The Stata Journal (2011)
11, Number 2, pp. 305–314

Speaking Stata: Compared with . . .

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

Abstract. Many problems in data management center on relating values to values in other observations, either within a dataset as a whole or within groups such as panels. This column reviews some basic Stata techniques helpful for such tasks, including the use of subscripts, `summarize`, `by:`, `sum()`, `cond()`, and `egen`. Several techniques exploit the fact that logical expressions yield `1` when true and `0` when false. Dividing by zero to yield missings is revealed as a surprisingly valuable device.

Keywords: dm0055, data management, panels, subscripting, summarize, by, sum(), cond(), egen, logical expressions

1 Introduction

Often in data management, there is a need to compare values in a variable with other values for different observations (rows, cases, or records in non-Stata terminology).

An easy first example is relating values to a single reference value in another observation: suppose, say, that you regard Texas or London or your home area as a reference, or that you wish to scale values relative to some base year, such as 1980 or 2000. A more challenging version of that problem repeats that calculation for different groups, say, panels in a longitudinal dataset. A yet more challenging version entails summarizing observations for a subset of the same group, such as when the characteristics of the children in a family are calculated for all observations in that family.

In this column, I review basic Stata techniques in this area. Some more-complicated problems in this territory were discussed in an earlier column (Cox 2002b), so in a sense this column should have been written first.

Even Stata veterans are likely to find something new here—they should particularly look at section 10 on dividing by zero, which can be an eminently logical move.

2 Subscripting identifies specific observations

We will start with using some other observation as reference. Let us read in some data:

```
. sysuse uslifeexp
```

This dataset contains various time series for life expectancy in the United States. Suppose we want to relate changes to the base year 1960. In this case, the dataset is of modest size and it is easy to find that values for 1960 are in observation 61. So we could use subscripting to relate values to that observation, as in

```
. generate le_female_index = 100 * le_female/le_female[61]
```

The advantage of this method is directness. The value desired as reference is already a data value, so there is no need to calculate it. Once we have found out which observation contains the value we need, we can just indicate the observation number by a subscript, here `[61]`. As is evident, the terminology here is not literal, because Stata does not write anything below the line, but is merely intended as evocative, calling to mind notation such as y_i, specifically, y_{61} to indicate the 61st value in a series.

The disadvantages of this method are a little more subtle. Suppose that we are careful and keep a record of our calculations, but we are not so careful as to add a comment explaining exactly what this calculation does. Then there could be a minor puzzle working out some time later what it is that we did. Or suppose that we change the sort order of the data for some reason. Then the new observation 61 is very likely to be a different observation, and the same command line would yield a different calculation, as we may or may not realize. Or suppose that we want to do something like this in different datasets, in which there is no predictable regularity about which observation contains the value we want. Then the lack of generality of the method is clear.

3 summarize leaves useful results in its wake

More general methods are at hand. If we

```
. summarize le_female if year == 1960
```

then the `if` condition will in this dataset identify just one observation, and the value of `le_female` for that observation will be accessible after `summarize` in one of `r(min)`, `r(mean)`, or `r(max)`. Conversely, if there are no such observations, or more than one such observation, `summarize` will tell you, and you need to work out what to do next. But suppose all is well, as in the dataset in question. Then the calculation to follow will be

```
. generate le_female_index = 100 * le_female/r(mean)
```

Clearly, you may use `r(min)` or `r(max)` rather than `r(mean)`, if you please, but the difference is quite immaterial, because the mean, minimum, and maximum of a single value are all identical to that value.

This method has the advantage over the first method of being easier to understand in a log file consulted some time after the event. Naturally, adding an explanatory comment would make it even easier.

What could go wrong? Suppose that there is no value for 1960. Then after summarize, the results would be missing and so would our new variable. Thus we would notice that problem sooner or later.

Conversely, suppose that there are two or more observations for 1960. As said, the fact will be evident in the results for summarize. If we were automating calculations, it would be a good idea to check that the number of nonmissing values, accessible after summarize in r(N), is equal to 1.

4 Tagging gives something to copy

Here is yet another way to do it. Like the summarize method just explained, this method is more elaborate than the first method, but it can be greatly extended to more complicated and more challenging problems. First, create an indicator tag or flag variable that is 1 for the observation we want to copy.

```
. gen byte tag = 1 if year == 1960
```

Notice the detail of creating a byte variable to reduce storage. This new variable, tag, will be 1 when the year is equal to 1960 and numeric missing (.) when the year is not equal to 1960. We can now

```
. sort tag
```

After sorting, the observation for the tagged year, 1960, will be sorted to observation one. The index now can be created by

```
. gen le_female_index = 100 * le_female/le_female[1]
```

You can see the advantage of this technique. It is a way of making Stata first find and then use the value for 1960, regardless of where it occurs in the dataset or whether the dataset had any particular sort order.

