The Stata Journal (2020)
20, Number 3, pp. 746–756 DOI: 10.1177/1536867X20953578

Speaking Stata: Is a variable constant?

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

Abstract. Whether a variable is in fact constant—so that it takes on exactly the same value in different observations—is of common concern in statistical work. The question may arise for all the observations in a dataset or for different subsets of the dataset. It may arise because constancy is desirable or because constancy is undesirable, but either way we often need to check simply and quickly. In this column, I discuss several methods for checking. I do not claim to offer a complete analysis of why you might or should care but will give examples arising from experience.

Keywords: dm0103, data management, panel data, longitudinal data, missing values, summarize, assert, tabulate, indicator variables

1 Introduction

Stata's main data model consists of observations (rows, cases, or records in other terminology) and variables (columns, features, or fields). Despite their name, reflecting lengthy statistical theory and practice, variables in Stata sometimes are, or at least should be, constant in value, either in an entire dataset or in subsets of observations. In this column, I discuss how to check for constancy, or its converse, variability.

The problem of checking for constancy overlaps with that of finding duplicate observations, as tackled by the `duplicates` command, and with that of identifying distinct observations. On the latter, see Cox and Longton (2008) for a discussion, which ranges widely and also introduces the `distinct` command. If that command interests you, then type

```
. search distinct, sj
```

to find the latest update at the time of reading. Note that some people say "unique" rather than "distinct".

2 Constant across the entire dataset?

Let us start with wanting to check whether a variable is constant across the entire dataset.

The larger context often runs that variables that are constant are not useful. Sometimes, they are just uninformative, period, as when data imported from a spreadsheet include variables that are entirely missing, because the corresponding columns were empty. Sometimes, they have meaning, but, given that they are constant, they cannot be used in modeling, either as outcome or response variables, or as predictor or explanatory variables. If you want to use gender as a predictor, but all your data are for one gender, that will not work. So, for some purposes, such variables should certainly be identified and perhaps even `drop`ped. Typically, modeling commands will catch such variables, if only by producing results that are missing or otherwise signal a major problem. Spotting them early in a project can save time and frustration.

That isn't always true. In some projects, there are multiple data files, and it can be essential to have an identifier defining each file before they are combined (which here usually means `append`ed). I have often seen this when students, collaborators, or even external data providers tidily separate different subsets into worksheets or a series of files in some other format. Usually, there comes a point, even at the outset of a project, when those files need to be combined. One Internet source I will leave nameless issues 12 separate files for certain monthly time series: one for all the Januaries in a series of years, another for all the Februaries, and so on. It is hard to imagine a project in the field in question for which that is a good way to store data.

To make this all concrete, I provide here a sample sandbox dataset with two numeric variables and two string variables.

```
. clear

. input n1 n2 str4(s1 s2)
            n1         n2         s1         s2
  1. 42 999 "frog" "frog"
  2. 42 42 "frog" "toad"
  3. end
```

Numeric variable `n1` and string variable `s1` are clearly constant, while numeric variable `n2` and string variable `n2` are not. That is evident from what was just entered, and in a (very) small dataset, it would be evident from, say, `list` or `edit`. Clearly, the need is for methods that will also work with (much) larger datasets, whether "large" means many observations or many variables or both.

2.1 Numeric variables first: The role of summarize and assert

First, consider the case of numeric variables.

After any introduction to Stata, users are likely to suggest this idea for any numeric variable: `summarize` the variable, and then check the minimum and maximum. If they are the same, the variable is constant; otherwise, it is not. This can be a good idea and may link very well with the rest of what you are doing, but it is not as general as possible.

Let's run through the details. Assume that numeric variable `n1` in our sandbox dataset is being checked. Then, we might write

```
. summarize n1, meanonly
. display r(max) - r(min)
```

Evidently, the difference between the maximum and minimum, namely, the range, should be zero if the variable is constant.

