



*The World's Largest Open Access Agricultural & Applied Economics Digital Library*

**This document is discoverable and free to researchers across the globe due to the work of AgEcon Search.**

**Help ensure our sustainability.**

Give to AgEcon Search

AgEcon Search

<http://ageconsearch.umn.edu>

[aesearch@umn.edu](mailto:aesearch@umn.edu)

*Papers downloaded from **AgEcon Search** may be used for non-commercial purposes and personal study only. No other use, including posting to another Internet site, is permitted without permission from the copyright owner (not AgEcon Search), or as allowed under the provisions of Fair Use, U.S. Copyright Act, Title 17 U.S.C.*

*No endorsement of AgEcon Search or its fundraising activities by the author(s) of the following work or their employer(s) is intended or implied.*

# THE STATA JOURNAL

## Editor

H. Joseph Newton  
Department of Statistics  
Texas A & M University  
College Station, Texas 77843  
979-845-3142; FAX 979-845-3144  
jnewton@stata-journal.com

## Associate Editors

Christopher F. Baum  
Boston College  
Rino Bellocco  
Karolinska Institutet, Sweden and  
Univ. degli Studi di Milano-Bicocca, Italy  
A. Colin Cameron  
University of California–Davis  
David Clayton  
Cambridge Inst. for Medical Research  
Mario A. Cleves  
Univ. of Arkansas for Medical Sciences  
William D. Dupont  
Vanderbilt University  
Charles Franklin  
University of Wisconsin–Madison  
Joanne M. Garrett  
University of North Carolina  
Allan Gregory  
Queen's University  
James Hardin  
University of South Carolina  
Ben Jann  
ETH Zürich, Switzerland  
Stephen Jenkins  
University of Essex  
Ulrich Kohler  
WZB, Berlin

## Stata Press Production Manager

## Stata Press Copy Editor

## Editor

Nicholas J. Cox  
Department of Geography  
Durham University  
South Road  
Durham City DH1 3LE UK  
n.j.cox@stata-journal.com

Jens Lauritsen  
Odense University Hospital  
Stanley Lemeshow  
Ohio State University  
J. Scott Long  
Indiana University  
Thomas Lumley  
University of Washington–Seattle  
Roger Newson  
Imperial College, London  
Marcello Pagano  
Harvard School of Public Health  
Sophia Rabe-Hesketh  
University of California–Berkeley  
J. Patrick Royston  
MRC Clinical Trials Unit, London  
Philip Ryan  
University of Adelaide  
Mark E. Schaffer  
Heriot-Watt University, Edinburgh  
Jeroen Weesie  
Utrecht University  
Nicholas J. G. Winter  
University of Virginia  
Jeffrey Wooldridge  
Michigan State University  
Lisa Gilmore  
Gabe Waggoner

**Copyright Statement:** The Stata Journal and the contents of the supporting files (programs, datasets, and help files) are copyright © by StataCorp LP. The contents of the supporting files (programs, datasets, and help files) may be copied or reproduced by any means whatsoever, in whole or in part, as long as any copy or reproduction includes attribution to both (1) the author and (2) the Stata Journal.

The articles appearing in the Stata Journal may be copied or reproduced as printed copies, in whole or in part, as long as any copy or reproduction includes attribution to both (1) the author and (2) the Stata Journal.

Written permission must be obtained from StataCorp if you wish to make electronic copies of the insertions. This precludes placing electronic copies of the Stata Journal, in whole or in part, on publicly accessible web sites, file servers, or other locations where the copy may be accessed by anyone other than the subscriber.

Users of any of the software, ideas, data, or other materials published in the Stata Journal or the supporting files understand that such use is made without warranty of any kind, by either the Stata Journal, the author, or StataCorp. In particular, there is no warranty of fitness of purpose or merchantability, nor for special, incidental, or consequential damages such as loss of profits. The purpose of the Stata Journal is to promote free communication among Stata users.

The *Stata Journal*, electronic version (ISSN 1536-8734) is a publication of Stata Press. Stata and Mata are registered trademarks of StataCorp LP.

# Speaking Stata: Making it count

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

**Abstract.** The `count` command has one simple role, to count observations in general or that satisfy some condition(s). This task can be useful when some larger problem pivots on counting, especially if `count` is used with a loop over observations or variables. I use various problems, mostly of data management, as examples. I also make comparisons with the use of `_N`, `summarize`, and `egen` for the same or similar problems.