There are many variations on this indicator variable technique. If the tag variable had been

```
. gen byte tag = year == 1960
```

then the new variable would be 1 if the year was 1960 and 0 otherwise. Thus sort tag would sort the tagged observation to the end of the dataset, and the appropriate value would be referenced by le_female[_N]. Using _N as subscript is a common Stata technique. _N in this situation is the number of observations in the dataset and so also the subscript of the last observation in the dataset. The more general principle is that subscripts, like much else in Stata, can be defined by expressions, which Stata will evaluate for you.

The expression year == 1960 is just one example of a logical test yielding 1 or 0 on evaluation. The help file for operators will remind you of the other operators that can appear in logical expressions; see help operators. For example, negating the logical test by `!(year == 1960)` would have meant that the tagged value would have been sorted to the start of the data. Negation using `!` exchanges 0s and 1s.

Again, what could go wrong is that there might in fact be no value for 1960. In that case, the tag variable will be identical throughout the dataset and the wrong value would be used. So we would need to be more careful with this technique. For example, if `tag` should be `1` in the first observation, check whether that is true. Similarly, it would be prudent to check that only the first observation was tagged with `1`. Possible techniques include using `summarize`, as above, or `count` (Cox 2007), or `assert` (Gould 2003).

5 The problem arises with panels too

Let us now consider extending the problem to panel data. We will use another of Stata's datasets:

```
. webuse grunfeld, clear
```

This is a well-behaved panel dataset in which each year in the dataset is matched by a nonmissing value for each panel and each variable. But we show a technique that does not make that assumption. The panel covers the period 1935 to 1954. Let's show how to scale each panel separately by values in 1939.

```
. gen byte baseyear = 1 if year == 1939
. by company (baseyear), sort: gen invest_index = 100 * invest/invest[1]
```

What is new here is that we are using `by:` to calculate separately within the groups it defines. The last command does three things in quick succession:

1. It declares that operations will be done separately by `company`.
2. It sorts first on `company` and then within `company` by the new variable `baseyear`. Because `baseyear` has values `1` or missing, values of `1` will be sorted to the start of each panel for an individual company.
3. It creates a new variable by using `generate` and the expression given. A key feature of using `by:` is that subscripts are interpreted within groups rather than within the entire dataset. Thus the subscript `[1]` refers to the first observation for each `company` in the current sort order.

The two lines of code above will give incorrect answers if in fact an individual company was not observed in 1939. In a real time-series problem, researchers would be likely to solve such a problem by interpolation or some other suitable method. For our purposes, let us focus on ensuring results that are not misleading. One useful method is to extend the expression

```
100 * invest/invest[1]
```

to say

```
    100 * baseyear[1] * (invest/invest[1])
```

The extra factor `baseyear[1]` will be `1` or missing, and so multiplying by it will leave valid values unchanged but will result in invalid values being returned as missing. Again, using subscript `[1]` ensures that the value for the first observation is used for every observation in its panel.

A twist on this problem is that values are to be related not to a reference year, but to a reference panel. A little thought shows that as far as Stata is concerned, this is exactly the same kind of question.

```
    . gen byte basepanel = 1 if company == 1
    . by year (basepanel), sort: gen invest_index = 100 * invest/invest[1]
```

The same caveat applies if there are gaps in the data.

For more on `by:`, see a tutorial in a previous column (Cox 2002a).

6 Use logical expressions and sum()

What do the following lines of code do?

```
    . sysuse uslifeexp, clear
    . gen basevalue = sum((year == 1960) * le_female)
    . gen le_female_index = 100 * le_female/basevalue[_N]
    . webuse grunfeld, clear
    . by company, sort: gen basevalue = sum((year == 1939) * invest)
    . by company: gen invest_index = 100 * (invest/basevalue[_N])
```

This new method uses the function `sum()` together with logical expressions. The expressions `year == 1960` and `year == 1939` will be evaluated as `1` when true and `0` when false. The cumulative sum produced by `sum()` will, at the end of the dataset, just be the total for `le_female` or `invest` in the year identified. It is born zero but is set to `le_female` for 1960 when that observation is encountered. So the last value, subscripted by `_N`, will be the value we need. A big advantage of this technique, as shown by the second code example above, is that it extends easily to panel data.

