Title

stata.com

Collection principles — Tags, dimensions, levels, and layout from first principles

Description Remarks and examples Also see

Description

This entry is a self-contained introduction to tags, dimensions, and levels and how you use them in `collect layout` to specify and create tables. It introduces other commands that are helpful in laying out tables along the way. It uses simple examples on real data to demonstrate all concepts.

It explains what tags are and why they are organized into dimensions that contain levels. It explains the inner workings of `collect layout` so you can understand when things do not go as you expect. It demonstrates how to create one-way, two-way, multiway, and stacked tables and discusses what to do when things go wrong.

Admittedly, there is quite a bit of overlap with [TABLES] **Intro 2**. Unlike **Intro 2**, this entry is focused solely on laying out tables.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Basic concepts
Basics in practice
How collect layout processes tag specifications
The process in practice

Basic concepts

How do you make collections work for you? The answer is you just use tags organized into the levels of dimensions to request tabular results. What? Let's give meaning to that sentence.

We start by collecting something. That something will be incredibly simple. The undocumented Stata command `echo` simply displays whatever number or string you type and returns that number or string in `r(value)`.

```
. echo 11
value = 11
. return list
scalars:
               r(value) =  11
```

To collect its results, we simply prefix our `echo` command with `collect:`, but let's do a little more. Let's collect the result and give it the tag `myres1`.

```
. collect, tag(myres1[]): echo 11
value = 11
```

Do not worry for now about the `[]` after `myres1`; just know that we have collected the value 11 and tagged it with `myres1`.

The collect system is built to create tables of results, perhaps lots of different results from different commands. The way we get things out of a collection is to lay out a table. We have only one value collected, so let's create the world's simplest table.

```
. collect layout (myres1)
Collection: default
      Rows: myres1
   Table 1: 1 x 1

┌────┐
│ 11 │
└────┘
```

`collect layout` is the command to specify the layout of a table. Its first argument is a parentheses-bound list of the tags that we want on the rows of the table, in this case `(myres1)`. A tag is simply a way to name and find things. We tagged our value 11 as `myres1`. When we asked for `myres1`, `collect layout` gave our 11 back to us.

You may have noticed that we did not include the `[]` on `myres`. We could have; it would make no difference.

Let's add another value to our collection.

```
. collect, tag(myres2[]): echo 22
value = 22
```

And let's show "all" of this as a table.

```
. collect layout (myres1 myres2)
Collection: default
      Rows: myres1 myres2
   Table 1: 2 x 1

┌────┐
│ 11 │
│ 22 │
└────┘
```

We could go on, but I think we are going to get tired of typing `myres1` ....

Tags do not have to be a simple name; in fact, they rarely are. Tables tend to put a set of related things on the rows and another set of related things on the columns. The contents of the table are the intersection of those related things. Consider a cross-tabulation of `region` and `sex`.

	Sex	
	Male	Female
Region		
NE	1,018	1,078
MW	1,310	1,464
S	1,332	1,521
W	1,255	1,373

"NE", "MW", "S", and "W" are the related things on the rows. "Male" and "Female" are the related things on the columns. The counts in the cells of the table are the intersection when both the row "thing" and column "thing" are true. On this table, that is all obvious, but it is also at the heart of how tags are used in the collect system.

Tags in the collect system provide a structure that directly supports sets of related things. Tags are organized as dimensions that contain levels. In the table above, `region` is a dimension, and the levels are `NE`, `MW`, `S`, and `W`. Likewise, `sex` is another dimension whose levels are `Male` and `Female`.

If this all seems like an unnecessary abstraction, it is not. The table above was a simple cross-tabulation of two categorical variables. But that need not be the case. One of our dimensions might be sets of regressions with different covariates. Or it might be sets of results from different datasets. All categorical variables can be dimensions, but not all dimensions can be categorical variables.

Let's now use the level within dimension organization to create a more interesting table. First, we clear our current collection.

```
. collect clear
```

We collect the results of an `echo` but give it two tags.

```
. collect, tag(myrow[1] mycol[1]): echo 11
value = 11
```

We have tagged value 11 with `myrow[1]` and `mycol[1]`. We read tag `myrow[1]` as "dimension `myrow`, level 1" or "level 1 in dimension `myrow`".

