The Stata Journal (2003)
3, Number 4, pp. 420–439

Speaking Stata: Problems with tables, Part II

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

Abstract. Three user-written commands are reviewed as illustrations of different approaches to tabulation problems, each one step beyond what is possible to do directly through official Stata. `tabcount` is a wrapper for `tabdisp` written to produce tables that show how often specified values occur or specified conditions are satisfied so that, in particular, tables may include explicit zeros whenever desired. `makematrix` is designed for situations in which a table of results may be compiled by populating a matrix. `matrix list` or `list` may then be used to display the table. `groups` shows frequencies of combinations of values using `list`. Users should find these commands to be helpful additions to the toolkit. Programmers may be interested in examples of the wrapper approach, calculating the values to be tabulated before passing them to a workhorse display command. This is the second of two papers on this topic.

Keywords: pr0011, tables, matrices, tabcount, makematrix, groups, tabdisp, list

1 Introduction

Tables are pervasive and, indeed, fundamental. Tables of data, tables of frequencies or summaries of data, and tables of model results are basic to both elementary and advanced statistical analysis. In the previous column (Cox 2003), we looked at how far official Stata commands provide direct solutions to tabulation problems and at how some preparation of variables to be tabulated allows fairly painless indirect solutions to such problems. To recap, the main general-purpose tabulation commands are `tabulate` ([R] **tabulate** and [R] **tabsum**), `table` ([R] **table**), and `tabstat` ([R] **tabstat**), while material may be prepared for tabulation as a set of variables, after which the table itself can be presented with `tabdisp` ([P] **tabdisp**) or `list` ([R] **list**).

In this paper, we turn to examples of user-written commands that can help resolve problems with tables. The three examples explained show ways in which problems that would otherwise be awkward to solve may be tackled with single commands. Users may find them helpful additions to the toolkit, while programmers may wish to study the code to see examples of wrapper commands in which we calculate the values to be tabulated before passing them to a workhorse display command. Software for the three commands, `tabcount`, `makematrix`, and `groups`, may be installed in a net-aware Stata by using `ssc` ([R] **ssc**).

Naturally, there are many approaches to problems of tabulation and to problems with a tabulation element. For another self-contained approach to managing what are in effect tables of results treated as datasets, see the programs discussed by Newson (2003).

2 Tabulating frequencies of specified values or conditions

Let us start with a simple truism, expressed whimsically: Stata is reluctant to display values not present in the dataset. Indeed, metaphysics is not at all Stata's strong suit. Suppose that a variable could take on integer values 1 through 5, but in fact 5 was not observed in the dataset at hand. Then, a tabulation using `tabulate` or `table` of that variable will show the frequencies of values 1, 2, 3, and 4. Simply, Stata has no way of knowing that the value might have been 5 (or indeed 6 or −1 or any other value consistent with the storage type assigned). Nevertheless, users often ask for tables that show explicitly that a value has zero frequency, either by a blank entry or by a literal zero. Showing zeros explicitly may be thought of as part of showing the structure of the data.

This request may arise in a variety of situations, univariate, bivariate, and multivariate. Often, the desire is for a set of tables to be presented in a standardized way. Here is a simple example. In the `auto` dataset, look at a tabulation of `rep78` by groups of `foreign`:

```
. bysort foreign: tab rep78

-> foreign = Domestic
     Repair
Record 1978 |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |          2        4.17        4.17
          2 |          8       16.67       20.83
          3 |         27       56.25       77.08
          4 |          9       18.75       95.83
          5 |          2        4.17      100.00
------------+-----------------------------------
      Total |         48      100.00

-> foreign = Foreign
     Repair
Record 1978 |      Freq.     Percent        Cum.
------------+-----------------------------------
          3 |          3       14.29       14.29
          4 |          9       42.86       57.14
          5 |          9       42.86      100.00
------------+-----------------------------------
      Total |         21      100.00
```

As values of `rep78` of 1 and 2 do not occur for foreign cars, no such rows are given. If you wanted rows with zero frequencies explicit, how could it be done? `tabulate` (and indeed also `table`) will put zeros in cells so long as there are nonzeros in the same row, column, etc. (There is a cosmetic difference in that `tabulate` shows a literal `0`, while `table` shows a blank.) As the example shows, however, rows and columns that would be all zeros are omitted completely.