With this technique, we need to ensure that there is only one value for each set identified, the entire dataset for the first example and each individual company in the second example. In fact, if there are no instances of the years indicated, the variables containing sums will be 0 and, in this case, dividing by 0 will yield missing, which is the right answer. But not all such problems imply division by 0, so in other problems we still need to watch out. And the difficulty still remains that there might be two or more observations for the years in question. In careful code, we would therefore also monitor `sum(year == 1960)` or `sum(year == 1939)` and check that its final values are `1` when that should be the case.

7 egen is an alternative

You may well know that egen offers a canned way to create the variables of the previous section.

```
. sysuse uslifeexp, clear
. egen basevalue = total((year == 1960) * le_female)
. gen le_female_index = 100 * le_female/basevalue

. webuse grunfeld, clear
. by company, sort: egen basevalue = total((year == 1939) * invest)
. by company: gen invest_index = 100 * (invest/basevalue)
```

A feature of egen's total() function, which is sometimes overlooked, is that it feeds on expressions, which can be more complicated than single variable names.

The similarity between the egen code here and the code of the previous section is no accident. What egen is doing is in essence identical to what was done with generate previously. Evidently, generate is a fundamental command that we all must use. Given that, the reaction might well be: Why bother with egen?

The differences here, which do provide an answer to that question, are all a little subtle.

First, the egen function is called total() in an attempt to underline that it is not the same beast as the function sum(). Long-time users of Stata will know, however, that before Stata 9 this function *was* called sum(), and in fact there is still an undocumented egen function called sum(), which is identical to total() in effect. To explain that again: in the context of egen, and only in that context, sum() means the undocumented egen function with that name, which produces overall sums or totals, and not running or cumulative sums or totals. For more on functions in the strict sense and egen functions, see Cox (2002b).

Second, egen produces a new variable that will be a constant within its group. Hence, there is no need to be careful about picking up the last value, subscripted with [_N], although there is no harm in specifying that subscript.

Third, egen includes many details in its implementation that are glossed over here. In total, those details make egen a little inefficient in machine time. But they add to its utility. In particular, egen takes much of the pain out of handling any extra if or in restrictions.

egen does not solve the particular difficulty of our running example that there might be more than one instance of year == 1939, or whatever. We could also keep track of the number of occurrences by (for example)

```
. by company, sort: egen basecount = total(year == 1939)
```

because the total of 0s and 1s is exactly the same as the count of 1s. Although there is an egen function called count(), that function is not what is wanted here.

```
. by company, sort: egen basecount = count(year == 1939)
```

would count how many times the expression `year == 1939` yielded nonmissing values, and 0s and 1s both qualify as nonmissing. So in this example, `egen, count()` would just return the number of observations in each panel.

8 egen has other applications

However, there is much that is positive about `egen`. `egen` offers a rich variety of functions for summarizing groups and so is the way to solve many more-complicated problems. For our next set of examples, we will load in Stata's `auto.dta` and imagine that we wish to compare according to the `foreign` variable, which distinguishes domestic cars, those made in the United States and for which `foreign` is `0`, and foreign cars, those made outside the U.S. and for which `foreign` is `1`.

```
. sysuse auto, clear
```

Many basic summary statistics can be produced for groups using one or more of `egen`'s functions and working under the aegis of `by:`, such as, for example, the mean weight:

```
. by foreign, sort: egen mean_weight = mean(weight)
```

Now let us imagine that we want to use cars with high `mpg` as a reference group. Inspection of the data shows that seven domestic and seven foreign cars have `mpg` over 25 miles per gallon, so using those as reference would be a fair thing to do.

```
. by foreign, sort: egen mean_weight_2 = mean(weight) if mpg > 25
```

is one way to use only those cars in the calculation. But the side-effect of using `if mpg > 25` is that this new variable is defined only for those cars above 25 miles per gallon. If we want to assign the new means to all observations in the same group, we can fix that with a variant on the `sort` and `replace` commands used earlier.

```
. by foreign (mean_weight_2), sort: replace mean_weight_2 = mean_weight_2[1]
```

The `sort` option of `by:` will sort the nonmissing values of `mean_weight_2` so that they are now the first block of observations within each group defined by `foreign`. At worst, there might be no observations within a group satisfying the `if` condition. If so, `mean_weight_2` will be created missing for all observations within a group, and the `replace` option will just replace missings with missings, which is not a problem.