Let's collect and tag more results from `echo` commands.

```
. collect, tag(myrow[2] mycol[1]): echo 21
value = 21
. collect, tag(myrow[1] mycol[2]): echo 12
value = 12
. collect, tag(myrow[2] mycol[2]): echo 22
value = 22
```

You might see where this is heading.

Now, we can create a table from our four collected values,

```
. collect layout (myrow[1] myrow[2]) (mycol[1] mycol[2])
Collection: Table
      Rows: myrow[1] myrow[2]
   Columns: mycol[1] mycol[2]
   Table 1: 3 x 2

──────────────────
          mycol
          1    2
──────────┼───────
myrow     │
  1       │ 11   12
  2       │ 21   22
──────────┴───────
```

The first parentheses-bound list still specifies the tags we want on the rows of our table. The second parentheses-bound list specifies the tags we want on the columns of our table.

You can specify multiple levels inside the `[]`; thus, a better way to type the layout command above is

```
. collect layout (myrow[1 2]) (mycol[1 2])
```

If you are following along, type it. You will get the same result.

Better still, you can refer to an entire dimension and all the tags defined by its levels by typing just the dimension name. The concise way to specify our table is

```
. collect layout (myrow) (mycol)
```

And now you see why `collect` organizes its tags as levels within dimensions.

Let's elaborate on that point just a bit. You can tell from this example that it may take more than one tag to uniquely identify a value. Each of our 4 values required 2 tags, for example, value 12 required `myrow[1]` and `mycol[2]`. Thus, there is a great advantage to representing tags as levels within dimensions. If it takes two tags to uniquely identify a value and you organize those tags as the rows and columns of a table, your values will naturally populate the cells of a table.

Moreover, this idea generalizes to higher-dimensional tables. If each of your values requires 3 tags and those tags can be arranged in 3 dimensions, you have the makings of a 3-dimensional (3D) table. One rarely presents 3D tables as their natural cube. It is hard to print. They are usually presented as tables with super rows or super columns. Regardless, dimensions give you a natural way to specify the structure of a table, whether that structure is a simple table with rows and columns or it is a table with columns, super columns, rows, super rows, and super-super rows.

If you came here just to learn about the terms "tag", "dimension", and "level", you can stop reading.

Basics in practice

Let's put this organization to use on a real collection.

Grab the venerable (but familiar) automobile dataset.

```
. sysuse auto
(1978 automobile data)
```

Clear our default collection.

```
. collect clear
```

And collect the results of a simple regression.

```
. collect: regress mpg displacement i.foreign
```

Source	SS	df	MS	Number of obs	=	74
				F(2, 71)	=	35.57
Model	1222.85283	2	611.426414	Prob > F	=	0.0000
Residual	1220.60663	71	17.1916427	R-squared	=	0.5005
				Adj R-squared	=	0.4864
Total	2443.45946	73	33.4720474	Root MSE	=	4.1463

mpg	Coefficient	Std. err.	t	P>|t|	[95% conf.	interval]
displacement	-.0469161	.0066931	-7.01	0.000	-.0602618	-.0335704
foreign						
Foreign	-.8006817	1.335711	-0.60	0.551	-3.464015	1.862651
_cons	30.79176	1.666592	18.48	0.000	27.46867	34.11485

Just so you know, every number saved in the `e()` results after `regress`, which includes every number displayed in the results above, has been pulled into the collection. You just have to tell `collect` how you would like them pulled out and displayed.

But, you wonder, we did not specify any tags. What can we possibly do? We can do a lot. `collect` creates tags for us behind the scenes. We get a list of the dimensions by typing

```
. collect dims
Collection dimensions
Collection: default
```

Dimension	No. levels
Layout, style, header, label	
cmdset	1
coleq	1
colname	4
colname_remainder	1
foreign	2
program_class	1
result	30
result_type	3
Style only	
border_block	4
cell_type	4

Let's focus on two of those dimensions. First, `colname`,

```
. collect levelsof colname
Collection: default
 Dimension: colname
    Levels: displacement 0.foreign 1.foreign _cons
```

Those look promising. They are the coefficient names from our regression. And, yes, the levels are strings—`displacement`, `0.foreign`, `1.foreign`, and `_cons`. Dimension levels can be either integers or strings, and the strings can have spaces if you wish.