**Keywords:** pr0029, count, counting, data management, dropping variables, finding matches

## 1 Introduction

Counting is basic in statistics, graphics, and data management. Often in Stata, counts are produced by some tabulation or other summary command, or as part of the preliminary calculations for a graphics command. However, there are also many problems that pivot on counting something directly. The main ways of doing this in Stata include using `_N`, the number of observations in the dataset or in some subset defined under `by`; the `summarize` command, which reports on the number of numeric values summarized; the `egen` command; and the `count` command. This column focuses on the last as a simple (and helpful) command that users often overlook. Along the way, I will comment on how else you might solve various problems, helping you understand how various Stata features compare.

## 2 The `count` command

`count` counts observations, either in the dataset as a whole or that satisfy some specified `if` or `in` conditions. This role sounds, and is, limited. I will show how useful `count` can be for some larger problems in conjunction with other Stata functionality.

Launch Stata (or clear any data in memory) and type

```
. count
```

Because you have no observations in memory, Stata returns the number 0. Typing

```
. return list
```

shows that a scalar `r(N)` is defined, which contains 0. (If you are not familiar with the idea of saved results in Stata, study the help for `return` and its references as far as

desired.) Being able to use this scalar is even more useful than being able to see a count in your Results window; you can exploit this scalar in do-files and programs or in code that loops over various possibilities.

Now read in some data, say,

```
. sysuse auto
```

and repeat the **count**; you will see a different result. If you read in the **auto** dataset, the count is 74. If you read in another dataset, the count will be the number of observations in that dataset. Type

```
. display r(N)
```

and you will see once more the number produced by the last **count** command. Often you must grab this number while it remains available, because Stata will without warning overwrite it, or even destroy it, the next time you issue an r-class command. (If that term r-class command is unfamiliar to you, study the help for **return**.)

Just **count** alone can be handy. Sometimes when I am typing along and start getting strange results—or no results at all—a quick **count** shows me that I have done something dopey. I might have **cleared** the previous dataset and tried to read in a new one but typed the filename incorrectly, so that Stata gave me an error message. But I was so busy entering commands that the error message did not register. Or I temporarily **expanded** the data but forgot to go back to the original dataset. In cases like these, something is not as expected and a quick **count** often lets me realize what is wrong.

There are, as usual, other ways to do it. Let us recap some other methods for finding the number of observations.

```
. di _N  
. d, s  
. l in l  
. browse in l  
. su
```

You could **display** **\_N**. That is as short and sweet as just typing **count**. You could **describe**, **short**: notice how I invoke the **short** option and use the briefest abbreviations possible. And even though you do not know the number of observations, you can always specify **in l** on a **list** or **browse**. Here **l** stands for **last**, indicating that you want to look at the last observation, which is numbered, so that you can then read off the total number of observations.

As of the 12 January 2007 update, Stata 9 will accept **in L** as an alternative. One use of that variant is whenever you find that a font in use does not differentiate sufficiently between **l** and **1** for immediate comprehension, just as some scientists use the nonstandard abbreviation **L** for liters because they consider the lowercase letter **l** not distinctive enough.

You will be accustomed to seeing a display of number of observations in the output of `summarize`, but the correct answer is not guaranteed: if you have missing values in all numeric variables, the total number of observations will not be shown anywhere in the output of `summarize`.

The last four methods are especially useful interactively. `describe`, `list`, `browse` (or, if you insist, `edit`), and `summarize` all have other uses if you are unclear about the basic characteristics of the data, not just the number of observations but also other details such as variable names or even values. In addition to these possibilities for interactive use, `_N` is an object that you also can use in do-files and programs.

`count` works by looping over observations and counting how many times a condition is true. `count` by itself is really a special or limiting case in which the condition is some condition that is vacuously true. Let us do something silly to emphasize how `count` really works, checking whether 2 is equal to 2:

```
. count if 2 == 2
```

It is as if `count` was thinking this way. Look at the first observation, and consider whether the condition specified is true when you consider the evidence of the first observation. In fact `2 == 2` is true regardless of what is in the first observation, but `count` has no sense of the absurd and will just chalk up 1 as the running total of observations for which the condition specified is true. Now look at the second observation. The story is the same: the condition is true, and so 2 is the new running total. And so on: whenever a condition is true for all observations, the count returned will be precisely the total number of observations.

