values as false. Thus `42` in `if 42` would count as true, and `0` in `if 0` would count as false, however unlikely it may be that anybody would write either in Stata. Stata would always try to do something given `if 42`, but it would never try that given `if 0`.

2. When producing results for true-or-false evaluations, Stata returns 1 for true and 0 for false. Thus the expression `x > 42` would return 1 if and only if `x` was indeed greater than 42, and 0 otherwise. (It is a side issue—but one that bites often enough to deserve a big flag—to emphasize in this example that any numeric missing value does count as greater than 42.) Similarly, the function call `missing(x)` will return 1 if and only if `x` is missing, and 0 otherwise. "Indicator variable" is a common term for variables that take on values 1 or 0. "Dummy variable" is another common term, especially for those who first met the idea in a regression course.

3. It follows that using 1 and 0 for true and false is at root just a convention, although a convention that appears supremely natural and helpful, especially with knowledge of those tricks possible with 1s and 0s. Thus adding lots of 0s and 1s is precisely how to count how many observations have the condition marked by 1s.

Such (0, 1) values are often labeled Booleans or logicals. However, you can, if you wish, use other conventions too. The easiest useful alternative is to use -1 for true and 0 for false. That last simple idea is the least standard detail in this column, so readers broadly familiar with the territory here may want to focus on that point alone.

George Boole (1815–1864) was a British mathematician who became a professor at Queen's College, Cork in Ireland (now University College Cork). He is best remembered for his works in logic and probability that lie behind the term "Boolean". His major book was Boole (1854). An earlier, shorter book (Boole 1847) is among those pieces collected in Boole (1952). Hailperin (1986) revisited this work from a more modern perspective. MacHale (1985, slightly revised 2014) gives a full-length biography that covers his personal life and his family, which remains distinguished to the present, as well as mathematical contributions. See also Iverson (1962) for a now classical discussion of logical values in computing; Knuth (2011) for an authoritative survey of zeroes and ones in programming; and Gregg (1998) for other material on Boolean algebra, circuit design, and the logic of sets.

The next section of this column uses a slightly tricky problem to illustrate the use of indicator variables and also their negations. The last section sketches some more general advice for programmers.

2 Illustration: First and last nonmissing values

Here is a concrete problem. A panel dataset defined by identifiers and a time variable is speckled with missing values. Your job is to find the first (earliest) and last nonmissing values for a particular variable. A sandbox for this problem could be the Grunfeld panel dataset, messed up randomly for the purpose. I set the seed for random numbers for reproducibility. Here I use Stata 14.1. If you are using a version before 14, your results will differ slightly, but the principles are unaffected.

```
. webuse grunfeld

. set seed 2803

. replace kstock = . if runiform() < 0.2
(35 real changes made, 35 to missing)
```

The Grunfeld dataset includes 200 observations, 20 companies each for 10 years, all nonmissing, so we expect about $200 \times 0.2 = 40$ values to be set to missing by such a call.

Now to the problem. If you know about `collapse` supporting identification of first and last nonmissing values, please set that aside. We should not need to destroy a dataset to find some of its contents. If you know that there are user-written `egen` functions to do this, or indeed of any other canned solution, please set that aside also. We seek a solution from first principles.

The problem would be easy if there were no missing values. We just need to sort the data into the right order and identify the first and last values using subscripts to identify the observations needed.

```
. bysort company (year): generate first = kstock[1]
. by company: generate last = kstock[_N]
```

The workhorse here is `by:`. `bysort company:` bundles two operations into one, to first `sort` on `company` and then to carry out calculations separately for each group of observations. As in elementary algebra, the operation on the inside, `sort`, is carried out first. In this case, each group of observations is for a single company defining a separate panel. If observations are already sorted by group (here by distinct values of `company`), the `sort` element is unnecessary but harmless. If observations are not already sorted, the `sort` element is essential.

Under the aegis of `by:`, we can sort on other variables as well as the group identifier. Here that also is essential. Sorting on `year` within each panel allows the first value to be identified using subscript `1` and the last to be identified using subscript `_N`, which under `by:` is the number of observations in each panel and hence also the subscript of the last observation in each panel. (Easy: if there are 10 observations in a panel, 10 is also the subscript of the last.) If this is unfamiliar, then you may also want to look for manual sections discussing `by:` or go to Cox (2002).

The parentheses `( )` around `year` are part of the syntax. The variables before the parentheses, just `company` in this example, define the groups for which separate calculations are needed. The variables inside the parentheses, just `year` here, define the order of observations within the group when that is important. If the order within the group is of no consequence, no variables need be included within parentheses. If we were calculating a group total or mean, the sort order would not matter.