9 Use cond() instead

Here is another way to get the variable wanted in the last section, but in a single command line. It is another exploitation of the fact that many `egen` functions can take an expression as argument, which need not be as simple as a single variable name.

```
. by foreign, sort: egen mean_weight_3 = mean(cond(mpg > 25, weight, .))
```

shows how to get that variable using the function `cond()`. The expression that is being supplied to the `egen` function `mean()` is

```
    cond(mpg > 25, weight, .)
```

which is `weight` when `mpg > 25` and missing otherwise. What is key here is that the `mean()` function, like most of Stata's statistical commands and functions, ignores missing values: they contribute neither to the sum of values nor to the count of values, and so not to the resulting mean, which is sum/count. But the resulting mean will nevertheless be assigned to each observation. If no values within a group satisfied the condition `mpg > 25`, then the mean would be returned as missing, but that is expected and reasonable.

We could not here use the trick of averaging

```
    (mpg > 25) * weight
```

because that would average a mix of `weight` when we wanted it and 0 when we did not. We could use that idea in calculating the sum and the count and then the mean, but that would be too round-about to be interesting.

A tutorial on `cond()` can be found in Kantor and Cox (2005).

10 Dividing by zero can be logical

Consider this alternative:

```
    . by foreign, sort: egen mean_weight_3 = mean(weight/(mpg > 25))
```

Using a principle we have exploited already, the logical expression `mpg > 25` evidently yields `1` for true and `0` for false. So, when we divide by the result of a logical expression, we divide by either 1 or 0.

When we divide by 1, we leave the numerator, here `weight`, as it was. In mathematics, the result when we divide by 0 is indeterminate. In Stata, such a result is returned as numeric missing (.). In this example, that is exactly what we want, because we want `egen`'s `mean()` function to ignore those observations. As emphasized in the last section, missings are ignored by most of Stata's statistical code.

We could use the device elsewhere. What is the cheapest car with `mpg > 25`?

```
    . by foreign : egen min_price = min(price/(mpg > 25))
```

Nothing stops the same variable being used in numerator and denominator of the expression.

In families, what is the age of the youngest adult?

```
. by family, sort: egen youngest_adult = min(age/(age >= 18))
```

Nothing stops the denominator expression being more complicated, according to what is needed.

What is the age of the youngest woman adult?

```
. by family : egen youngest_f_adult = min(age/(age >= 18 & female == 1))
```

We should note a couple of reservations. Specifically, watch out with tests such as `mpg > 25` or `age >= 18`, which will catch missings on `mpg` or `age` if they exist, because missings count as higher than any nonmissing value. Generally, note that this technique is not quite so general as using `cond()`, which has scope to assign a nonmissing alternative for the false branch of its condition.

Nevertheless, this trick might appeal on various grounds, partly its conciseness and partly its unexpectedness. Years of education and experience may well have accustomed you to the idea that dividing by zero is to be avoided, but there are occasions in Stata when it can be entirely logical.

11 Conclusions

Various basic techniques can be useful in this territory.

Subscripting.
: As in many other environments and languages, Stata supports subscripting to identify particular observations.

`summarize`.
: The `summarize` command can identify particular values, which are then accessible as saved results.

Logical expressions yield 1 or 0.
: Such results can be exploited in various ways, notably, through addition or multiplication.

Sorting.
: Using `sort` to put the value you want at the beginning or end of the group it belongs to is a good way to make it identifiable and thus easy to copy.

`by:` is your friend.
: With a good understanding of `by:`, doing something for every group (for example, panel) need not be more difficult than doing it for the whole dataset.

`egen` is a useful workhorse.
: Many of its functions work with `by:` too.

Many functions feed on expressions.
Expressions can be more complicated than single variable names.

`cond()`.
This useful function can be used to ignore irrelevant observations.

Dividing by zero.
Because dividing by zero yields missing, this also can be used to ignore irrelevant observations.

12 References

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

———. 2002b. Speaking Stata: On getting functions to do the work. *Stata Journal* 2: 411–427.

———. 2007. Speaking Stata: Making it count. *Stata Journal* 7: 117–130.

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

Kantor, D., and N. J. Cox. 2005. Depending on conditions: A tutorial on the cond() function. *Stata Journal* 5: 413–420.

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*.