### 3 if and in conditions

Naturally, finding out something about your data is more useful and interesting. With the `auto` dataset read in,

```
. count if mpg > 30
```

and many other such statements provide answers to simple questions. Here `mpg > 30` includes all observations that are missing on `mpg`, as numeric missing (`.` or any of `.a` to `.z`) counts as arbitrarily large. There are no missings on `mpg` in the `auto` dataset, but using

```
. count if inrange(mpg, 30, .)
```

is generally safer (Cox 2006) unless you really do want to catch the missings as well.

Now check your understanding of

```
. count if foreign
```

If the idiom is unfamiliar: `if foreign` is equivalent to `if foreign != 0` because in Stata nonzero corresponds to true and zero, to false. `foreign` is an indicator (binary,

dummy) variable that takes just two values in the `auto` dataset, 1 for foreign cars and 0 for domestic cars, both definitions from a United States point of view. Experienced Stata users become accustomed to shorthand versions like `if foreign`, although experienced users can get bitten too whenever an indicator variable such as `foreign` is missing. If you are averse to showing off with flourishes like `if foreign` or `if !foreign`, then spelling out what you mean does no harm:

```
. count if foreign == 1  
. count if foreign == 0
```

On the other hand, typing syntax like `if female` or `if !female`, articulating to yourself *if female* or *if not female*, can seem natural. (This approach leads to often given, and often ignored, advice to name indicator variables according to whatever is coded as one: thus if you coded 1 for female and 0 for male, then `female` is a better variable name than “gender” or “sex”. Avoid the position of the researcher who was asked after presenting his group’s results which way the gender variable went but could not recall.)

An `in` condition by itself is legal but not useful. `count in 1/20` will dutifully report 20, so long as there are at least 20 observations, but you learn nothing thereby. However, `if` and `in` conditions together can be useful. Suppose that you `sort` on some variable. Then you might be interested in asking questions about the observations containing the lowest values of that variable. For that you can combine `if` and `in` conditions, as in

```
. sort weight  
. count if mpg > 30 in 1/20
```

This sequence asks a question about the 20 observations with the lowest values of `weight`, the 20 lightest cars: how many have `mpg` that is more than 30 miles per gallon (or missing)?

## 4 Finding matches

### 4.1 A problem and a rough solution

Let us now turn to some more challenging problems.

First, imagine a dataset in which various people indicate their best friends. This may not be anyone’s real problem, but it is easy to think about and is similar to many other problems. The more general issue can be thought of as *finding a match*. So imagine a variable `id` in which we have names “Patricia”, “Bobby”, “David”, “Vince”, “Ana”, and so on, and a variable `bestfriend` in which these people name their best friend. One simple question is how many people name someone named in `id` as their best friend? The answer could be any integer from 0 upward.

Before we do anything, we should check the data. The implication is that each person in `id` should occur just once, so we can check that by

```
. isid id
```

or

```
. duplicates report id
```

The first is the more direct way of answering that question. You may know that there is a lower-level way of doing it:

```
. by id, sort: assert _N == 1
```

which evidently would change the order of the dataset, unless it was already sorted by `id`.

Enough procrastination; how are we going to answer this question: how many people name someone in `id` as their best friend? The hint of `by:` in the last Stata statement may suggest that we need to do something `by id:`, but that is not the answer. The instruction `by id:` would divide the dataset into as many subsets as there are observations, which would not help; we need to look beyond each observation to count how many matches there are.

One way to do it is by looping over the observations and just counting matches. Imagine doing it by hand. We certainly will not need to do that in practice, but let us take one step at a time. If we look at the first observation, we can count matches like this:

```
. count if id[1] == bestfriend
```

This command loops over all the observations. It compares `id[1]` to `bestfriend[1]`, then `id[1]` to `bestfriend[2]`, and so on, all the way down to the last observation. `count` will display a number and then put the result in `r(N)`. We would then need to do the same for `id[2]`:

```
. count if id[2] == bestfriend
```

But before we do that we should put the result for `id[1]` somewhere safe. In fact, before we do any counting, we should initialize a variable,

```
. gen n.bestfriend = 0
```

which leads to the idea that our scheme will be something like (in total)

```
. gen n.bestfriend = 0
. count if id[1] == bestfriend
. replace n.bestfriend = r(N) in 1
. count if id[2] == bestfriend
. replace n.bestfriend = r(N) in 2
```

and so on for all the other observations. (If you do not know about `generate` and `replace` and especially the difference between them, check out `help generate` or [D] `generate`.)