We will come back to what is before and what is inside the parentheses, because it is a handle helping with all kinds of calculations in Stata that in other software would often call for nested loops.

That code is not yet the solution to our problem, because we do have missing values. The first or last values in different panels could be missing. We must not just hope that is not so. But this is a good start, and the rest of the solution is to add a twist to keep missing values out of the way.

The best device is to create an indicator variable for missing values:

```
. generate ismissing = missing(kstock)
. bysort company (ismissing year): generate first = kstock[1]
```

Let's go through that more slowly. The new variable `ismissing` is 1 if `kstock` is missing and 0 otherwise. Within each panel, we `sort` first on `ismissing`, so all the observations with missing values on `kstock` with value 1 on `ismissing` get sorted after the others with value 0 on `ismissing`. Then, within such blocks, we sort on `year`. So the first value of `stock` within each panel should be the first nonmissing value, so long as there is at least one nonmissing value. At worst, all the values for a panel on `kstock` will be missing, and then calculation will return missing as the first nonmissing value, which seems fair enough.

As before, the order of variables is crucial: first `company`, then `ismissing`, then `year`. A different order would usually produce a different sort order for the dataset and a different answer, probably wrong.

As explained, the parentheses `()` control exactly how observations are sorted.

- A paraphrase of the last Stata command above is this: within blocks of observations defined by different values of `company`—sorted internally first by `ismissing` and then by `year`—find the first value of `kstock`.
- That is different from this: within blocks of observations defined by different values of `company` and `ismissing`—sorted internally by `year`—find the first value of `kstock`.
- Both are different from this: within blocks of observations defined by different values of `company`, `ismissing`, and by `year`, find the first value of `kstock`.

Which syntax you want is crucially dependent on the problem, but the parentheses are there to make the distinction you need. Getting it right is the tricky part of using `by:`. Getting it wrong a few times and thinking it through with small-sample datasets where you can see results easily and quickly is the way to learn how to get it right.

So far, so good, but what about the last nonmissing values? Because we sorted missings to the end of each panel to keep them out of the way, the last value for each panel will certainly be missing on `kstock` even if there is only one missing value in each panel. The solution is to flip the sort order around. This is where negation can be used to solve the problem.

```
. replace ismissing = -ismissing
```

Negation (note the minus sign `-` in the command just given) flips the 1s all to -1 and leaves the 0s untouched. Now, we can re-sort with the changed variable.

```
. bysort company (ismissing year): generate last = kstock[_N]
```

The observations with missing values for `kstock` (with `ismissing` -1) now always come before those with nonmissing values (with `ismissing` unchanged at 0). Within those two subsets, we sort on `year`. Thus the last nonmissing value should come last, and we can pick it up using the subscript `[_N]`.

As before, if all the data for a panel are missing, then the result would also be missing, and again that is fair enough.

By the way, why is the code not like this?

```
. generate ismissing = missing(kstock)
. bysort ismissing company (year): generate first = kstock[1]
. by ismissing company (year): generate last = kstock[_N]
```

The problem with that code is that missings on `stock` all map to missings on the new variables. That is often awkward. We could clean up afterward, but for most purposes, such code creates as many problems as it solves.

If you are interested in how we could clean up, here is one way.

```
. bysort company (first): replace first = first[1]
. bysort company (last): replace last = last[1]
```

This is similar logic. Within panels, we sort on the variable of interest, either `first` or `last`. Nonmissing values of interest will then be in the first observation of each panel and can be copied to all observations in the panel. We do not need conditions on `replace` such as `if missing(first)` or `if missing(last)` because there is no loss in overwriting nonmissing values with the same nonmissing values.

Naturally, you could argue that the observations with missings are just useless and might as well be dropped. That is true if all variables of interest are missing in those observations; otherwise, it would result in throwing away data that might be useful. (A recent column of mine, Cox (2015), introduced a program, `missings`, helpful for such problems.)

To recap the problem, let's gather all the commands for the recommended solution:

```
. webuse grunfeld, clear
. set seed 2803
. replace kstock = . if runiform() < 0.2
. gen ismissing = missing(kstock)
. bysort company (ismissing year): gen first = kstock[1]
. replace ismissing = -ismissing
. bysort company (ismissing year): gen last = kstock[_N]
```

We will not list the results here, partly because of space but mostly because you can try this out for yourself. But we should check on the results:

```
. count if missing(first, last)
. bysort company (first): assert first[1] == first[_N]
. bysort company (last): assert last[1] == last[_N]
```

The first check is whether any result is missing for the new variables. In principle, there could be a missing result if any panel was all missing. Because there are none, we need not take that further. If you have random numbers different from mine, then your results could be different.