Aside: r-class results, such as `r(max)` and `r(min)` here, are one kind of stored results, typically from a command that does some calculations but falls short of fitting a model to data and estimating its parameters. Even if you have not met them before, you can now recognize them by their names, always within `r()`. For an introduction to stored results in general, start with `help stored results`.

The `meanonly` option of `summarize` is misnamed in this sense: It produces more than the mean, which is precisely why we can access the minimum and maximum after it is run (Cox 2007). It is faster than plain `summarize` because it does less; and much faster than `summarize, detail` because it does much less. `summarize, meanonly` produces no output in the Results window, which is why we have to ask to see the returned results that it leaves in its wake. `display` is one way to do that. `return list` is another.

Calculating the range is an easy task. So is spotting by eye after `return list` that the maximum and minimum are the same. Another method is to use `assert` after `summarize, meanonly`.

```
. assert r(max) == r(min)
```

`assert` states a hypothesis, which may be true or false. This is pure logic, not significance testing. Here the hypothesis is that two returned results are equal. In other circumstances, `assert` may be used to check a hypothesis about data.

With `assert`, silence implies assent—to a hypothesis that is found to be true. Proverbially, no news is good news. A hypothesis that is false provokes an error message. With a hypothesis about the dataset, that hypothesis being false in just one observation of the dataset is enough to yield an error.

Note the double equals sign, `==`, used for testing equality in the last line of code, in contrast with the single equals sign, `=`, which usually implies assignment of one or more inputs to become an output or result. See the help for operators if you wish to see a full list:

```
. help operators
```

You can test the converse of any hypothesis too. Usually, the way to think is: What should be true, so that we need to know about exceptions?

If the concern is to identify variables that are genuinely variable, then a convenient formulation uses `!=` (not equals) as the operator.

```
. assert r(max) != r(min)
```

Here the idea is that the maximum should differ from the minimum, so it is departure from the state of inequality that will produce a flag.

2.2 What about missing values? And string variables?

We need now to face up to some complications that can bite. First, `summarize` ignores missing values.

Here the analysis branches naturally. If you do not care about missing values, then you will not care that `summarize, meanonly` is strictly telling you that nonmissing numeric values are identical (or different, as the case may be) without reporting anything directly about any missing values. What it conveys indirectly is that the number of nonmissing values is reported. That will be less than the number of observations in the dataset by the count of the missing values.

Many graphical, data management, and statistical commands automatically ignore missing values in any case, so in that sense they need not be an immediate problem.

Alternatively, if you do care about missing values, then explicit checks may be a good idea.

Second, a further complication that can bite is the existence of string variables. Many datasets include at least one string variable, commonly for holding information about identifiers. Some datasets hold many more string variables.

Ideally, data management methods work uniformly, or at least similarly, for numeric and string variables alike. Will the `summarize` and `assert` method above work for string variables too? Yes and no. Yes, in the sense that `summarize` just skips over string variables, ignoring them. That is good if that is what you want, and not otherwise.

A further fact, whether a limitation or an irrelevance: `summarize` will never tell you about missing string values in Stata's sense, namely, empty strings, `""`. This is a good point to spell out that

1. strings that are one or more spaces are not equal to empty strings.
2. leading or trailing spaces are used in comparisons of equality or inequality, so `"Alabama"`, `" Alabama"`, and `"Alabama "` are not equal, for all that people regard them as having the same meaning.
3. the character yielded by `uchar(160)` or `char(160)` is not a space either. This is an occasional problem that can baffle until it is realized.

The remedy for problems 1 and 2 is to clean up string variables using the `trim()` function. The remedy for problem 3 is usually to remove the character in question or sometimes to replace it with a space.

That said, what `summarize` does precisely given string variables needs careful attention. This sequence needs some thinking through. `s2` is a string variable in the current sandbox, which is not constant.

```
. summarize s2, meanonly
. return list
scalars: r(N) =  0
         r(sum_w) =  0
         r(sum) =  0
. assert r(min) == r(max)
```

Although `s2` is a string variable, it is not completely ignored by `summarize, meanonly` because that command leaves some r-class results in its wake. Then our test of whether the minimum and the maximum are equal produces silent assent. Why is that, especially because the variable is not constant?

