Title

stata.com

Intro 2 — A tour of concepts and commands

Description Remarks and examples Also see

Description

There is one key concept on which the collection system is built—tags.

In this entry, we introduce tags and how they are created and used by the `collect` commands. Along the way, we will introduce several of the most important `collect` commands. Our focus here is on concepts and general features. We will not attempt to cover everything. See [TABLES] **Intro 3** for the quickest overview of the features.

We make no attempt in this entry to create pretty or interesting tables. Our sole purpose is to introduce concepts and commands.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Tags, dimensions, and levels

Your goal is to construct tables from the results of one or more commands. You need something to organize results from commands in such a way that you can conveniently place the results onto the rows and columns of tables. You would also like to control how everything looks, from the row and column headers to numeric formats, or even to the background color of an emphasized result. You do all that using the collection system, and the collection system needs to do lots of bookkeeping. The bookkeeping system for `collect` is tags.

We start by collecting results. Collecting results is as simple as placing the prefix `collect` in front of any command that returns results. Let's also place a `by` prefix in front of our command so we have results by each level of the `by` variables.

Introducing collect:

```
. use https://www.stata-press.com/data/r18/nhanes2l
(Second National Health and Nutrition Examination Survey)
. collect clear
. sort sex region
. collect: by sex region: summarize weight
```

```
-> sex = Male, region = NE
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
      weight |      1,018    78.15295    12.89267      47.17     129.84

-> sex = Male, region = MW
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
      weight |      1,310    78.24791    13.50132       41.5     139.03

-> sex = Male, region = S
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
      weight |      1,332     77.5923    14.27054      30.84     158.53

-> sex = Male, region = W
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
      weight |      1,255    77.98812     13.6871      44.11     175.88

-> sex = Female, region = NE
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
      weight |      1,078    65.50096     14.0839      39.12     148.21

-> sex = Female, region = MW
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
      weight |      1,464    66.50488     14.7564      34.93     159.44

-> sex = Female, region = S
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
      weight |      1,521    67.16907    15.19103      35.27     138.91

-> sex = Female, region = W
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
      weight |      1,373    66.11902    14.66786      36.06     134.61
```

So we have computed means, standard deviations, and the minimum and maximum of `weight` for each combination of the levels of variables `sex` and `region`. By placing the `collect:` prefix in front of the `by:` command, we have collected those results into the default collection. We `collect cleared` first to be sure we were starting clean and not adding to an existing collection.

For readers more familiar with pounds, these weights are in kilograms; you can double these numbers in your head. Or multiply by 2.2 to be more accurate.

What do we mean by "collected"? The results have been stored, but more importantly they have been tagged. Let's trim down all of that `by:` output and focus on the means. Here is a "picture" of what `collect:` has done.

```
================================       ==========================================
by sex region: summarize weight  -->                  Collection
================================       ==========================================
```

Or, more specifically,

```
================================       ==========================================
 Variable |   ...      Mean           value            tags
----------+--------------------       ------ -----------------------------------
   sex = Male, region = NE
   weight |   ...  78.15295 ...  -->   78.15 sex[Male]   region[NE] result[mean]
   sex = Male, region = MW
   weight |   ...  78.24791 ...  -->   78.25 sex[Male]   region[MW] result[mean]
   sex = Male, region = S
   weight |   ...   77.5923 ...  -->   77.59 sex[Male]   region[S]  result[mean]
   sex = Male, region = W
   weight |   ...  77.98812 ...  -->   77.99 sex[Male]   region[W]  result[mean]
   sex = Female, region = NE
   weight |   ...  65.50096 ...  -->   65.50 sex[Female] region[NE] result[mean]
   sex = Female, region = MW
   weight |   ...  66.50488 ...  -->   66.50 sex[Female] region[MW] result[mean]
   sex = Female, region = S
   weight |   ...  67.16907 ...  -->   67.17 sex[Female] region[S]  result[mean]
   sex = Female, region = W
   weight |   ...  66.11902 ...  -->   66.11 sex[Female] region[W]  result[mean]
================================       ==========================================
```

Consider the first mean, 78.15. In the collection, it is tagged with `sex[Male]`, `region[NE]`, `result[mean]`. The second mean is tagged with `sex[Male]`, `region[MW]`, `result[mean]`. So one of its tags is the same as the first value—both are tagged `sex[Male]`. The `region` tags differ across the two means—`region[MW]` and `region[NE]`. All the values are tagged with `result[mean]`.