First comes the initialization that we need to do, ensuring that there is a place to put the counts, the new variable `n_bestfriend`. Second and third, each `count` automatically loops over the observations, looking in turn at each value of `bestfriend`, but we also need to arrange to loop over the values of `id`. Fourth, we pick up the value of `r(N)` as soon as it is produced, and we put it safely into a value of `n_bestfriend` before the next `count` overwrites it.

In principle, this is an answer to the question, albeit not a good one. We would need to repeat similar statements for every observation, which we do not want to do. Also, this answer will be noisy: every time Stata does a `count` or a `replace`, you will get a line of output that you will not care about, except insofar as it indicates that things are going according to plan. But we can fix those things.

Be clear, meanwhile, why

```
. count if id == bestfriend
```

is not the answer. This command loops over the observations but counts just those people who name themselves as their own best friends. The loop consists of checking whether `id[1] == bestfriend[1]`, `id[2] == bestfriend[2]`, and so forth. And it produces one number, whereas we want a variable indicating the counts for each individual.

## 4.2 Automating the loop over observations

Going back to our solution, we can make it quieter and loop automatically like this:

```
. gen n_bestfriend = 0
. quietly forval i = 1/'= _N' {
.     count if id['i'] == bestfriend
.     replace n_bestfriend = r(N) in 'i'
. }
```

`quietly` simply suppresses all output (except for any error messages) from the commands in the `forval` loop. `quietly` can be abbreviated down to `qui`. The braces here mark the body of the loop. The indenting here is, to Stata, purely cosmetic, but you should think of indenting as good writing style, making clear which statements belong together in the loop. If you get into the habit of indenting such blocks when you write, and do not postpone tidying up code, then your code will remain more readable.

`forval` is the most common abbreviation of `forvalues`, although you can also use `forv`. If you are not familiar with `forvalues`, then check out the online help, or [P] `forvalues`, or the more discursive tutorial in Cox (2002a). Or read these next paragraphs carefully.



`forvalues` loops over a set of numbers. It is fussy about what numbers it will accept: only arithmetic progressions, sequences (either rising or falling) with constant step. Here the set is specified as `1/'= _N'`. The second part that is bound in `'= '` instructs Stata to go off and evaluate the expression given, which is `_N`, and then to insert the result in the same place. That is, the effect is to insert in the command the total number of observations. Suppose that we have a file with data on 789 people. Then `_N` will be 789, and `forvalues` will get the instruction that begins

```
. quietly forval i = 1/789 {
```

Naturally, if you remembered that the number of observations was 789, you could type that directly, but you would then need to change the code if you ran it on a dataset of different size, which means almost always. The sequence `1/789` includes all the integers in that range. In our problem, we are going to loop over all the observations so that our sequence starts with 1 and ends with the total number of observations.

The `forvalues` command set up a counter, here called `i`, which technically is a local macro. Here it is set in turn to 1, 2, and so on, up to 789. Inside the loop, refer to the counter as `'i'`. Whenever Stata sees that reference to the counter, it will substitute the current value of the counter before trying to execute the command. If you had called the counter `j`, you would refer to it inside the loop as `'j'`. First time around the loop, the commands become

```
. count if id[1] == bestfriend
. replace n_bestfriend = r(N) in 1
```

and second time they become

```
. count if id[2] == bestfriend
. replace n_bestfriend = r(N) in 2
```

and so on. Because this is all done under the aegis of `quietly`, you will see no results unless you made an error: Stata would then break through the `quietly` and print an error message. In practice, Stata people often prefer noisy output until they are confident that the code is correct. Or they start out insisting on `quietly`, find an error, and then change the code to show results until they can see what is wrong.