The logic is that it is legal to refer to r-class results that do not exist: Stata just declares that those results are missing. As neither `r(max)` nor `r(min)` is defined by `summarize` here, both are reported as missing, and therefore they are agreed to be equal to each other. To show the principle, let us think of displaying a result that `summarize` never defines, say,

```
. display r(silly)
.
```

Note the result, the display of a single period representing system missing, Stata's default numeric missing value. Stata's attitude is that it does not know any value for `r(silly)` at the moment. Perhaps there is a command somewhere that leaves behind an r-class result `r(silly)`. Regardless of that, Stata can see no such result at the moment, and so `display` shows a missing value.

The bridge between what exists (ontology) and what is known (epistemology) is long, fragile, and dangerous. Here Stata is playing epistemologist, or more plainly saying: I don't know.

Stata jumps the other way if you name a variable that does not exist (which usually means that you get a variable name wrong, but either description can be correct). Stata is always confident about what variables are in memory in the current dataset, so the result is not an empty result but an error message.

So we need some caution about using `summarize` with string variables: it ignores them, which may be what you want, but there is scope to be misled if you don't follow exactly what it does.

There is more juice to be squeezed out of the ideas so far. Consider this code for some puzzling variable:

```
. assert puzzle == puzzle[1]
```

The assertion is that all values of `puzzle` are equal to the value of `puzzle` in the first observation. This is just one way of checking that values are equal to each other, as if all values are equal to the value in the first observation, then they are surely all equal (to each other). There is no particular magic in choosing the first observation as reference. You could, always, choose the last observation `puzzle[_N]` because that would work in a dataset of any size. You could choose observation 42, so long as you are sure that there are at least 42 observations in the dataset. By the way, this kind of code is obvious once understood, but all too likely to be puzzling to the Stata learner, or to yourself if you have forgotten the rationale some time after last using the code. Kind programming practice here might well include a comment with your code explaining what otherwise might appear cryptic.

A merit of that last line of code is that it applies to string variables as well as numeric variables. But once again the question arises: What about missing values? The code will certainly catch any variable that is missing in the same way for all observations. Missing in the same way only ever means empty strings for string variables. But missing in the same way means all values being either system missing, `.`, or just one of the extended missing values `.a` to `.z` for numeric variables. (If most or all of this is new to you, start at `help missing`.)

To keep matters concrete, let us add another observation to the sandbox with missing values.

```
. set obs 3
number of observations (_N) was 2, now 3
. replace n1 = . in 3
(0 real changes made)
. replace n2 = . in 3
(0 real changes made)
. replace s1 = "" in 3
(0 real changes made)
. replace s2 = "" in 3
(0 real changes made)
```

In our particular example, increasing the number of observations implies directly that values in the new observations are born as missing, but the code above spells out what the missing values are (what they are already, which is why there are reports of 0 real changes made).

Here is a way forward when that is not quite what you want. Focus first on the case of `n1` as a numeric variable. If we

```
. sort n1
```

then any missing values are sorted to the end of the dataset. That applies both to system missing values, represented by a period, `.`, and to extended missing values, any or all of `.a` to `.z`. The rationale is that among numeric values, missing values are always regarded as larger than all possible nonmissing values. It follows that (using `|` for logical or)

```
. assert n1 == n1[1] | missing(n1)
```

allows two possibilities, being equal to the first value or being missing. The function `missing()` returns 1 (true) if its argument is missing and 0 (false) otherwise, and that includes all kinds of missing values. The logic here is not broken either if there are no missing values or if all values are missing, because the "or" operator is inclusive.

If there are, contrary to expectation, rogue values such as 999 lurking in the data, the hypothesis will be false. The test will fail for `n2`, for example, even after sorting.

