The Stata Journal (2020)
20, Number 1, pp. 236–243 DOI: 10.1177/1536867X20909698

Speaking Stata: Concatenating values over observations

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

Abstract. Concatenation, or joining together, of strings or other values, possibly with extra punctuation such as spaces, is supported in Stata by addition of strings and by the `egen` function `concat()`, which concatenates values of variables within observations. In this column, I discuss basic techniques for concatenating values of variables over observations, emphasizing simple loops that can be tuned to suit variants as desired. Commonly, such concatenated strings report a profile or history of each individual within panel or longitudinal data. Such histories can then be analyzed further.

Keywords: pr0071, concatenation, strings, panel data, longitudinal data

1 The problem stated

In this column, I address a bundle of related questions that arises most often in a threefold context. First, people or other individuals (firms, places, etc.) each have an identifier. Second, each of those individuals is observed over time. Third, and in particular, categories are observed over time. So, imagine a toy dataset like this:

```
. clear
. input id time str1 whatever
1  1  "A"
1  2  "B"
1  3  "C"
2  1  "B"
2  2  "B"
2  3  "C"
end
```

The jargon of panel or longitudinal data will be familiar to many readers, and this is one such dataset.

An appealing step is to concatenate each history of category values. We can imagine joining the strings together, thinking that identifier 1 has a profile or history `"ABC"` and identifier 2 `"BBC"`. (More will be said shortly on verbs "concatenate" and "catenate" and associated nouns.)

So, how do we do that easily in Stata? And what related problems arise, and how are they tackled? All that is the focus of this column.

Before we get down to business, here are some broad-brush comments about what is and what is not included. Identifiers can be string or numeric, unlike what is compulsory with `tsset` or `xtset`. But as with either of those commands, there is no insistence that the numbers of times observed or the times themselves are identical across panels or that the times are regularly or equally spaced, just that there is an unambiguous time order within each panel. `whatever` being a string variable is incidental here, except that it is a good place to start thinking about the problem. We could equally have a numeric variable with value labels or a numeric variable that codes some categorical condition or even small counts.

Further, it should be clear that the example is deliberately short and simple. Serious examples could be much longer. Yet further, the entire string or history may not be of most interest, but for example, whether and which individuals include spells such as `"BBB"`, or whatever it is.

What I am not imagining is that it is especially interesting or useful to concatenate, say, `"12.3 23.4 34.5"`, producing strings of measured values. If that is merely my failure of imagination to foresee your interest, much of the following may still be of some help to you.

There are several similarities of spirit and some of substance between the ideas in this column and the much more elaborate project on sequence analysis reported by Kohler and Brzinsky-Fay (2005) and Brzinsky-Fay, Kohler, and Luniak (2006). It seems simpler to explain the few basic (and unoriginal) ideas here directly and independently. Furthermore, there are overlaps with work on spells or runs (Cox 2007, 2015).

2 What's in a name?

The Latin root here underlying "concatenation" is *catena*, meaning a chain, which is an occasional word in English too. For example, a catena in soil science is a series of soils observed down a hillslope, varying in accordance with variations in steepness, drainage, and so forth (as I learned in secondary school geography more than 50 years ago).

In computing, it is quite common to talk of concatenation of strings or, more generally, arrays, which can be seen to mean chaining (together). Here "together", and similarly the con- syllable, is at best emphatic and at worst redundant. Hence, "catenation" would have exactly the same meaning, but it appears much less popular, with notable exceptions in accounts of APL and related programming languages (Iverson 1962, 1977; Brown, Pakin, and Polivka 1988; Thomson and Polivka 1995).

Similarly, Unix users are likely to be aware of the command `cat`, which prints (meaning, shows) the files named by its arguments. Its name was explained by Kernighan and Pike (1984, 15): "'Catenate' is a slightly obscure synonym for 'concatenate.'" In his intriguing memoir, Kernighan (2020) expands on the circumstances of such names, ranging from developers' personal preferences for conciseness to the physical difficulty of typing on available keyboards.

3 Concatenation across variables

Concatenation across variables has been supported for a long time in Stata. If `frog`, `toad`, and `newt` are string variables, then the expression

```
frog + toad + newt
```

concatenates, meaning that the operator `+` with string operands yields addition of strings lengthwise. Nothing has been said about the contents of those variables, which may include spaces or other punctuation as particular string characters. Either way, other characters can be included in an expression, say,

```
frog + " " + toad + " " + newt
```

or something similar with commas or other punctuation.

The `egen` function `concat()` was published via Cox (1999) and folded into Stata 7 in 2000. It was itself a rewriting of an older 1998 command called `catenate`, still on Statistical Software Components. It allows all the above and more, including conversion on the fly of numeric variables to string and use of value labels where they exist and are wanted.