`tabcount` is designed for this need. You can spell out the values 1/5 as what you want shown and choose between blanks (the default) and literal zeros:

```
. bysort foreign: tabcount rep78, v(1/5)

-> foreign = Domestic

Repair
Record
1978          Freq.

       1          2
       2          8
       3         27
       4          9
       5          2

-> foreign = Foreign

Repair
Record
1978          Freq.

       1
       2
       3          3
       4          9
       5          9
```

As you might guess, the option name `v` is meant to suggest values. As another example, let us imagine a dataset in which the number of children per mother is a variable, so that even in a very large dataset the tail of the distribution may be rather straggly. With `tabulate` or `table`, our output might end something like this:

```
 10 |          6        8.11       89.19
 11 |          4        5.41       94.59
 13 |          2        2.70       97.30
 14 |          1        1.35       98.65
 16 |          1        1.35      100.00
```

If we want the complete set of rows, `tabcount` with the option `v(1/16)` will suffice:

```
      10 |         6
      11 |         4
      12 |
      13 |         2
      14 |         1
      15 |
      16 |         1
```

You can also specify sets of conditions (`c()`): a condition is an inequality or a value (and if a value, it is interpreted as an equality):

```
. bysort foreign: tabcount rep78, c(<=2 3 4 5) zero

-> foreign = Domestic

  Repair
  Record
  1978     |    Freq.
-----------+----------
       <=2 |       10
         3 |       27
         4 |        9
         5 |        2

-> foreign = Foreign

  Repair
  Record
  1978     |    Freq.
-----------+----------
       <=2 |        0
         3 |        3
         4 |        9
         5 |        9
```

That is, `c(<=2 3 4 5)` defines categories ≤ 2, (equal to) 3, (equal to) 4, (equal to) 5. Incidentally, there is no rule that conditions must be mutually exclusive—or indeed that they must be collectively exhaustive. This makes it easy, for example, to tabulate cumulative frequencies, whether defined by $<, \leq, >$, or $\geq$.

However, there is no syntax for specifying intervals with two limits, say, of the form $a \leq x < b$. You must create the coarsened variable(s) yourself beforehand.

With two or more variables, you must specify either a `v` option or a `c` option for each variable. Those options are tagged with 1, 2, etc., according to which variable is being referred to:

```
. tabcount foreign rep78, v1(0 1) c2(<=2 3 4 5)

           |   Repair Record 1978
  Car type |  <=2     3     4     5
-----------+------------------------
  Domestic |   10    27     9     2
   Foreign |          3     9     9
```

That is, `v1` or `c1` refers to the first variable named, `v2` or `c2` refers to the second variable named, and so forth. Seven variables is the limit, a limit that is imposed by `tabdisp`, which does the tabulation itself.

`tabcount` on the other hand is limited: it will not show you percents, cumulative percents, cumulative frequencies, or anything else apart from the frequencies. (As analytic weights are allowed, it is a little more general than just a counting program.)

These limitations are less than may appear at first sight because you may `replace` the dataset in memory with a reduced dataset:

```
. tabcount foreign rep78, v1(0 1) v2(1/5) replace
```

Car type	Repair Record 1978 1	2	3	4	5
Domestic	2	8	27	9	2
Foreign			3	9	9

```
. list
```

	_freq	foreign	rep78
1.	2	Domestic	1
2.	0	Foreign	1
3.	8	Domestic	2
4.	0	Foreign	2
5.	27	Domestic	3
6.	3	Foreign	3
7.	9	Domestic	4
8.	9	Foreign	4
9.	2	Domestic	5
10.	9	Foreign	5

As you may know, the existing official command `contract` ([R] **contract**) could do that for you in this case, but `tabcount` is more general than that. Unlike `contract`, it supports analytic weights; counting of any specified values, which might not exist in the data; and also counting of how often conditions as specified by inequalities or equalities are satisfied.

Given the `replace` option, the new reduced dataset can then be the basis for all sorts of customized tables. Let us suppose that we want cumulative frequencies and cumulative percents in our table:

```
. bysort foreign (rep78): gen cufreq = sum(_freq)
. by foreign: gen cupc = 100 * cufreq / cufreq[_N]
. tabdisp foreign rep78, c(cufreq cupc)
```

Car type	Repair Record 1978 1	2	3	4	5
Domestic	2	10	37	46	48
	4.166667	20.83333	77.08334	95.83334	100
Foreign	0	0	3	12	21
	0	0	14.28571	57.14286	100

You can control several details of presentation:

```
. tabdisp foreign rep78, c(cufreq cupc) format(%2.1f)
```

	Repair Record 1978				
Car type	1	2	3	4	5
Domestic	2.0	10.0	37.0	46.0	48.0
	4.2	20.8	77.1	95.8	100.0
Foreign	0.0	0.0	3.0	12.0	21.0
	0.0	0.0	14.3	57.1	100.0

As before, you could do something like this already with `table`, but once again, `tabcount` is in various ways more general.

There is more explanation of some other features, including saving one- and two-way tables of frequencies to matrices, and more examples, in the help file.

As already mentioned, the `tabcount` program is based on the official command `tabdisp`, to which there are two sides. First, `tabcount` calculates the frequencies before handing them to `tabdisp` for display. Second, `tabcount, replace` provides a starting point for subsequent customized tabulations, again typically with `tabdisp`. As with many other programs, `tabcount` raises a question of program design, one that is faced primarily by the programmer but has implications for any users: At what point is it better to stop complicating the syntax of the program by adding features better left to other manipulations? The decision in this case was to omit options for calculating percents, cumulative percents, cumulative frequencies, etc., on the grounds that these are not difficult to calculate separately. We will see another example in which the reverse decision was taken. This sounds like inconsistency—indeed it is inconsistency—but the deeper principle being followed is to try to write a program to do one thing well and to make it as easy as possible to hand over to other programs when other processing is desired.