By the way, if you wanted to write code precisely tailored to the expectation, such as

```
. assert n1 == 42 | missing(n1)
```

that would be clear and to the point. It just would not be so general in application.

Focus now on the case of string variables. We can use almost the same idea, but empty strings always sort to the beginning of the dataset. So,

```
. sort s1
. assert s1 == s1[_N] | missing(s1)
```

is the kind of variation needed. Again, what it checks is that all nonmissing values present are equal to each other, including the boundary case in which there are no nonmissing values; and further, that values may be missing. Otherwise put, `assert` will not ignore missing values by default. If you want to ignore missing values, you have to tell `assert` that they are possible.

2.3 Tabulations

By focusing (fixating?) on the use of `summarize` as the first serious idea, we have overlooked what may already have struck you as a simple, general approach: to use `tabulate` and look out for the number of rows shown. I delayed mention of this idea until the notion of returned results was familiar to you.

It is clear enough from the results of `tabulate n1` or `tabulate s1` that just one row is shown in the table; and from the results of `tabulate n2` or `tabulate s2` that two rows are shown. Those are results by default: with the option `, missing` specified, one or more rows will be added to the table if missing values are present. (Why "one or more"? Because with numeric variables, up to 27 different kinds of missing values may be tabulated.)

This is all fine for a small dataset, meaning now one with just a few variables. For a larger dataset, you do not want to be committed to looking through many tables, which is likely to prove tedious, time consuming, and error prone. Rescue from this dilemma comes from the saved result `r(r)`, which is the number of rows in the table. We will pick this up again in the next subsection.

2.4 Several variables

Let us now extend the problem to checking a bundle of variables for constancy while keeping an eye open for the possible occurrence of missing values.

An implication of the last subsection is that it may be prudent to segregate numeric and string variables and treat them separately. If missing values were not an issue, this loop would work either way:

```
. local constant
. foreach v of var * {
  2. capture assert `v´ == `v´[1]
  3. if _rc == 0 local constant `constant´ `v´
  4. }
. display "constant variables: `constant´"
```

The steps here are

1. clear any local macro called `constant`. If you are new to local macros, the executive summary is that they are places to hold text.
2. loop over all variables. The loop here uses `foreach`, so visit its help file if you want more details on the syntax.
3. test that all values of each variable are equal to the first value, so equal to each other. Each test will return true or false. A true result is followed by silence (no output), but a false result would produce an error message and would stop the loop. Putting `capture` in front eats any output, including any error messages, and lets the loop run through without interruption. What saves that approach from futility is that the error code is available immediately after the `capture`d command in `_rc`. Here an error code of 0 means the assertion is true. If that code of 0 is seen, then we have found a constant variable and can add its name to the macro.
4. exit the loop, and `display` the local macro containing the names of constant variables. This macro will be empty if there are no constant variables.

This loop would also work to detect constants if missing values were not an issue:

```
. local constant
. foreach v of var * {
  2. quietly tabulate `v´
  3. if r(r) == 1 local constant `constant´ `v´
  4. }
. display "constant variables: `constant´"
```

Note the prefix `quietly`. We often will not want even to see the tables, just to know how many rows they include. If you do prefer to see the tables, too, then leave off the `quietly`.

You may be familiar with `tab1` as a way to get several one-way tables through a single command. The problem with `tab1` here is that the only `r(r)` accessible afterward is for the last table shown.

This loop is also available as part of the functionality of `findname` (Cox 2010), which is in essence a superset of the official command `ds`, with some changed syntax, through

```
. findname, all(@ == @[1])
```

For the latest implementation of the command, `search findname, sj`.