Scanning the "picture", it is clear that each value is tagged with the levels of the `sex` and `region` variables from its `by` group. That seems sensible.

Each tag has two parts—*part1*[*part2*]. Having two parts lets us group related things using *part1*. Having two parts also lets us refer to all the tags with the same *part1* by just saying the name of *part1* and not having to enumerate all the names in *part2*.

In the collection system, we do not call them "part1" and "part2". We could, but eventually this entry would start to sound like a Dr. Seuss children's book. We call "part1" dimension, and we call "part2" level, or level within dimension, *dimension*[*level*].

Every tag always has this two-part structure.

In our collection, we have considered three dimensions—`sex`, `region`, and `result`. Dimension `sex` has two levels—`Male` and `Female`. Dimension `region` has four levels—`NE`, `MW`, `S`, and `W`.

We can specify all levels in the `sex` dimension by typing either `sex[Male] sex[Female]` or just `sex`.

Introducing collect layout

Let's take advantage of referring to groups of tags by just their dimension name and create our first table. The command for laying out tables is `collect layout`, and it wants us to specify what goes on the rows and columns of the table. We computed means across two categorical variables,

and `collect` tagged those means with the categories of those variables. Those tags seem like the natural things to put on the rows and columns of our table.

The basic syntax of `collect layout` is

`collect layout` (*row tags*) (*column tags*) (*table tags*)

We will specify all the `sex` tags for the rows and all the `region` tags for the columns. Recalling that the dimension names typed alone represent all the tags in the dimension, we type

```
. collect layout (sex) (region) (result[mean])

Collection: default
      Rows: sex
   Columns: region
    Tables: result[mean]
   Table 1: 2 x 4
```

	NE	MW	S	W
Male	78.15	78.25	77.59	77.99
Female	65.50	66.50	67.17	66.12

The row headers in the table result from enumerating all the tags in dimension `sex`—`Male` and `Female`. The column headers result from enumerating all the tags in the dimension `region`—`NE`, `MW`, `S`, and `W`. Each cell in the table is identified by the intersection of the levels of `sex` and `region` from the cell's row and column headers. So the first cell is identified by `sex[Male]` and `region[NE]`, and it is filled in with the value in the collection that has those two tags (78.15). Continuing down the first column, we see the cell at the bottom left of the table gets its tags from its row and column and is thus `sex[Female]` and `region[NE]`, which is 65.50 from the collection. And so on. That is how `collect layout` fills in a simple table like ours.

The only thing a bit surprising is that we specified something for the *table tags*, `result[mean]`, when we wanted only one table. We have not discussed it yet, but `summarize` stored multiple results, and the `collect` prefix collected all of them. In addition to the means, our collection contains the standard deviation, the minimum, the maximum, and several other results. So we needed to tell `collect layout` which statistic we wanted, and we did that by specifying a table tag. We wanted only one statistic, means, and only one table, so we specified only one tag—`result[mean]`.

We have been telling a little fib about the names of some dimension levels. The `by` variables `sex` and `region` are numeric variables in the dataset, and their values are labeled with the labels we see on the `by` results and in the table we produced—`Male`, `Female`, `NE`, `MW`, `S`, and `W`. To ease in mapping the results of our `by: summarize` command to the tags in the collection, we pretended that the levels of `sex` and `region` were the level labels. In truth, the collection mirrors the dataset. The levels of `sex` are actually numeric—`1` for `Male` and `2` for `Female`. The same is true for the levels of region—`1` for `NE`, `2` for `MW`, `3` for `S`, and `4` for `W`. The collection stores the labels for the levels separately.

We were not fibbing about `mean` in dimension `result`. `mean` really is the name of the level for the means. Dimension levels can be either numeric or string. If the string contains spaces, you must enclose it in quotes wherever it is used.