Our loop began

```
. quietly forval i = 1/'= _N' {
```

We could also have exploited the main message of this column and begun

```
. quietly count
. quietly forval i = 1/'r(N)' {
```

or

```
. quietly count
. quietly forval i = 1/'= r(N)' {
```

Either pair of statements would call `count` and then use the scalar `r(N)`, as before. We cannot just put `forval i = 1/r(N)` since `forvalues` expects the numbers defining the arithmetic progression to be explicit. But either `'r(N)'` or `'= r(N)'` obliges Stata to evaluate `r(N)`, so that `forvalues` sees the numeric result, 789 or whatever, and not what you typed, which was caught and interpreted first by Stata. There is little to choose between the two possibilities. Think of `'r(N)'` like this: it is as if `r(N)` had a local macro persona. `'= r(N)'` is using the more general idea that we have already seen, of instructing Stata to evaluate an expression on the side and then to insert the result of that evaluation at the same point in the command. Cox (2003) discusses this last idea.

But we did not use `count` in this way. No harm would have been done if we had, but if you know that you want the total number of observations, using `_N` can be simpler and more direct. If you want some other number of observations, then consider using `count` to get the result.

In thinking about any code, focusing on what might go wrong is always a good idea. One possible source of small problems is that comparison of `id` and `bestfriend` is utterly literal: either the string values are equal or they are not. Potential problems might include leading or trailing spaces, spelling mistakes, and inconsistent capitalization. Thus if `id` included "Jack" and one of Jack's friends specified "JACK" as her best friend, that is not the same to Stata. Some of these problems are easy (to remove marginal spaces, just use the `trim()` function), but others will require more attention. But let us leave these possible difficulties on one side.

### 4.3 Other matching problems

Having worked out the basic idea, we could use it in similar problems. We might have other variables, such as `female` (remember the suggestion made earlier about indicator variable names?), and be interested to work out whether best friends were of the same gender. The `count` command restricting matches to those of the same gender would be just

```
. count if id['i'] == bestfriend & female['i'] == female
```

Other extensions could be to data on friends of any degree, held in either multiple observations or multiple variables. Indeed, people in many fields work with interaction or transaction data of various kinds, including data on war (aggressor, victim), trade (importer, exporter), and sports (home team, away team), so that problems like those of finding matches are more common than you may have guessed from the initial example.

## 5 Dropping variables with no useful values

### 5.1 The problem and a solution using summarize

As a second substantial example, consider the following problem. Sometimes variables contain no useful values. The most obvious example is that all values in a variable are missing. This missingness could be just empirical: for some reason, all the values are unknown. Or it could arise for some computing reason. Some people use blank columns in spreadsheets to separate blocks of data. When read into Stata, blank columns will typically turn into variables full of missing values. Researchers often see no point in having such a variable in their dataset, especially if they are short of memory, so the question is how to identify such variables so that they can be dropped. See `help drop` or `[D] drop`. As `search missing` indicates, a user-written program, `dropmiss`, exists to drop variables or observations that are all missing (Cox 2001).

Other examples are easily possible: on Statalist someone expressed a wish to `drop` variables whenever they contained only missing values or zeros. Presumably zeros were also a kind of missing or irrelevant value as far as the researcher's project was concerned. This variant, beyond the scope of `dropmiss`, emphasizes a simple fact of life: user-written programs are all well and good, but your problem may be rather different from theirs. At that point, you need to see if you can solve the problem yourself. Let us walk our way through this variant.

For the moment, focus on the case in which all the variables are numeric. (It can take some experience of being tripped up when that is not true, because some variables are string, to become accustomed to considering the possibility of string variables.) With a small dataset, just looking at the data and dropping the variable if it qualifies is easy. But the challenge is clearly to automate the operation.

Faced with this problem, some users recall that `summarize` emits the number of observations being summarized. If we have some numeric variable `foobar`, then we could

```
. su foobar if foobar != 0
```

This command excludes any zeros explicitly. As you should be aware, `summarize` will exclude any missings automatically. If all values are either zero or missing, then `summarize` will return a scalar `r(N)` that contains 0. How do I know this? Partly from reading the documentation for `[R] summarize` but more from creating a tiny toy dataset, playing with various made-up variables, and looking at the results to see what is left in memory.

Let us develop this into a fuller solution. (Later we will look at an even better solution using `count`.)