We apologize for the name `colname`; it is a bit arcane. It comes from the fact that Stata matrices have `colnames`, and this dimension was taken from the `colnames` on the `e(b)` matrix saved by `regress`. We will also find that many different commands save many different things that need to go into the `colname` dimension. There simply is no good name for all the levels that can appear in `colname`. If it makes you feel any better, everyone does eventually get used to typing `colname`.

If you really cannot abide `colname`, you can actually change it. Type

```
. collect remap colname = parameters
```

Now, you can type `parameters` instead of `colname`.

Second, the dimension, `result`, sounds truly promising.

```
. collect levelsof result
Collection: default
 Dimension: result
    Levels: F N _r_b _r_ci _r_df _r_lb _r_p _r_se _r_ub _r_z cmd cmdline
            depvar df_m df_r estat_cmd ll ll_0 marginsok model mss predict
            properties r2 r2_a rank rmse rss title vce
```

That is a lot to figure out. We recognize some things: `r2` sounds as if it might be "R^2", and `rmse` might be "Root mean squared error". What about those underscore things—`_r_b`, `_r_se`, `_r_ci`. We might guess. Let's not. Let's use another command that gives us a bit more information, `collect label list`.

```
. collect label list result, all
  Collection: default
   Dimension: result
       Label: Result
Level labels:
           F  F statistic
           N  Number of observations
        _r_b  Coefficient
       _r_ci  __LEVEL__% CI
       _r_df  df
       _r_lb  __LEVEL__% lower bound
        _r_p  p-value
       _r_se  Std. error
       _r_ub  __LEVEL__% upper bound
        _r_z  t
         cmd  Command
     cmdline  Command line as typed
      depvar  Dependent variable
        df_m  Model DF
        df_r  Residual DF
   estat_cmd  Program used to implement estat
          ll  Log likelihood
        ll_0  Log likelihood, constant-only model
   marginsok  Predictions allowed by margins
       model  Model
         mss  Model sum of squares
     predict  Program used to implement predict
  properties  Command properties
          r2  R-squared
        r2_a  Adjusted R-squared
        rank  Rank of VCE
        rmse  RMSE
         rss  Residual sum of squares
       title  Title of output
         vce  SE method
```

We have listed all the levels of dimension `result` and the labels for each level. Now this dimension does look promising; it includes all the results from the regression. Apparently, level `_r_b` is the level that refers to the coefficients. `_r_se` refers to the standard errors of the coefficients. `_r_ci` is a little odd because apparently it contains a placeholder for the level of significance. Regardless, it looks like a confidence interval. Many of the levels are a one-to-one match with the names of the `e()` results—`df_m`, `df_r`, `ll`, `r2`, ... In fact, all the `e()` results are here, and they have the same names they had in `e()`.

We say all the `e()` results, but that is not quite true. `e(V)` is excluded unless you explicitly collect it. Why would we need the full VCE? Also, `e(b)` is not here. It is effectively here because you can use the level `_r_b` to access the coefficient values.

It seems as if we have enough information to pull some values out of the collection using their tags. Let's pull the value for R^2. From the listing above, we know the dimension (`result`) and level (`r2`) of its tag.

```
. collect layout (result[r2])
Collection: default
      Rows: result[r2]
   Table 1: 1 x 1

─────────────────────────
R-squared │    .5004596
─────────────────────────
```

How about grabbing all the results by just using the whole `result` dimension.

```
. collect layout (result)
Collection: default
      Rows: result
   Table 1: 22 x 1
```

F statistic	35.56533
Number of observations	74
Command	regress
Command line as typed	regress mpg displacement i.foreign
Dependent variable	mpg
Model DF	2
Residual DF	71
Program used to implement estat	regress_estat
Log likelihood	-208.7139
Log likelihood, constant-only model	-234.3943
Predictions allowed by margins	XB default
Model	ols
Model sum of squares	1222.853
Program used to implement predict	regres_p
Command properties	b V
R-squared	.5004596
Adjusted R-squared	.4863881
Rank of VCE	3
RMSE	4.146281
Residual sum of squares	1220.607
Title of output	Linear regression
SE method	ols

