The Stata Journal (2021)
21, Number 3, pp. 818–837 DOI: 10.1177/1536867X211045582

Speaking Stata: Ordering or ranking groups of observations

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

Abstract. Results for categorical variables may often be clearer if those variables are reordered or reranked, say, according to some measure of absolute or relative frequency or according to summaries of some other variable. Some graphical and tabulation commands have dedicated options serving that end. Otherwise, in practice a new order is often best achieved by creating a new variable holding the desired order using one or another `egen` function. There is usually a need to preserve the information in existing values or value labels and to watch out for ties. There may be a desire to reverse the direction of ranking from the default. I discuss procedures for datasets based on aggregate frequencies and for datasets based on individuals and introduce a new convenience command, `myaxis`, that handles many cases directly.

Keywords: st0654, myaxis, ordering, ranking, graphics, tabulation, categorical data, geometric means, confidence intervals, egen

1 Introduction

Suppose you have a categorical variable that has only an arbitrary order, for example, fruits that may be apples, bananas, or oranges. Such a variable is nominal scale (named) in the much-used classification of Stevens (1946, 1975). As with our fruits, the arbitrary order might be just alphabetical, but on the face of it no other order would be informative either. If there were categories in a natural or evident order, for example, opinions from strongly disagree to strongly agree, then the variable would instead be ordinal (ordered).

However, initial analysis using a nominal variable often suggests that results for such data would be clearer, or at least tidier, if the categories were reordered in graphs or tables, say, according to category frequency or abundance or to the magnitude of some outcome or response. Several Stata commands offer handles for such reordering, including `graph bar`, `graph hbar`, and `graph dot` on one hand and `tabulate` on the other. But it is helpful to have more general methods for ordering, particularly when you want something similar that is not directly available through dedicated options in other commands you are using.

Equivalently, the need may be described as one of ranking, but here the ranking is of groups of observations. That wording may not seem to help much, because ranking in Stata, and statistics generally, is usually phrased in terms of ranking values in each of several observations.

The focus of this column is on methods to produce such ordering or ranking of groups, which in practice often hinges on some convenient functions in `egen`. Contrary to general pedagogic practice, I start with examples that are about as tricky as is common (bad news first) and end with examples where handles for sorting into the desired order already exist (good news follows). I follow a detailed discussion of principles with material on a new convenience command, `myaxis`, that covers many simple cases.

2 Example: Hospitals differing in results

In a much-used textbook, Box, Hunter, and Hunter (1978, 145–149; 2005, 112–116) gave data on patients from five hospitals (A, B, C, D, and E) on the degree of restoration (no improvement, partial functional restoration, complete functional restoration) of certain joints impaired by disease effected by a certain surgical procedure. Hospital E is a referral hospital, but otherwise the identifiers are cryptic (for good reasons, let's assume). Box, Hunter, and Hunter carried out chi-squared analyses, focusing on the difference between Hospital E and the others. Box's (2013) autobiography is entertaining on the background to this book and on much else in his life and career.

The dataset has been widely used as an example, for instance, by Daly et al. (1995, 519–522) and in the help file of `distplot`. On the latter, see Cox (2019) or any later update as revealed by `search distplot, sj`. Although identifiers A to D are not informative, the alphabetical order of their real names would not obviously provide a better order, and those names would make sense only to people familiar with those hospitals.

To see the issue, we first read the data in directly.

```
. input str1 hospital float freq long restore
        hospital       freq       restore
  1.  "A" 13 1
  2.  "B" 5 1
  3.  "C" 8 1
  4.  "D" 21 1
  5.  "E" 43 1
  6.  "A" 18 2
  7.  "B" 10 2
  8.  "C" 36 2
  9.  "D" 56 2
 10.  "E" 29 2
 11.  "A" 16 3
 12.  "B" 16 3
 13.  "C" 35 3
 14.  "D" 51 3
 15.  "E" 10 3
 16.  end
. label values restore restore
. label define restore 1 "none" 2 "partial" 3 "complete"
. label variable restore "How far restored?"
. set scheme sj
. save sandboxhh
file sandboxhh.dta saved
```

For simple graphics, we can use `tabplot`. A detailed discussion of this command was given in Cox (2016a), but do `search tabplot, sj` for the latest update if you want to use it yourself. Using that command is manifestly a personal preference, but graphs using any of several other official or community-contributed commands would make the main point equally clearly.

So that we can focus on the main point of category order, I gather together first various graphics options shared by various figures. This is backward in that sensible options are often (in my case, almost always) worked out more slowly as one tinkers with varying draft graphs.

```
. local options1 separate(restore) subtitle("% by hospital") xtitle(Hospital)
. local options2 bar1(fcolor(gs4) lc(black)) bar2(fcolor(gs8) lc(black))
> bar3(fcolor(gs12) lcolor(black))
```

Figure 1 is a basic two-way bar chart to show the problem. Hospital E stands out, but the default alphabetical ordering is otherwise not helpful.

```
. tabplot restore hospital [w=freq], showval percent(hospital) `options1'
> `options2'
(frequency weights assumed)
```