Suppose that `foobar` turned out to yield `r(N)` of 0. Then we want to drop the variable. In Stata terms, this would be

```
. if r(N) == 0 drop foobar
```

This command is not the same as

```
. drop foobar if r(N) == 0
```

For one thing, the above command is illegal, as the documentation for **drop** will confirm. **drop** is a dangerous weapon that can do a great deal of damage to datasets, so prudence puts limits on what it can do: just one thing at a time, dropping variables or observations, but not both. Even if it were legal, the second form would not capture the exact logic. We make just one decision—if **r(N)** is zero, then we **drop** the variable, but otherwise we keep it—not a decision for every observation, as the second form would imply.

We could improve on this. For one thing, **summarize** by default does a moderate amount of work, most of which does not interest us here. The **meanonly** option is helpful here.

```
. summarize foobar if foobar != 0, meanonly
. if r(N) == 0 drop foobar
```

fires up the smallest subset of **summarize** that we can invoke. The name **meanonly** could be misleading: you might think, understandably, that only the mean is produced with this option. However, **summarize**, **meanonly** also yields the number of observations, as well as a few other results. It has the useful side effect of suppressing output, unless there are error messages. It is thus expected that the user will pick up what is desired from the saved results, as here.

This is all well and good for one variable, **foobar** or whatever else, but we need to loop over all the variables in the dataset. We know that variables might be string, but we will continue to focus on the numeric case.

The looping command **foreach** allows us to loop over variables, as we used the command **forvalues** to loop over observations in the earlier problem. As before, if you are not familiar with **foreach**, then check out the online help, or [P] **foreach**, or the more discursive tutorial in Cox (2002a). Or read these next paragraphs carefully.

```
. quietly foreach v of var * {
.     summarize 'v' if 'v' != 0, meanonly
.     if r(N) == 0 drop 'v'
. }
```

Outside the loop we declare a name for a local macro, here **v**. Inside the loop we can refer to that with **'v'**. The word **var** spells out that we want to loop over a variable list. What follows is a wildcard, **\***, specifying all variables. (If you have some previous experience of Stata programming, you know that this wildcard will include any temporary variables.)

The loop will work in this way. The wildcard **\*** will be expanded to the complete list of variable names. **foreach** will substitute each name in turn in the commands in the body of the loop and carry out the instructions. Because the whole is done **quietly**, you will see no output unless error messages are produced.

The examples here use the wildcard \*. Another way of specifying all variables, which you may know, is `_all`. That would be fine instead, and indeed a little more efficient.

Now we confront the question of what will happen if a variable is string. Applying `summarize` to a string variable is perfectly legal. `summarize` will return `r(N)` of 0, which is best thought of this way: there are no numeric values to be summarized. This finding may sound consistent with our approach so far, but it will happen even if the string variable never contains string missings or string zeros. An even bigger problem: if `foobar` is string, then

```
. su foobar if foobar != 0, meanonly
```

is illegal because `if foobar != 0` is a numeric comparison and only string comparisons are allowed with string variables (so that `if foobar != "0"` would be legal).

There are various possibilities for what we can do.

You might know that there are no string variables, in which case no problem arises in practice. `ds, has(type string)` is one way of checking for string variables.

You might decide that you are not attempting to drop string variables. Then all you need to do is catch any error that is produced. The `capture` command is useful here.

```
. quietly foreach v of var * {  
.     capture summarize 'v' if 'v' != 0, meanonly  
.     if _rc == 0 & r(N) == 0 drop 'v'  
. }
```

`capture` eats the error that arises if the variable is string. If the error occurs, the `summarize` command is not executed, having been declared illegal. Also, the nonzero return code that would have been displayed if `capture` had not been specified is accessible in `_rc`. That is crucial for automation.

The condition that is specified for a `drop` to take place is now twofold. First, the command must have been legal and successful, so that a zero return code was returned. Second, as before, `r(N)` must be 0. Both conditions are important. If the `summarize` command is not executed, then either there is no `r(N)` in memory—which could be so if the `summarize` command that was tried was the first in the loop—or there is an `r(N)` in memory, returned by the previous successful `r-class` command. Thus, any result in memory does not refer to the present variable and could lead to the wrong decision, even dropping a variable you would want to keep. That is why we need both `_rc == 0` and `r(N) == 0`.

To complete the picture, consider what happens if you refer to `r(N)` but no such scalar is in memory. Then the result is taken to be missing, which is fine for our problem. It means that no `drop` will take place accidentally even if `r(N)` is missing.