For a loop for numeric variables, which allows numeric missing values as well as constant numeric nonmissing values, we could use

```
. local nconstant
. ds, has(type numeric)
. foreach v in `r(varlist)´ {
  2. sort `v´
  3. capture assert `v´ == `v´[1] | missing(`v´)
  4. if _rc == 0 local nconstant `nconstant´ `v´
  5. }
. display "numeric constant variables: `nconstant´"
```

If you have been following carefully, you should now be able to write code to find string variables, allowing string missing values as well as constant string nonmissing values. You are allowed to check the syntax of `ds` or use `findname` instead.

If you are energetic enough to want an exercise, you might consider adapting the loop over `tabulate` for whenever there is a need to check for missing values.

3 Constant within subsets of the dataset?

We have saved for last one of the most common variants of the question, which is about subsets of the data. This variant arises particularly with panel or longitudinal data, in which again constancy on some variable may be desirable or undesirable, interesting or not, useful or not.

A common variant is that data come in groups of observations, one group for each family or household. Typically, a variable is specified for only one person, but it makes sense to spread that nonmissing value to other people in the same family or household. So the cautious check needed is that there is no variation in the nonmissing value in each group.

An immediate general answer is to reach for the prefix command `by:`. Suppose that we have a panel identifier `id`. Then, we can extend code to say

```
. bysort id: assert puzzle == puzzle[1]
```

This is easy but not nearly so useful as seen earlier, because the command will not give panel-by-panel results. A loop over panels is in practice not especially helpful either. People with many, many panels in particular usually want more than a long list of panels that satisfy some criterion.

The most useful technique is to calculate an indicator or dummy variable. For a recent overview of technique with such variables, see Cox and Schechter (2019). An indicator allows us to keep all the information in the data on constancy or difference. One line does most of the required work, such as

```
. bysort id (puzzle): generate byte same = puzzle[1] == puzzle[_N]
```

or (according to taste or convenience)

```
. bysort id (puzzle): generate byte different = puzzle[1] != puzzle[_N]
```

In the first, all the observations in each panel are 1 if all the values of `puzzle` are identical and 0 otherwise. In the second, it is the other way round: the indicator has value 1 if there are any differences and 0 if there are none. For this to be so, `puzzle` has to be sorted correctly within each panel, which those two commands achieve.

If missing values contaminate the data, there are various ways forward, including application of the ideas in section 2.3. But perhaps cleanest of all is to segregate missing values as a separate issue. There are many small variations on this method, but this code should give the flavor.

```
. generate byte present = !missing(puzzle)
. bysort present id (puzzle): generate byte different = puzzle[1] !=
> puzzle[_N]
```

An indicator variable has several advantages and makes data management, graphics, and modeling easier. For example, suppose only panels with variation and only nonmissing values are of use. Then,

```
. keep if different & present
```

selects observations accordingly. For examples and more discussion, see Cox (2001).

4 Conclusion

Focusing on which variables are constant in the data has as its complement, focusing on which variables are genuinely variable. The emphasis in this column has been to underline

1. the need to care about numeric and string variables—which may or may not require separate treatment.
2. the need to worry, at least initially, about what any missing values present imply for your purpose.

3. the scope for applying existing commands in a simple way, with at most a few extra constructs such as loops or use of prefix commands, which are all key Stata techniques in any case.

5 References

Cox, N. J. 2001. FAQ: How do I list observations in a group that differ on a variable? http://www.stata.com/support/faqs/data-management/listing-observations-in-group/.

———. 2007. Stata tip 50: Efficient use of summarize. *Stata Journal* 7: 438–439. https://doi.org/10.1177/1536867X0700700311.

———. 2010. Speaking Stata: Finding variables. *Stata Journal* 10: 281–296. https://doi.org/10.1177/1536867X1001000208.

Cox, N. J., and G. M. Longton. 2008. Speaking Stata: Distinct observations. *Stata Journal* 8: 557–568. https://doi.org/10.1177/1536867X0800800408.

Cox, N. J., and C. B. Schechter. 2019. Speaking Stata: How best to generate indicator or dummy variables. *Stata Journal* 19: 246–259. https://doi.org/10.1177/1536867X19830921.

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).