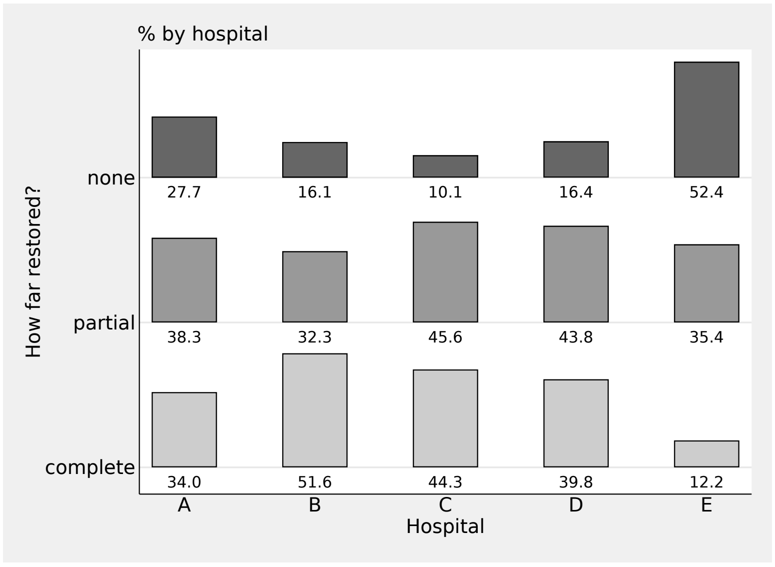

Figure 1. Restoration success in various hospitals. The referral hospital E is clearly different, but the ordering of the other hospitals could be improved.

3 Solution: Use egen's rank() function and change order, fix labels, plot again

To do better, we need to do the following:

1. Produce a new variable that contains the categories in a desired new order. Just occasionally, that might be a string variable if we are lucky or clever about wording. For example, consider string values `"best"`, `"middling"`, and `"worst"`. Those values would plot in the desired order but not because some homunculus inside Stata understands meanings. The right order would be produced for the wrong reason because Stata knows the alphabet. More usually, therefore, this new variable will be a numeric variable with integer values and value labels.
2. Fix value labels if needed, unless what you just did means that is done already, or exceptionally you do not care to see them.
3. Try a new plot. The needed details for a good plot may include better axis titles.

Looking at figure 1, let's suppose that we want to rank on the proportion of category 1 (no restoration, so essentially the bad news). We do this here step by step, showing some detailed techniques that serious Stata users are likely to need again and again.

The proportion of category 1 is the frequency of category 1 divided by the frequency of all categories. If you want to see percents rather than proportions on the graph, that is easy: you can see in figure 1 that `tabplot` has an option for showing them. All we are doing here is reordering the hospitals. Multiplying proportions by 100 would make no difference to the order.

```
. egen num1 = total(freq * (restore==1)), by(hospital)
. egen den = total(freq), by(hosp)
. generate pr1 = num1 / den
```

If the syntax `restore==1` is new to you, here is the story (Cox 2016b). The true-or-false comparison yielded by `==` returns 1 if `restore` is 1, so if the comparison is true, and 0 otherwise if the comparison is false. Applied as a weight, the result of the comparison multiplies the frequencies of category 1 by 1 and other frequencies by 0, and so `total()` just adds the frequency of category 1. Category 1 occurs just once in the dataset for each hospital, so the total of one value is that value. With this code, we are spreading the frequency of category 1 to be a value in all observations for the same hospital.

Backing up, we could have done this instead:

```
. egen den = total(freq), by(hosp)
. generate pr1 = freq / den if restore == 1
```

This code leaves `pr1` undefined for other values of `restore`, but that is not fatal. We will flag this alternative with $(*)$ to ease a later back-reference.

With the first method, the same value of `pr1` will be assigned to all observations in each hospital. (If you are following along by repeating the commands in your copy of Stata, you can see that by using `edit` or `list` to look at the data.) So if we pushed `pr1` through the `rank()` function of `egen` or any other code to produce ranks, the problem of ties would arise. In this example, with three observations for each hospital, the ranks would emerge as three 2s, three 5s, and so on. That can be puzzled out: The three observations with the lowest value would have been ranked 1, 2, and 3 if their values had been slightly different, but the same value should be assigned the same rank. The rule followed is to preserve the sum of the ranks that would have otherwise had been assigned, so the average, $(1+2+3)/3 = 2$, is assigned to each. And so on.

Ranks 2, 5, 8, 11, and 14 are at least equally spaced, so results could be worse, which is much of the point. In any dataset with different numbers of observations in the various categories, the ranks would not emerge even equally spaced. A general solution that avoids such problems is to rank a subset that is based on one observation per group and then spread the resulting ranks to the other observations in each group.

```
. egen rank1 = rank(pr1) if restore == 1
(10 missing values generated)
. bysort hospital (rank1): replace rank1 = rank1[1]
(10 real changes made)
```

We can now mention that this code would work too with the alternative flagged $(*)$ just a short while back.

Going back to first principles has further advantages. By default, ranking on a variable means that the lowest values get the lowest ranks, just as the lowest times win in track events in athletics. If you wanted the reverse order, you can rank on the negated variable, so here you could give `-pr1`, the minus sign indicating reverse order. That would exploit the detail, often overlooked, that `rank()` feeds on an expression, which can be more complicated than a bare variable name. (It does not hurt that `tabplot` has `yreverse` and `xreverse` options anyway, and some graph commands have equivalent options.)

Just to be awkward, but realistic, we may also need ways to break ties if two or more hospitals had the same value on any criterion. The `unique` option can be added to the `rank()` call to ensure breaking of ties.