So to be more truthful, the collection looks more like

```
=========================================
              Collection
=========================================
value            tags
-----   -----------------------------------
 78.15  sex[1] region[1] result[mean]
 78.25  sex[1] region[2] result[mean]
 77.59  sex[1] region[3] result[mean]
 77.99  sex[1] region[4] result[mean]
 65.50  sex[2] region[1] result[mean]
 66.50  sex[2] region[2] result[mean]
 67.17  sex[2] region[3] result[mean]
 66.12  sex[2] region[4] result[mean]
 -----------------------------------------

 dimension  level   label
 ---------  -----  ------
 sex          1     Male
              2     Female
 region       1     NE
              2     MW
              3     S
              4     W
 result     mean    Mean
 =========================================
```

From here on, we will use the actual numeric levels created by `collect` for dimensions `sex` and `region`.

Introducing collect recode

As a sidebar, with a small collection like ours, we could have easily turned our fib into the truth. The command `collect recode` recodes dimension levels from one value to another. Were we to type

```
. collect recode sex 1=Male 2=Female
. collect recode region 1=NE 2=MW 3=S 4=W
```

then everything we said above would be true. And we could use terms like `sex[Female]` rather than `sex[2]` in everything we type below.

Using collect layout

You might be thinking that you can do everything we have done so far with the `table` command, and you are right. In fact, you could have created a collection that is very similar to the one we are working with by typing

```
. table (sex) (region), statistic(mean weight)
```

Let's start doing things that you cannot do with `table` directly.

By the way, the collection that `table` creates is so similar to the one we created with `collect: by:` that you could do everything we do below after either the `table` command above or the `collect: by:` command we started with. The main difference you would see is that `table` computed subtotals for `sex` and `region` and created levels for those totals in the `sex` and `region` dimensions. You can prevent that by adding the option `nototals`.

First, let's transpose our table by swapping where `sex` and `region` appear in the command.

```
. collect layout (region) (sex) (result[mean])

Collection: default
      Rows: region
   Columns: sex
    Tables: result[mean]
   Table 1: 4 x 2

─────────────────────
     │  Male Female
─────┼───────────────
NE   │ 78.15  65.50
MW   │ 78.25  66.50
S    │ 77.59  67.17
W    │ 77.99  66.12
─────────────────────
```

Wait! You say, "I could have done that with `table` by typing".

```
. table (region) (sex), statistic(mean weight)
```

That is not the same thing. `table` went back through the dataset, recomputed statistics, and then presented them in tabular form. If your dataset had 1 billion observations, that could take some time. We just told `collect layout` to show us the existing collection in a different way.

Let's go on.

Selecting specific levels of a dimension

We have been using dimensions `sex` and `region` to represent all the tags associated with their levels. That implies that we did not need to use all the levels of `sex` and `region` in our layout command. And, indeed, that is true. We could type just a few tags specifically, or even one.

```
. collect layout (region[1] region[3] region[4]) (sex[2]) (result[mean])

Collection: default
      Rows: region[1] region[3] region[4]
   Columns: sex[2]
    Tables: result[mean]
   Table 1: 3 x 1

──────────────────
     │ Female
─────┼────────────
NE   │   65.50
S    │   67.17
W    │   66.12
──────────────────
```

We explicitly typed out the list of `region` tags. There is a shorthand for specifying lists of levels within a dimension—type the list within the brackets. The following would have produced an identical table:

```
. collect layout (region[1 3 4]) (sex[2]) (result[mean])
```

Taken to extremes, `collect layout` is the way to pull a single value out of a collection.

```
. collect layout (region[3]) (sex[2]) (result[mean])

Collection: default
      Rows: region[3]
   Columns: sex[2]
    Tables: result[mean]
   Table 1: 1 x 1

┌──────────┐
│  Female  │
├──────────┼
S │   67.17 │
└──────────┘
```

What is in my collection?

We have been ignoring that `result` dimension. Let's rectify that.

Introducing collect levelsof

First, let's list the levels of `result`.

```
. collect levelsof result

Collection: default
 Dimension: result
    Levels: N Var max mean min sd sum sum_w
```

If you use `summarize` much, that list of levels may look familiar. Let's use that list to be a little more explicit about what values `collect` actually collects. It collects everything that is returned by your command in `e()` or `r()`. The final `summarize` from our `by` command is the last r-class command we have run. Here are the results returned by that `summarize`.

```
. return list

scalars:
                  r(sum) =  90781.40996932983
                  r(max) =  134.6100006103516
                  r(min) =  36.06000137329102
                   r(sd) =  14.66785984772278
                  r(Var) =  215.1461125124382
                 r(mean) =  66.11901672930068
                r(sum_w) =  1373
                    r(N) =  1373
```

The names of the `r()` results returned by `summarize` are a one-to-one match with the level names in dimension `result`. They are ordered differently because `collect` keeps the levels sorted alphabetically (with capitals first). Regardless, the names of the levels are exactly the names of the `r()` results, with "`r()`" stripped away. The same would be true if we collected results from a command that returns in `e()`. Every result is collected, and it is tagged with its `r()` or `e()` name. Well, almost every result; we will amend that in `collect get`, but you will not care.