3 Making matrices

Tables may be thought of as composed of one or more matrices, which is very clear when we have merely a set of rows or a set of columns. For example, the correlations between two or more variables are usually presented as a table of a correlation matrix using `correlate` ([R] **correlate**). However, many projects require instead the compilation of a matrix of results from several different calculations. `makematrix` is here presented as a tool for such problems.

As matrices are standard data structures within Stata, you can do many things with them, such as joining them to other matrices, adding them, subtracting them, and so forth. Nevertheless, in working with matrices in Stata, one occasionally has to struggle with the consequences of an "official attitude", namely, that matrices are primarily what you use on the fly to fit models. Despite this, matrices are the nearest thing yet to table objects in Stata, so it is worth getting as far as you can with them.

Having said that, there are small limitations to the implementation of matrices in Stata. As one specific but sometimes irritating example, take the question of display format. Stata assumes, in effect, that the elements of a matrix are all quantities of the same kind, so that being able to specify a common format applying to all elements is as much flexibility as one needs. A correlation matrix is a case in point. `correlate` gives 4 decimal places in displaying correlations; if you wish to show fewer, as is common, there are various ways to put the correlation matrix into a named matrix, after which the format may be controlled using `matrix list, format()`. But a table of, say, frequencies, means and standard deviations, and skewness might call for no decimal places for the frequencies; some suitable precision for means and standard deviations; and, say, two or three decimal places for skewness. This cannot be achieved through `matrix list`. One remedy is to pass the matrix of results to `list`. `list` will be familiar as a staple interactive command, but in Stata 8 it is enhanced as a programmer's command so that it now offers improved ways of presenting data and results.

We will see how that works in a moment, but let us focus on the key point: `makematrix` runs a command repeatedly for a specified variable list (optionally, two variable lists) to produce a matrix of results. As usual, a matrix could be a vector. (As you may well know, Stata does not have a separate vector structure; row vectors and column vectors are just special cases of matrices, exactly as your linear algebra teachers insisted.) The matrix will be listed using `matrix list`, unless the `list` option is specified, in which case it will be listed using the `list` command. In other words, `makematrix` is a wrapper for `matrix list` and `list`, just as `tabcount` is a wrapper for `tabdisp`.

`makematrix` has various modes of operation, but first let us review some distinctions made purely for present purposes. Let us call a Stata (statistical) command univariate if it *requires* only one variable; it may repeat itself if supplied with two or more variables. `summarize` is a univariate command; it does work for two or more variables, by repeating its operation for those variables. Similarly, let us call such a command bivariate if it *requires* only two variables and may repeat itself otherwise. `correlate` is a bivariate command; it does work for three or more variables by repeating its operation for pairs of those variables. `spearman` ([R] **spearman**) is also a bivariate command, although it does not in fact accept more than two variables. Finally, let us call such a command multivariate if it produces just one set of results even if supplied with three or more variables. `regress` ([R] **regress**) is a multivariate command.

Consider again the typical output of `correlate` given a *varlist* of two or more variables, namely, a matrix of correlations for every pair of variables in *varlist*. How could we produce an equivalent directly for `spearman`? We need to find out that `spearman` leaves a correlation behind in `r(rho)`, ideally by reading the manual entry or alternatively by reverse engineering. Reverse engineering means—for an r-class command, such as `spearman`—using `return list` to see what is left behind, or—for an e-class command, such as `regress`—using `ereturn list` similarly.

```
. makematrix, from(r(rho)): spearman head trunk length displacement weight
                  headroom         trunk        length  displacement
    headroom             1
       trunk     .67678924             1
      length     .53235996     .71907323             1
displacement     .47845891     .57664675     .85248218             1
      weight     .52808385     .65644851     .94895697     .90538822
                    weight
      weight             1
```

The result is displayed using `matrix list`, and we will normally want to tidy up the presentation, say, by

```
. makematrix, from(r(rho)) format(%4.3f): spearman head trunk length
> displacement weight
                  headroom         trunk        length  displacement
    headroom         1.000
       trunk         0.677         1.000
      length         0.532         0.719         1.000
displacement         0.478         0.577         0.852         1.000
      weight         0.528         0.656         0.949         0.905
                    weight
      weight         1.000
```

However, let us leave these details of presentation on one side for a moment. In this example, given a bivariate command, a *varlist*, and a single result from which to compile the matrix, `makematrix` takes each pair of variables from *varlist*, runs a bivariate command for that pair, and puts a single result in the cell defined by each pair of variables. So, both rows and columns are specified by *varlist*.