The second and third checks are whether results are constant within panels. If we sort on `first` (similarly on `last`) within panels, then any different values would be shaken apart. Here the result of `assert` would be an error message if the assertion were not true: literally, no news is good news. Know that there was no such message. For more on `assert`, see any or all of its help, its manual entry, and Gould (2003).

3 Invitation: Some wider uses

Let's close with some general advice, especially for programmers. A common early device in programs is writing

```
. marksample touse
```

which creates an indicator variable with value 1 for observations to be used and with value 0 otherwise. That meaning explains the conventional name, `touse`, meaning (if you did not spot it) "to use". In practice, that variable is temporary, referred to thereafter as `‘touse’`, an incidental detail here.

Such a variable permits all kinds of useful stuff, often starting with

```
. count if `touse´
```

to determine if there are not any observations to apply a command to, in which case your program should probably bail out now, even if a zero count is good news for some purpose. (Perhaps the purpose of the command is to look for something unwanted, so a zero count is indeed good news.)

Sometimes, it is simpler to throw out observations you do not want to work with, provided that the dataset has been `save`d or `preserve`d or is not of long-term utility:

```
. keep if `touse´
```

Often the main point is just to control which part of the dataset is used, say, for a graph or some statistical analysis, so the qualifier `if ‘touse’` could be common in a program.

None of the uses mentioned so far in this section is affected by negating `‘touse’` so that its values are -1 and 0. We know -1 is not 0, hence true, and 0 manifestly remains 0 and false. But why would you do that? The main reason is whenever sorting the observations you want to the top of the dataset makes anything easier.

For example, I find that `list` is frequently a good way to output results. Its excellent `subvarname` option provides an easy way to label columns. Its separation options `separator()` and `sepby()` are often useful. It is smart on your behalf about spacing and boxing. And there are other advantages besides: see, for example, Harrison (2006).

Once I have counted how many rows will be in the table, the output is then often arranged by a single command of the form `list` ... `in 1/`*whatever*. The observation numbers will often be relevant to the displayed results; if not, they can always be suppressed.

That is a small trick, but one that I have found helpful in programming. Indeed, working interactively as well, it can be useful whenever the observations of most interest come first in the dataset.

4 Conclusion

What makes a language practical to learn and to use? Often it is that a small number of key concepts allow a large number of problems to be solved directly and efficiently.

In this column, we have looked at a bundle of key Stata concepts that very much belong together: indicator variables, `by:` for groupwise calculations, and deliberate and delicate control of sort order to enable exactly what you want. A particular twist is that negating an indicator can be useful too: logical values of -1 remain true and immediately allow a sort order that can be as or more convenient than the standard order in which true values follow false.

5 References

Boole, G. 1847. *The Mathematical Analysis of Logic, Being an Essay Towards a Calculus of Deductive Reasoning*. Cambridge: Macmillan, Barclay, and Macmillan.

———. 1854. *An Investigation of the Laws of Thought, on Which are Founded the Mathematical Theories of Logic and Probabilities*. London: Walton and Maberley.

———. 1952. *Studies in Logic and Probability*. London: Watts.

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

———. 2015. Speaking Stata: A set of utilities for managing missing values. *Stata Journal* 15: 1174–1185.

Gould, W. 2003. Stata tip 3: How to be assertive. *Stata Journal* 3: 448.

Gregg, J. R. 1998. *Ones and Zeros: Understanding Boolean Algebra, Digital Circuits, and the Logic of Sets*. Piscataway, NJ: IEEE Press.

Hailperin, T. 1986. *Boole's Logic and Probability. A Critical Exposition from the Standpoint of Contemporary Algebra, Logic and Probability Theory*. Amsterdam: North-Holland.

Harrison, D. A. 2006. Stata tip 34: Tabulation by listing. *Stata Journal* 6: 425–427.

Iverson, K. E. 1962. *A Programming Language*. New York: Wiley.

Knuth, D. E. 2011. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Upper Saddle River, NJ: Addison–Wesley.

MacHale, D. 1985. *George Boole: His Life and Work*. Dublin: Boole Press.

———. 2014. *The Life and Work of George Boole: A Prelude to the Digital Age.* Cork, Ireland: Cork University Press.

About the author

Nicholas Cox is a statistically minded geographer at Durham University. He contributes talks, postings, FAQs, and programs to the Stata user community. He has also coauthored 15 commands in official Stata. He was an author of several inserts in the *Stata Technical Bulletin* and is an editor of the *Stata Journal*. His "Speaking Stata" articles on graphics from 2004 to 2013 have been collected as *Speaking Stata Graphics* (College Station, TX: Stata Press, 2014).