The ranking has done no more than assign the new ranks 1, 2, 3, 4, and 5 to the hospitals. We should add value labels to preserve the information in the original data. `labmask` is a command dedicated to this purpose (Cox 2008).

```
. labmask rank1, values(hospital)
```

`labmask` here takes the values of `hospital` and uses them to define value labels for `rank1`. The slightly whimsical name echoes a notion that value labels are like a mask that numeric values wear. The same syntax would have been used if `hospital` had been a numeric variable without value labels, while the extra option `decode` is needed if existing value labels are to be preserved.

Now we can try a different plot (figure 2):

```
. tabplot restore rank1 [w=freq], showval percent(hospital) `options1'
> `options2'
(frequency weights assumed)
```

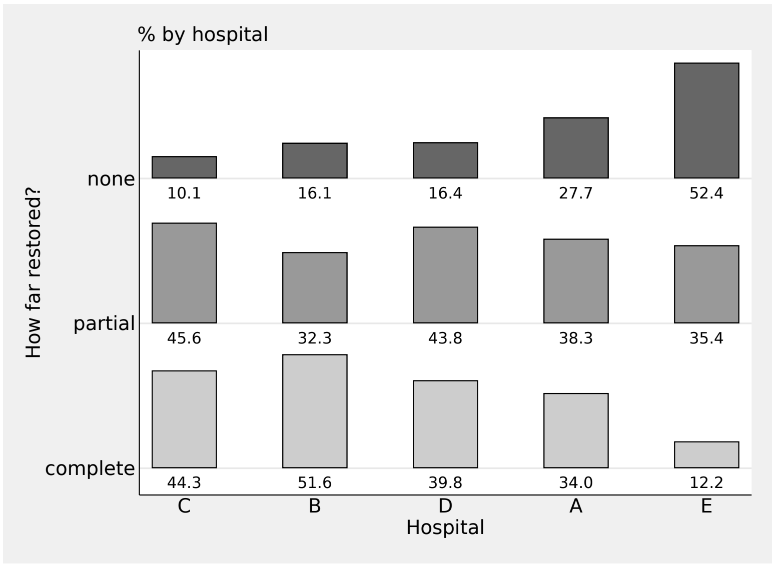

Figure 2. Restoration success in various hospitals. The hospitals are ordered by proportion of no restoration. Recall that E is a referral hospital.

Ranking on any other criterion just entails the same kind of sequence. Ranking on the proportions of partial or complete restoration implies an easy change from `restore == 1` to `restore == 2` or `restore == 3` in the code above.

Here is something a little more challenging: the use of a weighted mean over scores 1 to 3. At this point, I do not want to start a secondary debate on how logical or admissible it is to calculate means for ordinal grades. But I will season the dish with some references to the long-running and lively discussion of such matters within statistical science: Duncan (1984); Velleman and Wilkinson (1993); and Hand (1996, 2004).

Note that the sum of frequencies has already been calculated as the variable `den`, so that will serve as denominator. We need a numerator too.

```
. egen num = total(restore * freq), by(hosp)
. generate wtmean = num/den
```

There is a good way to select just one observation from several for each hospital. The `tag()` function of `egen` assigns 1 to one observation in any group and 0 to the others in the same group. Because they are all the same, it does not matter which one is chosen, but the code uses the first observation seen in each group. A public function goes back to 1999, but the basic idea was even then quite standard (Cox 1999). That aside as a variant trick, we can follow a similar path to a different plot (figure 3).

```
. egen tag = tag(hospital)
. egen rank = rank(wtmean) if tag
(10 missing values generated)
. bysort hospital (rank): replace rank = rank[1]
(10 real changes made)
. labmask rank, values(hospital)
. tabplot restore rank [w=freq], showval percent(hospital) `options1'
> `options2'
(frequency weights assumed)
```

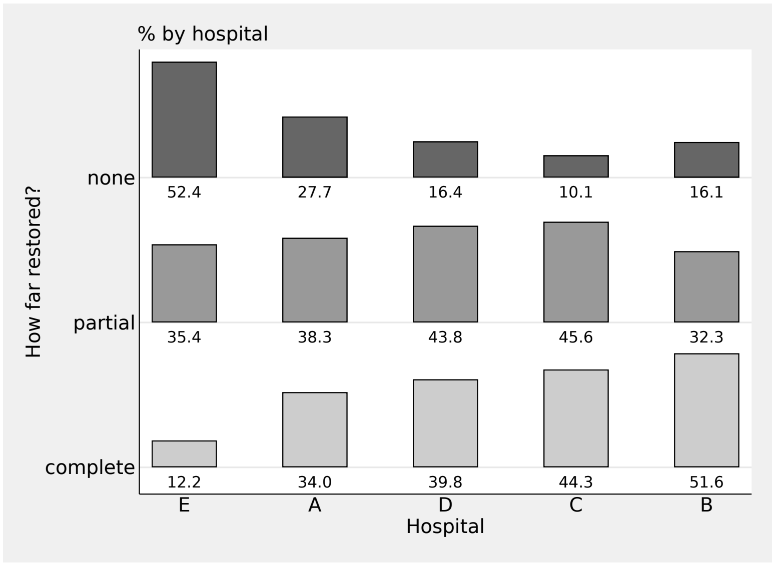

Figure 3. Restoration success in various hospitals. The hospitals are ordered by weighted mean score. Recall that E is a referral hospital.