Alternatively, we might want different sets of variables on the rows and the columns, perhaps specifying a submatrix of the full matrix. The option `cols()` can be used to specify variables to appear as columns. The variables in *varlist* will then appear as rows. Say that we did a principal component analysis of five variables and followed with calculation of scores:

```
. pca head trunk length displacement weight
(obs=74)
            (principal components; 5 components retained)
Component    Eigenvalue      Difference     Proportion     Cumulative
---------------------------------------------------------------------
     1        3.76201         3.02600        0.7524         0.7524
     2        0.73601         0.42791        0.1472         0.8996
     3        0.30809         0.15546        0.0616         0.9612
     4        0.15263         0.11136        0.0305         0.9917
     5        0.04127               .        0.0083         1.0000
               Eigenvectors
    Variable |      1          2          3          4          5
-------------+-------------------------------------------------------
    headroom |   0.35873    0.76396    0.52238   -0.12093    0.01297
       trunk |   0.43335    0.36648   -0.76764    0.29135    0.06120
      length |   0.48631   -0.23721   -0.10501   -0.57452   -0.60509
displacement |   0.46105   -0.33903    0.34841    0.70653   -0.22786
      weight |   0.48420   -0.33293    0.07372   -0.26689    0.76029
```

```
. score score1-score5
          (based on unrotated principal components)
             Scoring Coefficients
    Variable |       1           2           3           4           5
-------------+------------------------------------------------------------
    headroom |   0.35873     0.76396     0.52238    -0.12093     0.01297
       trunk |   0.43335     0.36648    -0.76764     0.29135     0.06120
      length |   0.48631    -0.23721    -0.10501    -0.57452    -0.60509
displacement |   0.46105    -0.33903     0.34841     0.70653    -0.22786
      weight |   0.48420    -0.33293     0.07372    -0.26689     0.76029

. makematrix, from(r(rho)) cols(score?): correlate head trunk length
> displacement weight
                 score1      score2      score3      score4      score5
    headroom  .69579216   .65541006   .28995191  -.04724258   .00263525
       trunk  .84053038    .3144061  -.42608327   .11382425   .01243294
      length  .94323831  -.20350815  -.05828833  -.22445161  -.12292224
displacement  .89424409  -.29085394   .19339097   .27602318  -.04628849
      weight  .93915804  -.28562389    .0409204  -.10426623   .15445146
```

Here, the full correlation matrix of variables and scores, as would be produced by `correlate`, is a 10×10 matrix, and the submatrix produced by `makematrix` is only a 5×5 matrix. The default number of decimal places is clearly ridiculous, and we would normally want to work on the column headers. The matrix result can be left in memory as a named matrix and then further manipulated:

```
. makematrix R, from(r(rho)) cols(score?): correlate head trunk length
> displacement weight
R[5,5]
                 score1      score2      score3      score4      score5
    headroom  .69579216   .65541006   .28995191  -.04724258   .00263525
       trunk  .84053038    .3144061  -.42608327   .11382425   .01243294
      length  .94323831  -.20350815  -.05828833  -.22445161  -.12292224
displacement  .89424409  -.29085394   .19339097   .27602318  -.04628849
      weight  .93915804  -.28562389    .0409204  -.10426623   .15445146

. matrix colnames R = "score 1" "score 2" "score 3" "score 4" "score 5"

. matrix li R, format(%4.3f)

R[5,5]
             score 1  score 2  score 3  score 4  score 5
    headroom   0.696    0.655    0.290   -0.047    0.003
       trunk   0.841    0.314   -0.426    0.114    0.012
      length   0.943   -0.204   -0.058   -0.224   -0.123
displacement   0.894   -0.291    0.193    0.276   -0.046
      weight   0.939   -0.286    0.041   -0.104    0.154
```

(*Continued on next page*)

Another application of the `cols()` option is perhaps more commonly desired:

```
. makematrix, from(r(rho) r(p)) label cols(price): spearman mpg-foreign
                              rho           p
       Mileage (mpg)  -.55546596   7.272e-07
  Repair Record 1978   .10275187   .40082135
       Headroom (in)    .1174198   .33661622
 Trunk space (cu ft)   .42395912   .00028325
        Weight (lbs)   .50135653   .00001143
         Length (in)   .50145304   .00001138
    Turn Circle (ft)   .32117803   .00712682
Displacement (cu in)   .41612747   .00037625
          Gear Ratio   -.3053873   .01072089
            Car type   .08065421   .51002468
. makematrix, from(r(rho) r(p)) list label format(%4.3f %6.5f) sep(0)
> cols(price): spearman mpg-foreign
```

	rho	p
Mileage (mpg)	-0.555	0.00000
Repair Record 1978	0.103	0.40082
Headroom (in.)	0.117	0.33662
Trunk space (cu. ft.)	0.424	0.00028
Weight (lbs.)	0.501	0.00001
Length (in.)	0.501	0.00001
Turn Circle (ft.)	0.321	0.00713
Displacement (cu. in.)	0.416	0.00038
Gear Ratio	-0.305	0.01072
Car type	0.081	0.51002

As this example shows, we can also ask for the results to be shown using the `list` command, which opens a wider range of presentation possibilities. The `label` option asks for variable labels to be shown, and the numeric variables can be assigned display formats on the fly. Those chosen were selected to emphasize this flexibility rather than to assert that p-values make sense to 5 decimal places.