Well, that is both more and less than we probably expected. Regarding the "more", we probably do not care about "Command" or "Command line" or several of the other string results (really macro results). Let's ask specifically for what we want and for the order we want.

```
. collect layout (result[N F df_r df_m r2 r2_a rmse ll])
Collection: default
      Rows: result[N F df_r df_m r2 r2_a rmse ll]
   Table 1: 8 x 1
```

Number of observations	74
F statistic	35.56533
Residual DF	71
Model DF	2
R-squared	.5004596
Adjusted R-squared	.4863881
RMSE	4.146281
Log likelihood	-208.7139

More importantly, where are our coefficients? The answer is that no coefficient can be uniquely identified by just the tag `result[_r_b]`. There were three coefficients in our model, one for `displacement`, one for `1.foreign`, and one for `_cons`. Tag `result[_r_b]` refers to all of those, but `collect layout` needs us to tell it where each of those coefficients goes in the table. We have not done that. Just as we needed both a `row` and a `column` dimension to create our table in *Basic concepts*, we need another dimension to create a table with coefficients. Recall that the `colname` dimension enumerated the coefficient names; that is what we need.

```
. collect layout (colname) (result[_r_b])
Collection: default
      Rows: colname
   Columns: result[_r_b]
   Table 1: 4 x 1
```

	Coefficient
Displacement (cu. in.)	-.0469161
Domestic	0
Foreign	-.8006817
Intercept	30.79176

We put the `colname` dimension on our table's rows and the `result` dimension on our table's columns. We also limited the `result` dimension to the level `_r_b`.

Let's get a more complete regression table by adding some levels to the `result` dimension.

```
. collect layout (colname) (result[_r_b _r_se _r_p _r_ci])
Collection: default
      Rows: colname
   Columns: result[_r_b _r_se _r_p _r_ci]
   Table 1: 4 x 4
```

	Coefficient	Std. error	p-value	95% CI	
Displacement (cu. in.)	-.0469161	.0066931	0.000	-.0602618	-.0335704
Domestic	0	0			
Foreign	-.8006817	1.335711	0.551	-3.464015	1.862651
Intercept	30.79176	1.666592	0.000	27.46867	34.11485

How collect layout processes tag specifications

When we specify layouts, it is helpful to understand what `collect layout` does with the tags we specify for the rows and columns. When we type

```
. collect layout (result[N F r2])
```

a search is performed to see whether any values are tagged `result[N]`. If exactly one value with that tag is found, `collect layout` creates a row in the table for `result[N]` and places that value into the newly created row. If nothing is found with that tag, `collect layout` does nothing. If more than one thing with that tag is found, `collect layout` does nothing. Then, the process repeats for values tagged `result[F]` and finally `result[r2]`. That is it.

When we type the command

```
. collect layout (result)
```

the process is as we just described, but it is done for every level in the dimension `result`, not just for the levels `N`, `F`, and `r2`.

Let's call this process enumerating the levels of a dimension.

Enumerating a single dimension is all that is required for a one-way table, like the one we just specified. Two-way tables add just a bit to this process. When we type

```
. collect layout (colname) (result)                                    (1)
```

the command not only enumerates the levels in dimension `colname` and `result` but also interacts all the levels of `colname` and `result`. Let's specify the levels we want to make this a bit easier to explain.

```
. collect layout (colname[displacement _cons]) (result[_r_b _r_se r2])  (2)
```

`collect layout` begins with the tag `colname[displacement]`, which might form the first row. It looks sequentially for all pairings of `colname[displacement]` with the levels of `result`. It looks first for values that are tagged `colname[displacement]` and tagged `result[_r_b]`. If it finds exactly one value, it creates the row for `colname[displacement]` and the column for `result[_r_b]` and places the value it finds in that row/column position. It then looks for values tagged `colname[displacement]` and `result[_r_se]`. If it finds exactly one value with those tags, it places that value in the correct row and column. It then does the same thing for the tags `colname[displacement]` and `result[r2]`. That completes the process for the `displacement` row in the table.