As we saw earlier, every collected value has multiple tags, but one of them will always be its `result[`*name*`]`, where *name* is taken from its `e()` or `r()` name.

The simple list of levels from `collect levelsof` does not tell us much. We can learn a bit more about the levels by listing their labels.

Introducing collect label list

```
. collect label list result, all

  Collection: default
   Dimension: result
       Label: Result
Level labels:
           N  Number of observations
         Var  Variance
         max  Maximum
        mean  Mean
         min  Minimum
          sd  Std. dev.
         sum  Sum of variable
       sum_w  Sum of the weights
```

Now we are getting somewhere. Those results are everything that was reported on the `summarize` output plus a "`Sum of variable`", "`Sum of the weights`", and a "`Variance`".

Where do result labels come from?

Where did those labels come from? They are system default labels for collections. There is a default label for nearly every result returned in `r()` or `e()` by official commands.

Introducing collect label levels

It is easy for you to change a label. Perhaps you think "`Number of observations`" is too verbose, particularly if you want to make it a column in a table. Let's make it way shorter; lots of folks just go with "`N`".

```
. collect label levels result N "N", modify
```

Maybe we should also shorten the other two long labels.

```
. collect label levels result sum "Sum" sum_w "Sum wts.", modify
```

Introducing collect label save

Later, after you have made lots of label changes, you can save your preferred labels in a file. Type

```
. collect label save mylabels
```

where `mylabels` is whatever filename you prefer. Over time, you may override most of the default system labels.

Introducing collect label use

You can later apply those labels to a collection by typing

```
. collect label use mylabels
```

You do not have to worry if your collection does not contain some of the things you are labeling. The labels exist separately, and there is no harm in labeling things not in your collection. In fact, if those things are later created in your collection because you collect more results, they will get your labels automatically. So you can type `collect label use mylabels` when you first create a collection or right before you create a table; it makes no difference.

Now that we know what other results are tagged by dimension `result`, let's put some of those in a table. One possibility that comes to mind is to remove the shackles of `result[mean]` from our earlier layout command and ask for all levels of `result` as tables.

```
. collect layout (region) (sex) (result)

Collection: default
      Rows: region
   Columns: sex
    Tables: result
   Table 1: 4 x 2
   Table 2: 4 x 2
   Table 3: 4 x 2
   Table 4: 4 x 2
   Table 5: 4 x 2
   Table 6: 4 x 2
   Table 7: 4 x 2
   Table 8: 4 x 2

N
```

	Male	Female
NE	1018.00	1078.00
MW	1310.00	1464.00
S	1332.00	1521.00
W	1255.00	1373.00

```
Variance
```

	Male	Female
NE	166.22	198.36
MW	182.29	217.75
S	203.65	230.77
W	187.34	215.15

(*output omitted*)

```
Sum wts.
```

	Male	Female
NE	1018.00	1078.00
MW	1310.00	1464.00
S	1332.00	1521.00
W	1255.00	1373.00

Interactions in collect layout

Well, that was easy to type but not very interesting. For a table like this, it is time to learn about interactions.

First, let's consider our collection for a minute. We chose this particular problem earlier because it was two dimensional, just like many tables. We chose `region` for the rows and `sex` for the columns. The interaction of those two dimensions produces the cells in the table. By "interaction", we mean all combinations of the levels of `region` with the levels of `sex`. In one cell, you must be both male and in the Northeast. In another cell, you must be both female and in the South.

But wait. Our collection does not really have just two dimensions. That was an artifact of our considering only the mean. We have a whole other dimension—`result`. Our results really form a cube—`region X sex X result`. There is a value in every cell of that cube. Now you see the reason we call *part1* of our tags a dimension.