As this example also shows, we can show two or more scalar results from each command run. This is possible in various ways. A univariate command can be repeated, each time yielding two or more scalars:

```
. makematrix, from(r(mean) r(sd) r(skewness)): su head trunk length displacement
> weight, detail
                  mean          sd    skewness
    headroom   2.9932432   .84599477   .14086508
       trunk   13.756757   4.2774042   .02920342
      length   187.93243    22.26634  -.04097455
displacement    197.2973   91.837219   .59165653
      weight   3019.4595   777.19357   .14811637
```

```
. makematrix, from(r(mean) r(sd) r(skewness)) list format(%2.1f %2.1f %4.3f)
> sep(0): su head trunk length displacement weight, detail
```

	mean	sd	skewness
headroom	3.0	0.8	0.141
trunk	13.8	4.3	0.029
length	187.9	22.3	-0.041
displacement	197.3	91.8	0.592
weight	3019.5	777.2	0.148

`makematrix` reasons in this way: The user wants three scalars, which I will show in three columns. So, I must run the command specified in turn on each variable supplied, which I will show on the rows. For each variable in varlist, `makematrix` runs a univariate command and puts two or more scalars in the cells of each row.

A bivariate command can be repeated, each time yielding two or more scalars:

```
. makematrix, from(r(rho) r(p)) lhs(rep78-foreign): spearman mpg
                    rho           p
       rep78   .30982668   .00957855
    headroom  -.48660171   .00001103
       trunk  -.64977398   3.759e-10
      weight  -.85755073   1.778e-22
      length   -.8314402   4.710e-20
        turn  -.75767499   5.548e-15
displacement  -.77126724   9.009e-16
  gear_ratio   .60982891   8.061e-09
     foreign   .36289624   .00148459
```

`makematrix` reasons in this way: The user wants two scalars, which I will show in two columns. So, I must run the command specified in turn on the variable supplied. The option `lhs()` is also specified, so that must be used to supply the other variable. Whenever `lhs()` is specified, it specifies the rows of the matrix; that is, in this case, the rows show the results from `spearman rep78 mpg` to `spearman foreign mpg`. Notice how the variables specified in `lhs()` appear on the left-hand side of the *varlist* that `spearman` runs. (`lhs()` also names the left-hand side of the matrix, but that is a happy accident.) This is also allowed:

```
. makematrix, from(r(rho) r(p)) rhs(rep78-foreign): spearman mpg
                    rho           p
       rep78   .30982668   .00957855
    headroom  -.48660171   .00001103
       trunk  -.64977398   3.759e-10
      weight  -.85755073   1.778e-22
      length   -.8314402   4.710e-20
        turn  -.75767499   5.548e-15
displacement  -.77126724   9.009e-16
  gear_ratio   .60982891   8.061e-09
     foreign   .36289624   .00148459
```

In this case, the rows show the results from `spearman mpg rep78` to `spearman mpg foreign` and are exactly the same as in the previous example. Again, whenever `rhs()`

is specified, it specifies the rows of the matrix. Notice how the variables specified in `rhs()` appear on the right-hand side of the *varlist* that `spearman` runs. (By a small stretch, you can also think of it as naming the right-hand side of the matrix, given that we could repeat the row names on that side.) In other cases, which option is used may well matter. Here is an example.

```
. makematrix, from(e(r2) e(rmse) _b[_cons] _b[mpg]) lhs(rep78-foreign) list
> dp(3 2 2 3) abb(9) sep(0) divider: regress mpg
```

	r2	rmse	_b[_cons]	_b[mpg]
rep78	0.162	0.91	1.96	0.068
headroom	0.171	0.78	4.28	-0.061
trunk	0.338	3.50	22.91	-0.430
weight	0.652	461.96	5328.76	-108.432
length	0.633	13.58	253.16	-3.063
turn	0.517	3.08	51.30	-0.547
displacement	0.498	65.52	435.85	-11.201
gear_ratio	0.380	0.36	1.98	0.049
foreign	0.155	0.43	-0.37	0.031

```
. makematrix, from(e(r2) e(rmse) _b[_cons] _b) rhs(rep78-foreign) list
> dp(3 2 2 3) abb(9) sep(0) divider: regress mpg
```

	r2	rmse	_b[_cons]	_b
rep78	0.162	5.41	13.17	2.384
headroom	0.171	5.30	29.77	-2.830
trunk	0.338	4.74	32.12	-0.787
weight	0.652	3.44	39.44	-0.006
length	0.633	3.53	60.16	-0.207
turn	0.517	4.05	58.80	-0.946
displacement	0.498	4.13	30.07	-0.044
gear_ratio	0.380	4.59	-2.26	7.813
foreign	0.155	5.36	19.83	4.946