4 Solution: Use egen's group() function instead

Let's back up and show a different solution, but one in similar spirit, this time using the `group()` function of `egen`. If you typed in the code given earlier, you can skip data input and go back to the dataset `saved` from the original data.

```
. use sandboxhh, clear
```

Then, different code follows to get a plot with proportion with no restoration as the ordering criterion, with commentary only on what is different.

```
. egen num1 = total(freq * (restore==1)), by(hospital)
. egen den = total(freq), by(hosp)
. generate pr1 = num1 / den
. egen rank1 = group(pr1 hospital)
. labmask rank1, values(hospital)
. tabplot restore rank1 [w=freq], showval percent(hospital) `options1' `options2'
```

The resulting graph is just figure 1 again.

So we exploit the fact that `group()` gives a variant of ranking, but a variant that we often want: All observations with the same lowest value are assigned 1, all observations with the next lowest value are assigned 2, and so on. The reason for specifying `hospital` too as an argument—note the two variable names `pr1` and `hospital`—is to break ties if there are any; it does no harm otherwise.

If you wanted reverse ranking, you might need to negate one or more of the variables before they are fed to `group()`. There is no handle for that otherwise. `group()` does not allow anything but variable names in its main argument, and there is no such option. Again, reverse ranking may be allowed anyway by your graphics or tabulation command. Yet again, at least in simple cases, reversing the ranks is a matter of subtraction. If your ranks run 1 to 5 and you want to reverse the order, subtract the ranks from 6: $6 - 1 = 5$, and so on.

5 What about individual data?

The dataset used so far was presented in concise form with a frequency variable. You can produce such a dataset yourself by applying `contract` to a dataset on individuals (here patients). Conversely, you can use `expand` to go the other way to get a dataset in which each observation is a patient from a dataset with a frequency variable. If you imagine, or even execute,

```
. expand freq
(352 observations created)
```

then the code to get figure 1 yet again is now even simpler:

```
. egen pr1 = mean(restore==1), by(hospital)
. egen rank1 = group(pr1 hospital)
. labmask rank1, values(hospital)
. tabplot restore rank1, showval percent(hospital) `options1' `options2'
```

The simple but fundamental trick here is that the proportion of any category is just the mean of the corresponding indicator variable. See Cox (2016b) and Cox and Schechter (2019) if that sounds novel or puzzling.

Naturally, we do not need this code for our dataset. It is given as an example because datasets often arrive in similar form, with one observation for each individual.

6 Example: Geometric means

I continue with another example in which the small labor of sorting into congenial order is delegated to a graphical command. The example is included partly as a reminder to myself to write about geometric means in some future column. The geometric mean as the exponential of the mean logarithm remains valuable as a way of nodding to logarithmic thinking while producing a summary in the original units of measurement. Its use for summarizing data, in a suitably broad sense, seems to go back at least to Galileo Galilei, better known as an astronomer and a physicist (Reston 1994, 218):

> In 1627 Galileo ‘was presented with an amiable dispute between a Florentine gentleman and a parish priest over the proper method to price a horse ... one bidder—undoubtedly the priest—had offered ten crowns and the other one thousand. In arriving at the proper value, the equestrians asked Galileo to be their arbiter. Was it better to employ an arithmetic or a geometric proportion in arriving at a fair price between divergent estimates? A geometric proportion was Galileo’s answer. The real value of the horse was one hundred.’

Geometric means are possible only when all values are positive. They are useful mostly when data arrive positively skewed. Their canonical application is to lognormal distributions, for which geometric means and medians coincide, but it has wider utility. (Note: If all values are negative, the signs can be treated as conventional and ignored.) Community-contributed code for putting geometric means into variables is easy enough to find, although by the time you have found and installed it, you could have done it slowly with two or at most three commands. The solution here exploits the detail, often overlooked, that the `egen` function `mean()`, just like `rank()`, feeds on an expression, which can be more complicated than a bare variable name.

The next application is to hourly wages from the U.S. National Longitudinal Survey of Young Women and Mature Women in 1988. The graph orders industries by their geometric means of hourly wage (figure 4).

```
. sysuse nlsw88, clear
(NLSW, 1988 extract)
. egen gmean = mean(ln(wage)), by(industry)
. replace gmean = exp(gmean)
(2,246 real changes made)
. egen tag = tag(industry)
. graph dot (asis) gmean if tag, over(industry, sort(1) descending)
> linetype(line) lines(lc(gs12) lw(vthin))
> ytitle(Geometric mean hourly wage (USD)) ysc(alt)
```

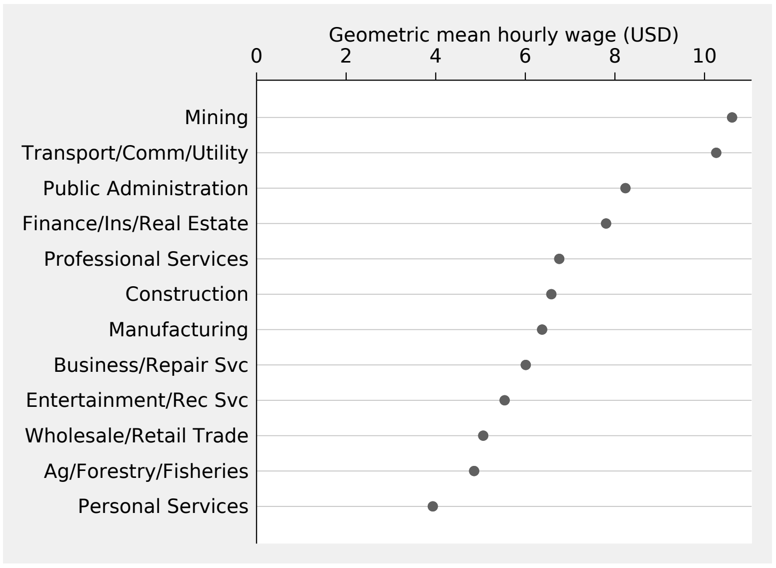

Figure 4. Geometric mean hourly wage for different industries

I find the default dotted grid in `graph dot` a little distracting—and it is often degraded on export to other software. So my usual choice is, as just given, more like

```
    linetype(line) lines(lc(gs12) lw(vthin))
```

Using a light-gray color to downplay less important graph elements is a standard yet underplayed strategy. See, for example, Schwabish (2021) for general emphasis and Cox (2009) for particular Stata applications. Another personal choice is `ysc(alt)` for many graphs with something of a table flavor. See Cox (2012a) for more on what goes (best) on what axis.