`collect layout` then repeats that whole process with `colname[_cons]` to create the potential second row in the table.

Why do we say "potential" second row? Because it is possible that for some pairings of the levels of `colname` and `result`, `collect layout` will not find a unique value. Or that it will always find multiple values. If either happens for a whole row or column, then that row or column is not created.

The whole process is hardly different when we type

```
. collect layout (colname) (result)                                    (3)
```

In this case, `collect layout` enumerates over all the levels of `result` within all the levels of `colname`, rather than just the three levels of `result[_b_r _b_se r2]` within the two levels of `colname[displacement _cons]`, which we explicitly specified in (2).

An important thing to realize is that `collect layout` must find exactly one thing or it does nothing. Why can't it handle finding more than one thing? The row and column arguments to `collect layout` specify both what to look for and where to put it. Each level from the row specification is a possible row for the table. Each level from the column specification is a possible column for the table. Finding multiple values for a row and column combination means that we have not told `collect layout` where those values go. It means that we have not included enough dimensions in our specification.

We can also tell you that in (2) our use of `r2` in `result[_r_b _r_se r2]` had no effect on the table. It yields exactly the same table as typing `result[_r_b _r_se]`. Type it and see. Why? Because the value of R^2 is a model statistic, not a coefficient. It is not tagged with any specific variable. It is not tagged with `colname[displacement]` or with `colname[_cons]`. You cannot find model statistics when the `result` dimension is interacted with `colname`. More on that in *The process in practice*.

`collect layout` always interacts row and column specifications. That is really what makes a table a table. We can also explicitly specify interactions. That lets us create multiway tables rather than just two-way tables.

The process in practice

Our collection currently has a single regression. What if we wanted to compare that regression with another regression? Let's add `weight` to our regression and collect those results.

```
. collect: regress mpg displacement i.foreign weight
```

Source	SS	df	MS	Number of obs	=	74
				F(3, 70)	=	45.88
Model	1619.71935	3	539.906448	Prob > F	=	0.0000
Residual	823.740114	70	11.7677159	R-squared	=	0.6629
				Adj R-squared	=	0.6484
Total	2443.45946	73	33.4720474	Root MSE	=	3.4304

mpg	Coefficient	Std. err.	t	P>|t|	[95% conf.	interval]
displacement	.0019286	.0100701	0.19	0.849	-.0181556	.0220129
foreign						
Foreign	-1.600631	1.113648	-1.44	0.155	-3.821732	.6204699
weight	-.0067745	.0011665	-5.81	0.000	-.0091011	-.0044479
_cons	41.84795	2.350704	17.80	0.000	37.15962	46.53628

We might want to see how the additional covariate affects the coefficient on `displacement`.

```
. collect layout (colname) (result[_r_b _r_se _r_p _r_ci])
Collection: default
      Rows: colname
   Columns: result[_r_b _r_se _r_p _r_ci]
   Table 1: 1 x 4
```

	Coefficient	Std. error	p-value	95% CI	
Weight (lbs.)	-.0067745	.0011665	0.000	-.0091011	-.0044479

That is disappointing. We typed just what we typed to create a table from a single regression. We added another whole regression, and we get just one row?

Let's apply what we learned in *How collect layout processes tag specifications*. The first thing `collect layout` searched for was the first level of dimension `colname` interacted with the first specified level of dimension `result`. That would be the two tags `colname[displacement]` and `result[_r_b]`. That search finds two values: −0.047 from the first regression and 0.002 from the second regression. `collect layout` did not find a unique value, so it did nothing. That same thing happens when `collect layout` searches for `colname[displacement]` in combination with `result[_r_se]`, `result[_r_p]`, and `result[_r_ci]`. So there is nothing to report for the whole potential first row. The whole sequence happens again for the second level of `colname`—`colname[0.foreign]`. Two values are again found for each of the specified levels of `result`.

The only time `collect layout` finds a single value for each level of `result` is when it enumerates the `weight` level of dimension `colname`. That is the only coefficient that appears in only one of our two regressions. We clearly need to somehow add a dimension to our table, a dimension whose levels represent our regressions.