The first series of regressions predicts `rep78` to `foreign` in turn from `mpg`. The second series predicts `mpg` from `rep78` to `foreign` in turn. The R^2 results will be the same, but not the root mean square errors or the intercepts or slopes, as these are two different sets of models. Note that `_b` by itself has the interpretation of `_b[`*row_variable*`]`. `dp()` is a lazy alternative to `format()` used to specify the number of decimal places.

In fact, `lhs()` and `rhs()` can be used to produce a series of multivariate results. Suppose that we have calculated `weightsq`, i.e., `weight^2`.

(*Continued on next page*)

```
. gen weightsq = weight^2
. makematrix, from(e(r2) e(rmse)) lhs(mpg-trunk length-foreign) list dp(3 2)
> sep(0) divider: regress weight weightsq
```

	r2	rmse
mpg	0.672	3.36
rep78	0.222	0.89
headroom	0.236	0.75
trunk	0.457	3.20
length	0.900	7.12
turn	0.736	2.29
displacement	0.826	38.90
gear_ratio	0.577	0.30
foreign	0.379	0.37

This series of models predicts `mpg` to `foreign` in turn from `weight` and `weightsq`. When either `lhs()` or `rhs()` is specified, they define the varying rows, while the *varlist* supplied is fixed for each run of the command.

There is one more nuance to be explained. Say that you want a table of sums for a set of variables. You might try

```
. makematrix, from(r(sum)): su head trunk length displacement weight, meanonly
```

	headroom	trunk	length	displacement
headroom	221.5	221.5	221.5	221.5
trunk	1018	1018	1018	1018
length	13907	13907	13907	13907
displacement	14600	14600	14600	14600
weight	223440	223440	223440	223440

	weight
headroom	221.5
trunk	1018
length	13907
displacement	14600
weight	223440

However, `makematrix` cannot distinguish between this and a similar problem with a bivariate command; it will thus attempt to run `summarize` on all distinct pairs of variables. This will succeed, except that what is left behind in `r(sum)` will be the sum of the second of each pair of variables. What you will prefer is a vector, and that is the option to specify:

```
. makematrix, from(r(sum)) vector: su head trunk length displacement weight,
> meanonly
```

	sum
headroom	221.5
trunk	1018
length	13907
displacement	14600
weight	223440

There is more, for which please see the help as usual.

4 Group frequencies

One point emphasized several times so far in the previous column and in this one is the scope for using `list` for tabulations. We have just seen how specifying a `list` option in `makematrix` opens up the presentation possibilities of the `list` command. In the same way, the final example in this paper is a command based on `list` as a display command. Both also can capitalize on the many new features introduced in `list` in Stata 8.

Everyone knows that, even with two-way tables, there can be too many columns for comfort. The problem of space is usually compounded with three-way and higher tables. Even if there is enough space, the sparsity (lots of zeroes) of some tables makes other kinds of tabulation attractive in at least some circumstances.

`groups` is perhaps best explained—in terms of what it does, rather than precisely how it is implemented—as a hybrid or cross-breed of `tabulate` and `list`. The results of `groups foreign` look very much like the results of `tabulate foreign`, and indeed `groups` is designed to be that way:

```
. groups foreign
```

foreign	Freq.	Percent	Cum.
Domestic	52	70.27	70.27
Foreign	22	29.73	100.00

A two-way table, on the other hand, is, as it were, stretched downwards so that it is a listing, a "long" structure rather than a "wide" one in the jargon especially associated with `reshape` ([R] **reshape**). The same principle is also applied to three-way and higher tables.

```
. groups foreign rep78
```

foreign	rep78	Freq.	Percent
Domestic	1	2	2.90
Domestic	2	8	11.59
Domestic	3	27	39.13
Domestic	4	9	13.04
Domestic	5	2	2.90
Foreign	3	3	4.35
Foreign	4	9	13.04
Foreign	5	9	13.04

(*Continued on next page*)

A `fillin` option is available for Sartrean existentialists who like to contemplate nothingness:

```
. groups foreign rep78, fillin
```

foreign	rep78	Freq.	Percent
Domestic	1	2	2.90
Domestic	2	8	11.59
Domestic	3	27	39.13
Domestic	4	9	13.04
Domestic	5	2	2.90
Foreign	1	0	0.00
Foreign	2	0	0.00
Foreign	3	3	4.35
Foreign	4	9	13.04
Foreign	5	9	13.04

`groups` can be issued `by` *varlist*`:`. That is the key to how percents are calculated. At the same time, let us look at the option `order(h)`, which puts the highest frequencies first, and the option `N`, which is an option of `list`:

```
. bysort foreign: groups rep78, ord(h) N
```