Yet another possibility is to make sure that the commands are applied only to numeric variables. We can use `ds` as a preliminary filter:

```
. ds, has(type numeric)
. quietly foreach v in `r(varlist)' {
.     capture summarize `v' if `v' != 0, meanonly
.     if _rc == 0 & r(N) == 0 drop `v'
. }
```

or we could check in advance that each variable is indeed numeric:

```
. quietly foreach v of var * {
.     capture confirm numeric var `v'
.     if _rc == 0 {
.         summarize `v' if `v' != 0, meanonly
.         if r(N) == 0 drop `v'
.     }
. }
```

## 5.2 A solution using count

We have looked in some detail at a solution that pivots on `summarize`. That is a practical solution, but a better one pivots on using `count` instead.

We need not recapitulate the discussion of trapping string variables and can show one such solution immediately:

```
. quietly foreach v of var * {
.     capture confirm numeric var `v'
.     if _rc == 0 {
.         count if `v' == 0 | missing(`v')
.         if r(N) == _N drop `v'
.     }
. }
```

This solution is aimed directly at the heart of the matter and does nothing that is not related to the problem. (Even `summarize`, `meanonly` will have done some irrelevant work whenever there were nonzero and nonmissing values.) Here we `count` zeros and missings and check whether the number is equal to the number of observations `_N`. If it is, then all the values of the variable being examined are missing or irrelevant, so we can `drop` it. You may prefer to invert the comparison, as in

```
. count if !(`v' == 0 | missing(`v'))
. if r(N) == 0 drop `v'
```

These solutions are open to the possibility of inserting code for some different treatment of string variables (rather than just doing nothing). The most natural possibility seems to be just to check for missings:

```

. quietly foreach v of var * {
.     capture confirm numeric var 'v'
.     if _rc == 0 {
.         count if 'v' == 0 | missing('v')
.         if r(N) == _N drop 'v'
.     }
.     else {
.         count if missing('v')
.         if r(N) == _N drop 'v'
.     }
. }

```

We can now simplify to

```

. quietly foreach v of var * {
.     capture confirm numeric var 'v'
.     if _rc == 0 count if 'v' == 0 | missing('v')
.     else count if missing('v')
.     if r(N) == _N drop 'v'
. }

```

which is where we will leave the problem.

## 6 by and egen

`count` is useful, but exaggeration is not helpful to anyone, so one point deserves emphasis. Many counting problems do not require recourse to `count`, or indeed to loops over observations or variables. Perhaps most counting problems can be solved with the use of `by:` and/or `egen` and no user-defined loops. For more on those features, see the documentation or Cox (2002b,c).

As just one example, consider a dataset on people in families with family identifier `family`, gender variable `female`, and age in years `age`.

How do you calculate the number in each family?

```

. by family, sort: gen number = _N

```

or

```

. egen number = total(1), by(family)

```

How do you calculate the number in each family of each gender?

```

. by family female, sort: gen ngender = _N

```

or

```

. egen ngender = total(1), by(family female)

```

(What's a total count? It is  $1 + 1 + \dots + 1$ , with summation over all the relevant observations.)

How do you calculate the number in each family aged 20 and under?

```
. egen under21 = total(age < 21), by(family)
```

and so on.

## 7 Conclusion

Counting is important. The issue is how to count when familiar commands offer no apparent solutions. Often the answer is to reach for the `count` command, sometimes with a loop over observations or variables.

## 8 References

- Cox, N. J. 2001. dm89: Dropping variables or observations with missing values. *Stata Technical Bulletin* 60: 7–8. Reprinted in *Stata Technical Bulletin Reprints*, vol. 10, pp. 44–46. College Station, TX: Stata Press.
- . 2002a. Speaking Stata: How to face lists with fortitude. *Stata Journal* 2: 202–222.
- . 2002b. Speaking Stata: How to move step by: step. *Stata Journal* 2: 86–102.
- . 2002c. Speaking Stata: On getting functions to do the work. *Stata Journal* 2: 411–427.
- . 2003. Speaking Stata: Problems with lists. *Stata Journal* 3: 185–202.
- . 2006. Stata tip 39: In a list or out? In a range or out? *Stata Journal* 6: 593–595.

### 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 wrote several inserts in the *Stata Technical Bulletin* and is an editor of the *Stata Journal*.