Mentioning that might create an expectation that a new `egen` function or functions will feature prominently in this column. Taste and experience imply otherwise. Showing how to `generate` what you want from first principles seems a better way to explain what to do, not least because you will then be better placed to do something a bit different if that is what you want.

4 Concatenation across observations

Let me underline that concatenation of string values across observations has long been possible in Stata. The point of this column is just to discuss how best to do it. Thus, in the toy dataset of section 1,

```
whatever[1] + whatever[2] + whatever[3]
```

yields `"ABC"` for identifier 1, while

```
whatever[4] + whatever[5] + whatever[6]
```

yields `"BBC"` for identifier 2. Further, by comparison with code recently given, you can imagine how to insert spaces or other characters as needed. A much better way to get both results at once is

```
. bysort id (time): generate history = whatever[1] + whatever[2] + whatever[3]
```

If you did not know that, then you have now heard about what is likely to be the most valuable tip in all of this column for your future use of Stata. Taking that more slowly (see also Cox [2002]), we note that

```
bysort id (time):
```

ensures sorting of the dataset first by identifier and then by the time variable and specifies that what is to follow is to be carried out separately within the groups of observations defined by the distinct values of `id`, otherwise known as our panels. Then, within that framework,

```
generate history = whatever[1] + whatever[2] + whatever[3]
```

adds the first three values (in our toy dataset, that is all of them) of the variable `whatever` in each panel. Crucially, within the framework defined by `by:`, subscripts such as `[1]`, `[2]`, and `[3]` start afresh within each panel.

That is good, but going further in the same direction does not appeal. In the expression needed for the history, adding 3 terms is bearable, but adding 30 or 300 terms, or many more, will not be. Moreover, the recipe is not general if the number of observations in each panel is not the same. It so happens that a reference to a nonexistent value, say, `whatever[4]`, results in an addition of an empty string, so nothing at all, but such code is at best awkward and generally should be avoided.

There is a more general recipe without too much associated pain. The procedure includes three steps:

- Initialize a new variable:

```
. bysort id (time): generate history = whatever[1]
```

- Add new values in a cascade so that each value is the previous value plus the value of the current value:

```
. by id: replace history = history[_n-1] + whatever if _n > 1
```

- Consistently copy the total string from the last observation in each panel to all observations in the same panel:

```
. by id: replace history = history[_N]
```

Key here is, once again, that under `by:`, not just specific subscripts such as `[1]` but also more general subscripts such as `[_n-1]` and `_N` are defined within the groups of observations defined by the distinct values of the variable or variables fed to `by:`, here just `id`. In short, the machinery ensures separate calculations for each panel.

The cascade just mentioned works like this. The qualifier `in` is not allowed with `by:`, but this is no problem, because the scope to use `if` with subscripts is even more useful. So, after initializing `history` to be the value of `whatever[1]`, the cascade starts with the second observation in each panel, so long as it exists. For that second observation, the computation becomes

```
. by id: replace history = history[1] + whatever if _n == 2
```

which means, as we could type if we so wished,

```
. by id: replace history = history[1] + whatever[2] if _n == 2
```

Then, in turn for the third observation, again spelling out all the details,

```
. by id: replace history = history[2] + whatever[3] if _n == 3
```

And so on: An excellent fact is that this looping is automated for us, and Stata takes care of any awkward details such as differing numbers of observations in the various panels.

The process ends with tidying up so that all observations in a panel contain the same values for `history`. That is what the statement below the last bullet point does. Contrariwise, if it were important that the history just be "the history so far", then that last statement can be skipped.

A helpful detail about this approach is that punctuation between strings need be added only once. So, if we had reason to add spaces between strings, the only tweak needed is to

```
. by id: replace history = history[_n-1] + " " + whatever if _n > 1
```

and evidently commas or other punctuation can be added just as easily.

Now, let us see what needs to be varied if the variable of interest were not string but numeric. In practice, this would usually mean integers, often small integers. The variation needed is just conversion to string on the fly. Let's give the code in one, but after mapping `whatever` to numeric to give an example you could try:

```
. label define whatever 1 A 2 B 3 C
. encode whatever, gen(something) label(whatever)
```

When the numeric values are just single-digit integers 0 to 9, then no ambiguity can arise, and people might often prefer a compact concatenation:

```
. bysort id (time): generate history2 = strofreal(something[1])
. by id: replace history2 = history2[_n-1] + strofreal(something) if _n > 1
. by id: replace history2 = history2[_N]
```

`string()` is a synonym for `strofreal()` and indeed likely to be recalled as a matter of course by many long-term Stata users.

Let us spell out that spaces or other punctuation should typically be used whenever ambiguities might arise in interpreting a string correctly, as when 1, 0, and 10 are all possible values. It may also be a good idea for readability, as when a person's moves from a place are coded by `"WY OH MI"` or `"GB FR DE"`.

Let's now consider some variations on the theme so far. The underlying message is that, given the main ideas, these are easy to solve with simple modifications of the code.