-> foreign = Domestic

rep78	Freq.	Percent	Cum.
3	27	56.25	56.25
4	9	18.75	75.00
2	8	16.67	91.67
1	2	4.17	95.83
5	2	4.17	100.00
N	5	5	5

-> foreign = Foreign

rep78	Freq.	Percent	Cum.
4	9	42.86	42.86
5	9	42.86	85.71
3	3	14.29	100.00
N	3	3	3

The frequencies shown by default are raw frequencies and percents, with one or more variables in *varlist*, and cumulative percents with just one variable in *varlist*. The underlying surmise is that cumulatives are rather more arbitrary with two or more variables, being necessarily dependent on the order of variables. That is not the law, however, and a `show()` option allows you to have none or one or two or three of those—and, indeed, cumulative frequencies are also available on request:

```
. groups mpg, show(f F)
```

mpg	Freq.	Cum.
12	2	2
14	6	8
15	2	10
16	4	14
17	4	18
18	9	27
19	8	35
20	3	38
21	5	43
22	5	48
23	3	51
24	4	55
25	5	60
26	3	63
28	3	66
29	1	67
30	2	69
31	1	70
34	1	71
35	2	73
41	1	74

Here, `f` stands for frequency, and `F` stands for cumulative frequency. In addition, reverse cumulatives, number or percent greater, rather than number or percent less than or equal to, are also available. As a special case, there is also a `show(none)`, which is more useful than it sounds, as the next example shows.

A further option, `select()`, lets you select which groups are to be listed, for example by a condition on the frequencies. `select(f == 1)` selects those groups that occur precisely once, in which case there is no need to see a frequency column of 1s, and the percents and cumulative percents are possibly of no use or interest:

```
. groups mpg, sel(f == 1) show(none)
```

mpg
29
31
34
41

Note, by the way, that the first principles solution

```
. bysort varlist: list if _N == 1
```

shows precisely this information, plus rather a lot of unwanted junk.

The `select()` option can be used in another way. `select(5)` says, list just the first five of the groups that would otherwise have been listed. By default, with just one variable specified, that is just the five lowest groups of values of the variable. Each group, naturally, could occur more than once:

```
. groups mpg, sel(5)
```

mpg	Freq.	Percent	Cum.
12	2	2.70	2.70
14	6	8.11	10.81
15	2	2.70	13.51
16	4	5.41	18.92
17	4	5.41	24.32

You can now guess that `select(-5)` starts at the other end and counts downwards, so it says, list just the last five of the groups that would otherwise have been listed.

```
. groups mpg, sel(-5)
```

mpg	Freq.	Percent	Cum.
30	2	2.70	93.24
31	1	1.35	94.59
34	1	1.35	95.95
35	2	2.70	98.65
41	1	1.35	100.00

In other words, these commands give you pictures of the tails of a distribution.

You can specify `order(high)` or `order(low)`, namely, specify a listing in order of the frequencies, not the values of the variables in each group. In the first case, `select(5)` gives you the 5 groups that are most frequent.

```
. groups mpg, sel(5) ord(h)
```

mpg	Freq.	Percent	Cum.
18	9	12.16	12.16
19	8	10.81	22.97
14	6	8.11	31.08
21	5	6.76	37.84
22	5	6.76	44.59

If you specify `fillin` (compare with [R] **fillin**) with two or more variables, groups of those variables with zero frequencies are shown explicitly. These are the cells that would be shown by 0s with `tabulate` or by blanks with `table`. `select()`ing zeros gives you a listing of the cells not present in your dataset. That is not often wanted, but when it is, it can be tricky to automate, unless you know about `fillin`, the command after which the option is named. So, we are almost back where we started, with a stab at displaying values not present in the dataset.

```
. groups foreign rep78, fill sel(f == 0) show(none)
```

foreign	rep78
Foreign	1
Foreign	2

`groups` is just sitting on the shoulders of the giant `list`, so there are several ways to tweak appearances. Here is one:

```
. groups foreign rep78, sepby(foreign)
```

foreign	rep78	Freq.	Percent
Domestic	1	2	2.90
Domestic	2	8	11.59
Domestic	3	27	39.13
Domestic	4	9	13.04
Domestic	5	2	2.90
Foreign	3	3	4.35
Foreign	4	9	13.04
Foreign	5	9	13.04

We did get the same appearance earlier, but that was just fortuitous, as the default of separating lines every 5 happened to give a sensible answer.

5 Tabling: an agenda

Although we have reviewed many different approaches to tabulation and perhaps demonstrated to you that more is possible and that more is easy than you previously thought, there are tabulation problems at present that are difficult or impossible given Stata's present capabilities. What follows is, necessarily, a partial and personal list.