Ordering industries by geometric mean wage is suggested as one helpful choice, but other choices are entirely possible. In this dataset, the order of the variable `industry` is evidently, in economic terms, a sequence from primary through secondary to tertiary or even quaternary sectors, so that might be helpful too.

7 Missing values

The small question of wanting to reorder categorical variables leads to several smaller questions, and this column has not covered them all.

Are there missing values for categorical variables? You need to decide whether to include or to exclude them, and you may find that the default behavior of a command jumps the other way. This does not seem much of an issue in practice, so I will leave the matter there.

8 Tables as well as graphics

What about tables? Some table commands also have handles to vary sort order: `tabulate` is a good example. Mostly, you need to create the ordered variable you want first, as in this column. That said, do not underestimate the scope for do-it-yourself tables on the fly, using, say, `list` or `tabdisp` (Cox 2003, 2012b).

Given the geometric means just calculated, here is a simple table constructed on the fly.

```
. char gmean[varname] "geometric mean wage (USD/hr)"
. gsort -tag -gmean
. format gmean %3.2f
. list industry gmean if tag, noobs sep(0) abbrev(28) subvarname
```

```
  +----------------------------------------------------------+
  |               industry   geometric mean wage (USD/hr)   |
  |----------------------------------------------------------|
  |                 Mining                          10.62   |
  |  Transport/Comm/Utility                         10.26   |
  |   Public Administration                          8.24   |
  | Finance/Ins/Real Estate                          7.80   |
  |   Professional Services                          6.76   |
  |            Construction                          6.58   |
  |           Manufacturing                          6.37   |
  |     Business/Repair Svc                          6.01   |
  |   Entertainment/Rec Svc                          5.54   |
  |  Wholesale/Retail Trade                          5.06   |
  |   Ag/Forestry/Fisheries                          4.86   |
  |       Personal Services                          3.93   |
  +----------------------------------------------------------+
```

So far, so good, but by many standards we are still at the trivial shallow end of the tabulation pool. Let's try a two-way table and spell out a device that can help in tidying up a messy table. We need an example large enough to be convincing as messy and small enough not to take up too much space. Keeping with the industry variable, we note a variable `age` with a range from 34 to 46 years. Let's bin that a little arbitrarily. I like self-describing bins:

```
. generate age_cat = cond(age >= 42, 42, cond(age >= 38, 38, 34))
. label define age_cat 34 "34-" 38 "38-" 42 "42-"
. label values age_cat age_cat
. label variable age_cat "age intervals (years)"
```

So our binning variable has distinct values 34, 38, and 42, corresponding to the lower inclusive limits of bins 34 up, 38 up, and 42 up. The value labels are explicit about what happens at the limits. Pragmatically, a few women aged 46 have been included in the top bin. For more on binning, including the need to be explicit about binning rules, see Cox (2018).

We know how to get geometric means into a variable. The issue is going to be getting them into a good order. If `gmean` is still lurking from a previous burst of calculation, we need to get rid of it, as done here, or else use a different variable name.

```
. drop gmean
. egen gmean = mean(ln(wage)), by(industry age_cat)
. replace gmean = exp(gmean)
(2,246 real changes made)
. tabdisp industry age_cat, c(gmean) format(%3.2f)

─────────────────────────────────────────────
                         │    age intervals
                         │       (years)
                industry │   34-     38-    42-
─────────────────────────┼───────────────────
                         │
  Ag/Forestry/Fisheries  │  3.71   7.99   3.25
                 Mining  │ 12.50          6.51
           Construction  │  6.70   6.41   6.66
          Manufacturing  │  6.41   6.51   6.17
 Transport/Comm/Utility  │  9.76   9.94  11.47
 Wholesale/Retail Trade  │  5.24   5.16   4.71
Finance/Ins/Real Estate  │  7.59   8.27   7.45
    Business/Repair Svc  │  7.31   4.88   6.01
      Personal Services  │  4.02   4.08   3.61
  Entertainment/Rec Svc  │  6.91   4.49   5.07
  Professional Services  │  6.56   7.00   6.63
  Public Administration  │  8.07   8.75   7.49
                      .  │  4.24   3.22   7.45
─────────────────────────────────────────────
```

Bringing up the rear are a bunch of people with industry unknown, so we will leave them out as sadly uninformative. We can tidy up the table in various ways, but one simple choice is to order on geometric mean wage for age group 34 to 37. The trick is to ensure that observations for other age categories are populated too so that all observations of interest get classified correctly. Here is one way to do it, explained in detail in an earlier column (Cox 2011):

```
. egen GMEAN = mean(cond(age_cat == 34, ln(wage), .)) if industry < .,
> by(industry)
(14 missing values generated)
```