Let's again list all the dimensions in our collection and see whether there is anything promising.

```
. collect dims
Collection dimensions
Collection: default
```

Dimension	No. levels
Layout, style, header, label	
cmdset	2
coleq	1
colname	5
colname_remainder	1
foreign	2
program_class	1
result	30
result_type	3
Style only	
border_block	4
cell_type	4

`cmdset` looks promising. Let's learn a bit more about that dimension.

```
. collect label list cmdset, all
  Collection: default
   Dimension: cmdset
       Label: Command results index
Level labels:
           1
           2
```

We see `Command results index`, which does indeed look promising.

How do we add that dimension? We previously hinted that multiway tables could be specified by interacting additional dimensions with those already specified on the rows or columns. We perform that interaction using the same operator we use to create interactions in factor variables—`#`.

Let's try interacting dimension `cmdset` with dimension `colname`. We will interact with `colname` because it is on the row dimension and we do not have room for any more columns.

```
. collect layout (colname#cmdset) (result[_r_b _r_se _r_p _r_ci])
Collection: default
      Rows: colname#cmdset
   Columns: result[_r_b _r_se _r_p _r_ci]
   Table 1: 14 x 4
```

	Coefficient	Std. error	p-value	95% CI	
Displacement (cu. in.)					
1	-.0469161	.0066931	0.000	-.0602618	-.0335704
2	.0019286	.0100701	0.849	-.0181556	.0220129
Domestic					
1	0	0			
2	0	0			
Foreign					
1	-.8006817	1.335711	0.551	-3.464015	1.862651
2	-1.600631	1.113648	0.155	-3.821732	.6204699
Weight (lbs.)					
2	-.0067745	.0011665	0.000	-.0091011	-.0044479
Intercept					
1	30.79176	1.666592	0.000	27.46867	34.11485
2	41.84795	2.350704	0.000	37.15962	46.53628

That is not bad. We have all the coefficients, standard errors, p-values, and confidence intervals from both regressions. They are not exactly organized the way they are in most comparative regression tables.

Let's go for that organization. We will need to put the dimension `cmdset` onto the columns, and then interact the coefficient names (dimension `colname`) with the statistics (dimension `result`) on the rows. We have room on the rows, so let's just ask for all the levels of dimension `result`.

```
. collect layout (colname#result) (cmdset)

Collection: default
      Rows: colname#result
   Columns: cmdset
   Table 1: 40 x 2
```

	1		2	
Displacement (cu. in.)				
Coefficient		-.0469161		.0019286
95% CI	-.0602618	-.0335704	-.0181556	.0220129
df		71		70
95% lower bound		-.0602618		-.0181556
p-value		0.000		0.849
Std. error		.0066931		.0100701
95% upper bound		-.0335704		.0220129
t		-7.01		0.19
Domestic				
Coefficient		0		0
df				
Std. error		0		0
Foreign				
Coefficient		-.8006817		-1.600631
95% CI	-3.464015	1.862651	-3.821732	.6204699
df		71		70
95% lower bound		-3.464015		-3.821732
p-value		0.551		0.155
Std. error		1.335711		1.113648
95% upper bound		1.862651		.6204699
t		-0.60		-1.44
Weight (lbs.)				
Coefficient				-.0067745
95% CI			-.0091011	-.0044479
df				70
95% lower bound				-.0091011
p-value				0.000
Std. error				.0011665
95% upper bound				-.0044479
t				-5.81
Intercept				
Coefficient		30.79176		41.84795
95% CI	27.46867	34.11485	37.15962	46.53628
df		71		70
95% lower bound		27.46867		37.15962
p-value		0.000		0.000
Std. error		1.666592		2.350704
95% upper bound		34.11485		46.53628
t		18.48		17.80

That is most of what we would want in a comparative regression table and a bit more. We probably do not want both the confidence interval and separately its upper and lower bound. Folks would

disagree about which among the standard error, t statistic, p-values, and confidence interval should be included.

More importantly, where are the overall model F statistic, the R^2, and the other model results? We saw earlier that these are in the `result` dimension, and we asked for everything in the `result` dimension.