Combining tables One common need is to put together what are, in effect, sub-tables into combined tables. It could be argued that Stata should not interfere between you and your word or text processor; anyway, at first sight, it offers next to no tools for doing this, except that, in a sense, there is a bunch of commands for joining tables so long as they are (expressible as) Stata matrices. This line of attack is probably underappreciated; at the same time, it falls short of what I guess people often need here.

In the long run, we may need a miniature language for combining tables. In effect, tables could be seen as objects, and there would be a set of operations for combining them, with tunable control of output form, e.g., elementwise addition, subtraction, multiplication, division; joining along rows; joining along columns; layering. Each combining would produce alignment and be more than what anybody could do as a cut/copy/paste exercise. I guess that this would be a substantial project for StataCorp.

Multiple variables Stata does not offer much support for tabulating frequency or proportion or percent results from several variables simultaneously. Suppose, for example, I have variables on trips to the theater, cinema, opera house, funfair, etc., and I want a single table for all variables so I can compare frequency distributions.

Some approaches in this area were previously discussed by Cox and Kohler (2003). Much can be done once you see that a different data structure is often the key (using `stack` and especially `reshape`, etc.), but most users understandably prefer getting results on the fly to mapping to a different data structure. (Even seeing that you need a different structure can depend on a lot of experience. Doing the restructuring can be tricky, too.)

Sorting Sorting on the margins is often of limited analytical use. To see patterns rather than to provide easy lookup (what is the population of Texas? Look under "Texas" ...), you often need to sort tables on their contents (i.e., cell entries). From Stata 8, `tabulate` has a sort option, but in general, sorting of tables is not provided very widely.

Cell composites What I call cell composites are cells containing values from two or more variables, whether variables in your dataset or temporary variables constructed by the command running. Suppose that you wanted cells with concatenated strings

cell_frequency (*row_percent*)

This is quite distinct cosmetically from what might be called cell stacks, *cell_frequency* presented above *row_percent*.

In general, Stata directly supports cell stacks but not output like the first form. Cell stacks can be more space-consuming and difficult to read in some circumstances, although it is also easy to run out of space with the first form.

Much is possible once you see that setting up tabulation as a display of string variables is the key. However, this requires some prior manipulations and, indeed, moderate fluency with some Stata basics. Canned solutions, whether official commands or user-written programs, appear lacking. What would be most desirable is support for output specifications, so that if I want a table to show *cell_frequency* (*row_percent*), something like

```
"#1 (#2)"
```

would specify "the first number followed by a space followed by a parenthesis followed by the second number followed by a parenthesis".

Cell text Think of the number of ways in which you might specify substantive missings as one example. Depending on the boss's whims, the house rules, the journal's prescribed style, or your own tastes, you could want `NA` or `--` or `(no data)`, and so forth. This is an example of how, frequently, even in a numeric table, you often want extra text. Or think of cell entries that are footnoted. Again, much is possible once you see that setting tabulation up as a display of string variables is the key. However, this again requires some prior manipulations and, indeed, moderate fluency with some Stata basics.

Table design In fact, we can easily extend this. This last problem is really a rag-bag of all sorts of small and large design issues, such as support for different fonts and bold, italic, etc.; different kinds of dividers and separators; control of titles, subtitles, notes, etc.; control of margin layout; and multiple formats, as highlighted in the discussion of `makematrix`.

There is a territorial issue here, as with our very first problem, on how far Stata should get into terrain that normally you would negotiate with (or in some cases without) the assistance of your word or text processing software. A lot can be done with SMCL, but either for one-off tasks or for repetitive tasks, that often requires Stata programming or at least considerable Stata expertise.

6 Summary

The practical importance of tables can need little emphasis. In this column and its predecessor, I have selected some highlights from the wide range of possibilities opened up by both official and user-written commands. That still leaves a large agenda. Do seize your opportunity of bringing tabulation needs to the attention of StataCorp.

7 What is next?

Graphics, graphics, graphics, and graphics. The year 2004 will be graphics year for this column, as we look at kinds of graphs, how to choose them, how to get them, how to tweak them, and how to use the new Stata graphics.

8 Acknowledgments

Kit Baum, Michael Blasnik, Shannon Driver, Ken Higbee, and Fred Wolfe took part in helpful interactions while these programs, or their predecessors, were developed.

9 References

Cox, N. J. 2003. Speaking Stata: Problems with tables, Part I. *Stata Journal* 3(3): 309–324.

Cox, N. J. and U. Kohler. 2003. Speaking Stata: On structure and shape: the case of multiple responses. *Stata Journal* 3(1): 81–99.

Newson, R. 2003. Confidence intervals and p-values for delivery to the end user. *Stata Journal* 3(3): 245–269.

About the Author

Nicholas Cox is a statistically minded geographer at the University of Durham. He contributes talks, postings, FAQs, and programs to the Stata user community. He has also co-authored fourteen commands in official Stata. He was an author of several inserts in the *Stata Technical Bulletin* and is Executive Editor of *The Stata Journal*.