We do not need to exponentiate that; the values of `GMEAN` are necessarily in the desired order already. Now the commands are already familiar or will shortly make sense:

```
. egen order = group(GMEAN)
(14 missing values generated)
. label variable order "geometric mean wage (USD/hr)"
. labmask order, values(industry) decode
. tabdisp order age_cat if order < ., c(gmean) format(%3.2f)

─────────────────────────┬────────────────────
                         │   age intervals
geometric mean wage      │      (years)
(USD/hr)                 │    34-    38-    42-
─────────────────────────┼────────────────────
  Ag/Forestry/Fisheries  │   3.71   7.99   3.25
      Personal Services  │   4.02   4.08   3.61
 Wholesale/Retail Trade  │   5.24   5.16   4.71
          Manufacturing  │   6.41   6.51   6.17
  Professional Services  │   6.56   7.00   6.63
           Construction  │   6.70   6.41   6.66
  Entertainment/Rec Svc  │   6.91   4.49   5.07
    Business/Repair Svc  │   7.31   4.88   6.01
Finance/Ins/Real Estate  │   7.59   8.27   7.45
  Public Administration  │   8.07   8.75   7.49
 Transport/Comm/Utility  │   9.76   9.94  11.47
                 Mining  │  12.50          6.51
─────────────────────────┴────────────────────
```

We have been careless about the possibility of ties. And we could have a small discussion about whether a graph would work as well or better.

9 Example: Confidence intervals

Another kind of challenge is solved by creating a dataset consisting entirely of results, so that then a `sort` on one or more variables suffices to get the order you want. Using `statsby` to get a dataset for a confidence interval plot is a standard example (Cox 2010). We go back to those geometric means for the hourly wage. This code starts with reading the dataset in once more, partly to give a self-contained example that is easier to adapt for your own use if you have a similar need.

```
. sysuse nlsw88, clear
(NLSW, 1988 extract)
. statsby N=r(N_pos) gmean=r(mean_g) ub=r(ub_g) lb=r(lb_g), by(industry):
> ameans wage
  (output omitted)
. sort gmean
. generate order = _n
. labmask order, values(industry) decode
```

The next two commands are presented disingenuously. Some experience and experiment underlined that numbers in mining are very small and the associated confidence interval correspondingly wide. Then it seemed that showing sample sizes would be a good idea, but some trial and error is needed to work out where and how they should be shown (figure 5).

```
. generate where = 51
. scatter order gmean, ms(Dh) ||
> scatter order where, ms(none) mla(N) mlabsize(medsmall) mlabpos(9)
> xmla(49 "{it:n}", labsize(medsmall) tlc(none)) ||
> rcap ub lb order, horizontal ytitle("") yla(1/`=_N', noticks ang(h) valuelabel)
> xsc(alt) xla(0(10)40, grid) t1title(95% CIs for geometric mean wage in USD/hr)
> legend(off)
```

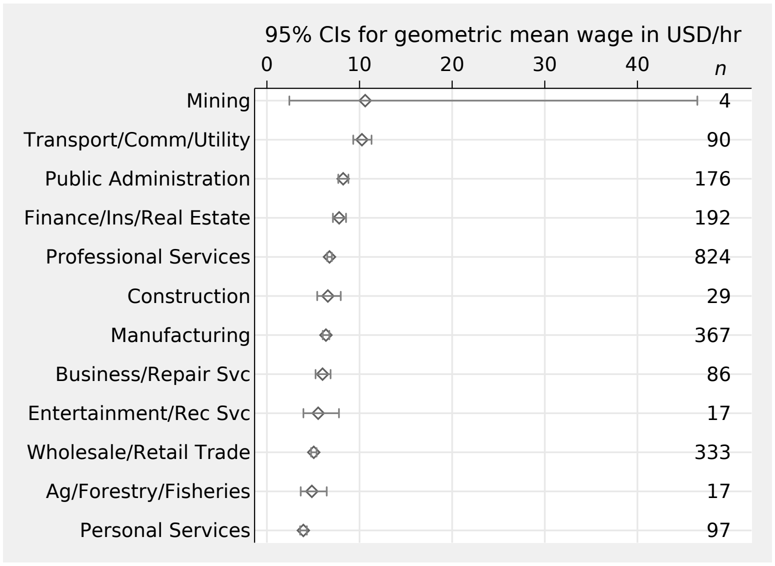

Figure 5. Confidence intervals for geometric mean wage by industry. Despite the intrinsic interest of this graph, its importance here is how it was produced by using `statsby` to collate results of `ameans`. Sorting categories by their geometric means was then straightforward.

10 A convenience command: myaxis

A tension or tradeoff for all Stata users, whether beginners or more experienced, is how far to break down a problem into a series of simple code steps and how far to seek or (depending on your experience) even to write a new command that ideally will solve the problem in one call. Many, perhaps even most, users tend to resolve this through do-files so that they bundle a series of commands into one script. As they extend or correct their code, they can cut down on the amount of retyping because all previous work has been saved in a file. A benefit but also a cost of do-files is that they can be utterly ad hoc and geared to particular datasets.

Users who have begun to program still face a dilemma. Programming can be premature: You can write programs that do not deserve existence, but usually they fade into oblivion without pain. I have written commands that I have regretted. Some such commands I later thought trivial because just a few standard lines could replace them without any need for me to rediscover the syntax, let alone remember what the command was called. Others were insufficiently focused or too elaborate to seem convenient or congenial on later use.