- *Time order, but omit repeats of the previous.* Here the idea is that `"AAABBC"` contracts to `"ABC"`, while `"AAABBCAA"` contracts to `"ABCA"`. That would be (here for string `whatever`)

```
. bysort id (time): generate history3 = whatever[1]
. by id: replace history3 = history3[_n-1] + whatever if _n > 1 &
> whatever != whatever[_n-1]
. by id: replace history3 = history3[_N]
```

- *Alphanumeric order.* So `"ABCA"`, `"AABC"`, and `"CBAA"` (and various others) are all equivalent.

```
. bysort id (whatever): generate history4 = whatever[1]
. by id: replace history4 = history4[_n-1] + whatever if _n > 1
. by id: replace history4 = history4[_N]
```

- *Alphanumeric order, but omit repeats.* This case should now be evident using ideas from the last two.

5 Tagging each panel just once

Panel datasets imply two scales: the small scale of individual observations and the larger scale of the panels themselves. In the code discussed here, each panel's history is repeated in all observations of that panel (unless, as mentioned briefly, code shows the history so far). Often, we wish, say, to count, tabulate, or graph results for panels, not their constituent observations. The standard trick here is to use the `tag()` function of `egen` so that

```
. egen tag = tag(id)
```

creates an indicator variable that is 1 for one observation in each panel and 0 for the others so that doing anything `if tag` selects each panel just once. `if tag` is a convenient and safe abbreviation of `if tag == 1` because the created variable is only ever 1 or 0 and true (nonzero) values arise only as values of 1. The `tag()` function, too, was published in Cox (1999) and folded into Stata 7 in 2000. My recollection is that such ideas were part of Stata folklore even then and often discussed on Statalist.

Because groups could generally be as small as single observations, there are only two systematic ways to tag just one observation: to tag the first observation or to tag the last; in groups of one, that is the same observation. For your information, `tag()` tags the first observation in each group, but it is, or should be, immaterial which is tagged because `tag()` is intended for situations in which values are identical within each panel so that working with just one is sufficient as well as desired.

6 Using this idea

That's the main idea, and speaking proverbially, all is easier than rocket science or brain surgery. Uses of the idea are manifold and include searches for particular patterns, say, using the string function `strpos()` or regular expression machinery.

One method for counting substrings within strings was discussed in Cox (2011). The idea is to compare the lengths of strings with those of substrings removed. Thus, the difference

```
length("ABABAB") - length(subinstr("ABABAB", "A", "", .))
```

is necessarily the number of occurrences of `A` within the string.

7 Conclusion

Standard advice within the Stata community is that panel or longitudinal data are almost always better analyzed with a long layout (structure or format, if you insist) in which each individual panel is represented by one or more observations. This column introduces a simple method that subverts that principle in a sometimes useful way by concatenating strings or other values in a series of observations into single strings. The machinery needed is just simple looping set up within the framework of `by:`.

8 References

Brown, J. A., S. Pakin, and R. P. Polivka. 1988. *APL2 at a Glance*. Englewood Cliffs, NJ: Prentice Hall.

Brzinsky-Fay, C., U. Kohler, and M. Luniak. 2006. Sequence analysis with Stata. *Stata Journal* 6: 435–460. https://doi.org/10.1177/1536867X0600600401.

Cox, N. J. 1999. dm70: Extensions to generate, extended. *Stata Technical Bulletin* 50: 9–17. Reprinted in *Stata Technical Bulletin Reprints*. Vol. 9, pp. 34–45. College Station, TX: Stata Press.

———. 2002. Speaking Stata: How to move step by: step. *Stata Journal* 2: 86–102. https://doi.org/10.1177/1536867X0200200106.

———. 2007. Speaking Stata: Identifying spells. *Stata Journal* 7: 249–265. https://doi.org/10.1177/1536867X0700700209.

———. 2011. Stata tip 98: Counting substrings within strings. *Stata Journal* 11: 318–320. https://doi.org/10.1177/1536867X1101100212.

———. 2015. Stata tip 123: Spell boundaries. *Stata Journal* 15: 319–323. https://doi.org/10.1177/1536867X1501500121.

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

———. 1977. *Algebra: An Algorithmic Treatment*. Pleasantville, NY: APL Press.

Kernighan, B. W. 2020. *UNIX: A History and a Memoir*. Seattle, WA: Kindle Direct Publishing.

Kernighan, B. W., and R. Pike. 1984. *The UNIX Programming Environment*. Englewood Cliffs, NJ: Prentice Hall.

Kohler, U., and C. Brzinsky-Fay. 2005. Stata tip 25: Sequence index plots. *Stata Journal* 5: 601–602. https://doi.org/10.1177/1536867X0500500410.

Thomson, N. D., and R. Polivka. 1995. *APL2 in Depth*. New York: Springer.

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 16 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* (2014, College Station, TX: Stata Press).