This again comes down to how `collect layout` constructs the table by enumerating the levels in the specified dimensions. We discussed earlier that the row specification is interacted with the column specification. We specifically requested an interaction of `colname` and `result` on the rows. So, because `collect layout` enumerates all combinations of `cmdset`, `colname`, and `result`, it is always trying to find a unique value for a specific level of each of these dimensions.

If we want overall model results, the problem with the fully interacted enumeration is that it always includes a level for `colname`. Model results cannot be tagged with a `colname`. They are not associated with any variable or other parameter. We need to ask for results that do not include a `colname`. Easy enough; we never said that the dimensions in row or column specifications had to be interacted. They can also be stacked, one after the other. We can add the `result` dimension to our row specification again, but this time not interacting it with `colname`.

```
. collect layout (colname#result result) (cmdset)
```

We have added a whole new set of enumerations to our table. After enumerating all possible combinations of the levels of `colname`, `result`, and `cmdset`, `collect layout` will then enumerate all possible combinations of just `result` and `cmdset`.

Before we run that, let's put back our request for a subset of the levels of `result` when interacted with `colname`. We will leave all the model results, just to see what is there.

```
. collect layout (colname#result[_r_b _r_se _r_z _r_p] result) (cmdset)
```

Okay, we, the authors, tried that, and the result will not fit in the width of the page you are reading. So let's ask for only one of our regressions first, just so we can see what is there.

```
. collect layout (colname#result[_r_b _r_se _r_z _r_p] result) (cmdset[1])
Collection: default
      Rows: colname#result[_r_b _r_se _r_z _r_p] result
   Columns: cmdset[1]
   Table 1: 40 x 1
```

	1
Displacement (cu. in.)	
Coefficient	-.0469161
Std. error	.0066931
t	-7.01
p-value	0.000
Domestic	
Coefficient	0
Std. error	0
Foreign	
Coefficient	-.8006817
Std. error	1.335711
t	-0.60
p-value	0.551
Intercept	
Coefficient	30.79176
Std. error	1.666592
t	18.48
p-value	0.000
F statistic	35.56533
Number of observations	74
Command	regress
Command line as typed	regress mpg displacement i.foreign
Dependent variable	mpg
Model DF	2
Residual DF	71
Program used to implement estat	regress_estat
Log likelihood	-208.7139
Log likelihood, constant-only model	-234.3943
Predictions allowed by margins	XB default
Model	ols
Model sum of squares	1222.853
Program used to implement predict	regres_p
Command properties	b V
R-squared	.5004596
Adjusted R-squared	.4863881
Rank of VCE	3
RMSE	4.146281
Residual sum of squares	1220.607
Title of output	Linear regression
SE method	ols

Goodness, that even includes the command line as typed. For our comparison, let's request a subset of the model results by specifying specific levels of dimension `result`.

```
. collect layout (colname#result[_r_b _r_se _r_z _r_p] result[N F r2 ll])
> (cmdset)
Collection: default
      Rows: colname#result[_r_b _r_se _r_z _r_p] result[N F r2 ll]
   Columns: cmdset
   Table 1: 27 x 2
```

	1	2
Displacement (cu. in.)		
Coefficient	-.0469161	.0019286
Std. error	.0066931	.0100701
t	-7.01	0.19
p-value	0.000	0.849
Domestic		
Coefficient	0	0
Std. error	0	0
Foreign		
Coefficient	-.8006817	-1.600631
Std. error	1.335711	1.113648
t	-0.60	-1.44
p-value	0.551	0.155
Weight (lbs.)		
Coefficient		-.0067745
Std. error		.0011665
t		-5.81
p-value		0.000
Intercept		
Coefficient	30.79176	41.84795
Std. error	1.666592	2.350704
t	18.48	17.80
p-value	0.000	0.000
Number of observations	74	74
F statistic	35.56533	45.88031
R-squared	.5004596	.6628796
Log likelihood	-208.7139	-194.1637

There is a lot we could do to make this table prettier. You can learn about that by reading the examples in this manual. What we hope is that you are now more comfortable with how and why "you just use tags organized into the levels of dimensions to request tabular results."

Also see

[TABLES] **Intro 2** — A tour of concepts and commands

[TABLES] **collect layout** — Specify table layout for the current collection