The structure of this column loosely matches my work on the topic. I first decided to write up the little tricks so far discussed that I was using again and again and recommending to others. Then later the small project crystallized in a decision to bundle the main ideas into a command called `myaxis`.

Let's first exemplify how `myaxis` works for some of the problems discussed so far.

Here is how to re-create figure 2, starting from calculation of `pr1`.

```
. myaxis RANK1=hospital, sort(mean pr1) subset(restore==1)
. tabplot restore RANK1 [w=freq], showval percent(hospital) `options1' `options2'
```

`RANK1` is the new variable created using calculations for the subset of observations `restore == 1`. `pr1` already exists as a constant for each hospital, but using the `egen` function `mean` is a simple and harmless way to pick up that constant because the mean of one (repeated) value is just that value. Otherwise explained, the first argument of `sort()` is always the name of an `egen` function. That could be a community-contributed `egen` function so long as its code is visible along your `adopath`.

Variable and value labels are handled automatically, so there is no need to invoke `labmask` or otherwise to define labels.

Here is how to re-create figure 3, starting from calculation of `wtmean`.

```
. myaxis RANK=hospital, sort(mean wtmean)
. tabplot restore RANK [w=freq], showval percent(hospital) `options1' `options2'
```

Perhaps more strikingly, the step backward of trying to think more generally showed that other related problems could be treated easily within a larger framework. With the auto data read in, these simple exercises can be executed by curious readers. The results are suppressed to save space.

```
. sysuse auto, clear
```

First up is a simple tabulation where we decide we want to sort categories by frequency. As it happens, `tabulate` already has a `sort` option, but knowing how to do it could be useful for many other commands. One obvious and one more subtle twist to the syntax: `descending` reverses the conventional sort order because people usually want to see the most frequent categories first. Note that `sort()` has one argument only, although a second argument now turns out to be tacit. `count` means in context `count rep78`, meaning use the `egen` function `count()` to count nonmissing values of repair record `rep78`.

```
. myaxis wanted=rep78, sort(count) descending
. tabulate wanted
. tabulate wanted, nolabel
```

The flavor of the next example is similar. We want categories to be sorted by the mean of miles per gallon `mpg`, so we get `myaxis` to work that out and record the order of categories (again, high `mpg` is good; low litres per km or per 100 km is good if such units are more familiar to you).

```
. myaxis wanted2=rep78, sort(mean mpg) descending
. format mpg %2.1f
. tabulate wanted2, summarize(mpg)
```

As a final twist, we look ahead to a two-way table. We want to bring in whether cars are domestic (manufactured inside the United States) or foreign (not so) and decide to sort on the foreign performance.

```
. myaxis wanted3=rep78, sort(mean mpg) subset(foreign==1) descending
. tabulate wanted3 foreign, summarize(mpg) nost nofreq
```

Although `myaxis` can be used for many problems, no command ever tackles all the problems in its territory. For example, `myaxis` makes no attempt to support weights directly if only because weights do not match the framework of `egen`. Problems involving weights require a focused decision on what makes sense and how to calculate it. Our very first example in Section 2 was really about frequency weights, so weights need not be difficult.

11 Syntax of myaxis

`myaxis` reorders a categorical variable by a specified sort criterion.

`myaxis` *newvar*=*varname* [*if*] [*in*], `sort(`*criterion*`)`
[`subset(`*true_or_false_condition*`)` `missing` `descending` `varlabel(`*string*`)`
`valuelabelname(`*string*`)`]

11.1 Description

`myaxis` maps an existing "categorical" variable, meaning usually a numeric variable with integer codes and value labels, or equivalently a string variable, to a new variable with integer values 1 up and with value labels, sorted according to a specified criterion.

11.2 Remarks

The command name `myaxis` is to be parsed "my axis". The second element "axis" arises from a leading application of the command. You have a categorical variable that would define an axis of a graph or one dimension of a table (the rows or the columns, say), but the existing order of categories is not ideal. Some graph and table commands offer sorting on the fly, but this command may help wherever other commands do not offer that.

The first element "my" is at best harmless whimsy, but it arises because mentions of a command named just `axis` would be harder to spot among other uses of the term.

The problem is split by `myaxis` into these parts:

1. Calculation of a numeric variable on which to sort categories. `myaxis` treats this as an application of `egen`. Note: If a variable already exists that defines the sort order and is constant within categories, then asking for (say) its minimum, mean, or maximum within each category will suffice.
2. Deciding whether you want ascending order (the default) or descending order (highest value goes first). Descending order requires negation of the variable from the first step.
3. Mapping your categorical variable to integers 1 up. The `group()` function of `egen` does the work here, but `myaxis` is careful to split ties according to the original variable. (For example: Suppose nominal categories A, B, C, D, and E have frequencies 7, 7, 42, 3, and 1 and you want them sorted by frequency. You do not want A and B lumped together, because they have the same frequency.)
4. Fixing a variable label. `myaxis` uses a new variable label if supplied; otherwise, it uses the original variable label; and, if that does not exist, it uses the original variable name.
5. Fixing value labels. This is even more important than the previous point for helpful display in a graph or table. `myaxis` uses the original value labels if defined and otherwise the original string or numeric values.

11.3 Options

`sort(`*criterion*`)` specifies the criterion for sorting. `sort()` is required. The criterion should always include the name of an `egen` function. That function may be community contributed so long as the code is visible along your `adopath`. The criterion may also include the name of an existing variable, and that is essential whenever the sort criterion is not based on *varname*.

`subset(`*true_or_false_condition*`)` specifies a subset of the data on which the sort criterion should be calculated. Concretely, imagine two variables that define y and x axes of a graph or rows and columns of a table. You might want rows to be sorted by values calculated for a particular column or columns to be sorted by values calculated for a particular row.

`missing` specifies that missing values of *varname* be included. The default is to ignore them.

`descending` specifies sorting with highest value first. The default sort order is ascending with lowest value first.

`varlabel(`*string*`)` specifies a variable label for the new variable. Otherwise, see point 4 in section 11.2 *Remarks*.

`valuelabelname(`*string*`)` specifies a new value label name for the value labels of the new variable. This will be needed if there is already a set of value labels called *newvar*.

12 Conclusion

Results for nominal variables may often be clearer if those variables are reordered or reranked, say, according to some measure of absolute or relative frequency or according to summaries of some other variable. Some graphical and tabulation commands have dedicated options serving that end. Alternatively, a dataset consisting of results can be `sort`ed directly. We have seen examples using `graph dot` and `graph twoway`, while `graph bar` or `graph hbar` could have been used to provide other illustrations.

Otherwise, in practice a new order is often best achieved by creating a new variable holding the desired order, using along the way one or more `egen` functions: `rank()`, `group()`, `tag()`, `total()`, `count()`, and `mean()` all appeared in examples, and several more functions are available. There is usually a need to preserve the information in values or value labels and to watch out for ties. There may be a desire to reverse ranking from the default. Procedures for datasets based on aggregate frequencies and for datasets based on individuals may also differ in detail.

All of this can be done step by step, and understanding the small details will serve you well in many other problems. Alternatively, the new command `myaxis` offers a convenience command for many such problems and is introduced in this column.

13 Programs and supplemental materials

To install a snapshot of the corresponding software files as they existed at the time of publication of this article, type

```
. net sj 21-3
. net install st0654      (to install program files, if available)
. net get st0654          (to install ancillary files, if available)
```

14 References

Box, G. E. P. 2013. *An Accidental Statistician: The Life and Memories of George E. P. Box*. Hoboken, NJ: Wiley.

Box, G. E. P., J. S. Hunter, and W. G. Hunter. 2005. *Statistics for Experimenters: Design, Innovation, and Discovery*. 2nd ed. Hoboken, NJ: Wiley.

Box, G. E. P., W. G. Hunter, and J. S. Hunter. 1978. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*. New York: Wiley.

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.

———. 2003. Speaking Stata: Problems with tables, Part I. *Stata Journal* 3: 309–324. https://doi.org/10.1177/1536867X0300300308.

———. 2008. Speaking Stata: Between tables and graphs. *Stata Journal* 8: 269–289. https://doi.org/10.1177/1536867X0800800208.

———. 2009. Stata tip 78: Going gray gracefully: Highlighting subsets and downplaying substrates. *Stata Journal* 9: 499–503. https://doi.org/10.1177/1536867X0900900311.

———. 2010. Speaking Stata: The statsby strategy. *Stata Journal* 10: 143–151. https://doi.org/10.1177/1536867X1001000112.

———. 2011. Speaking Stata: Compared with .... *Stata Journal* 11: 305–314. https://doi.org/10.1177/1536867X1101100210.

———. 2012a. Speaking Stata: Axis practice, or what goes where on a graph. *Stata Journal* 12: 549–561. https://doi.org/10.1177/1536867X1201200314.

———. 2012b. Speaking Stata: Output to order. *Stata Journal* 12: 147–158. https://doi.org/10.1177/1536867X1201200109.

———. 2016a. Speaking Stata: Multiple bar charts in table form. *Stata Journal* 16: 491–510. https://doi.org/10.1177/1536867X1601600214.

———. 2016b. Speaking Stata: Truth, falsity, indication, and negation. *Stata Journal* 16: 229–236. https://doi.org/10.1177/1536867X1601600117.

———. 2018. Speaking Stata: From rounding to binning. *Stata Journal* 18: 741–754. https://doi.org/10.1177/1536867X1801800311.

———. 2019. Software Updates: gr41_5: Distribution function plots. *Stata Journal* 19: 260. https://doi.org/10.1177/1536867X19833285.

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.

Daly, F., D. J. Hand, M. C. Jones, A. D. Lunn, and K. J. McConway. 1995. *Elements of Statistics*. Wokingham: Addison–Wesley.

Duncan, O. D. 1984. *Notes on Social Measurement: Historical and Critical*. New York: Russell Sage Foundation.

Hand, D. J. 1996. Statistics and the theory of measurement. *Journal of the Royal Statistical Society, Series A* 159: 445–492. https://doi.org/10.2307/2983326.

———. 2004. *Measurement Theory and Practice: The World through Quantification*. London: Arnold.

Reston, J., Jr. 1994. *Galileo: A Life*. New York: HarperCollins.

Schwabish, J. 2021. *Better Data Visualizations: A Guide for Scholars, Researchers, and Wonks*. New York: Columbia University Press.

Stevens, S. S. 1946. On the theory of scales of measurement. *Science* 103: 677–680. https://doi.org/10.1126/science.103.2684.677.

———. 1975. *Psychophysics: Introduction to Its Perceptual, Neural, and Social Prospects*. New York: Wiley.

Velleman, P. F., and L. Wilkinson. 1993. Nominal, ordinal, interval, and ratio typologies are misleading. *American Statistician* 47: 65–72. See also discussions and replies 47: 314–316 and 48: 61–62. https://doi.org/10.1080/00031305.1993.10475938.

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).