`collect layout` automatically interacts the row and column specifications. For our current example, each row represents a level of dimension `region`, and each column represents a level of `sex`. Each cell results from the interaction of the levels of its row and column. When we added the `result` dimension to create separate tables, each `sex`, `region`, and `result` triad represented one of the cells in one of the tables. Each cell was the result of a three-way interaction.

There is a term for interactions that you place on the rows of tables—"super rows". Likewise, tables can have super columns. If a table has either super rows or super columns, it is representing an underlying three-dimensional set of results. If it has both super rows and super columns, it is representing a four-dimensional set of results. You might have super-super rows or super-super columns. `collect` allows over 20 supers in each of the row, column, and table specifications; so you can represent up to silly-dimensional results.

Adding a super row or a super column is as easy as explicitly interacting two dimensions in the `collect layout` specification. You interact two dimensions by placing a `#` between them. Let's put our original row and column dimensions both onto the rows.

```
. collect layout (sex#region) (result[mean])
Collection: default
      Rows: sex#region
   Columns: result[mean]
   Table 1: 10 x 1
```

	Mean
Male	
NE	78.15
MW	78.25
S	77.59
W	77.99
Female	
NE	65.50
MW	66.50
S	67.17
W	66.12

Now the levels of dimension `sex` form super rows and the levels of `region` form rows within `sex`. These are the same results from our very first table, just organized differently.

We moved `result[mean]` to the column specification because there was no longer a reason to specify a tables dimension.

We could have specified a tables dimension and typed

```
. collect layout (sex#region) () (result[mean])
```

Note that an empty `()` is perfectly acceptable. It indicates that there are no tags for the columns.

We could even have pulled the interaction of dimension `result` into the rows specification and not specified any columns or tables.

```
. collect layout (sex#region#result[mean])
```

All of these commands produce a single column of results. Type them and see. The labels change a bit because `collect layout` tries to keep you informed of what you are seeing.

Now we are ready to put our three-dimensional data onto a table. Let's try `result` on the columns of the table.

```
. collect layout (sex#region) (result)

Collection: default
      Rows: sex#region
   Columns: result
   Table 1: 10 x 8
```

	N	Variance	Maximum	Mean	Minimum	Std. dev.	Sum	Sum wts.
Male								
NE	1018.00	166.22	129.84	78.15	47.17	12.89	79559.70	1018.00
MW	1310.00	182.29	139.03	78.25	41.50	13.50	1.0e+05	1310.00
S	1332.00	203.65	158.53	77.59	30.84	14.27	1.0e+05	1332.00
W	1255.00	187.34	175.88	77.99	44.11	13.69	97875.09	1255.00
Female								
NE	1078.00	198.36	148.21	65.50	39.12	14.08	70610.03	1078.00
MW	1464.00	217.75	159.44	66.50	34.93	14.76	97363.14	1464.00
S	1521.00	230.77	138.91	67.17	35.27	15.19	1.0e+05	1521.00
W	1373.00	215.15	134.61	66.12	36.06	14.67	90781.41	1373.00

We hope that is what you were expecting.

Introducing collect style cell

Some of the numbers are oddly formatted, for example, two decimal places on the observation count! This is a good time to admit that we cheated a bit at the outset. We changed the default formatting to get pretty numbers we could talk about. If you have been following along, you were already onto us because your tables showed more decimal places than ours.

Here is what we typed earlier but did not tell you about:

```
. collect style cell result, nformat(%8.2f)
```

Styles control literally everything about how a table looks. Without getting too much into styles right now, what our style command "said" was, "Set the numeric format for all results to be %8.2f." Let's set it back to its system default and redraw our table.

```
. collect style cell result, nformat(%9.0g)

. collect preview
```

	N	Variance	Maximum	Mean	Minimum	Std. dev.	Sum	Sum wts.
Male								
NE	1018	166.221	129.84	78.15295	47.17	12.89267	79559.7	1018
MW	1310	182.2857	139.03	78.24791	41.5	13.50132	102504.8	1310
S	1332	203.6484	158.53	77.5923	30.84	14.27054	103352.9	1332
W	1255	187.3368	175.88	77.98812	44.11	13.6871	97875.09	1255
Female								
NE	1078	198.3562	148.21	65.50096	39.12	14.0839	70610.03	1078
MW	1464	217.7513	159.44	66.50488	34.93	14.7564	97363.14	1464
S	1521	230.7675	138.91	67.16907	35.27	15.19103	102164.2	1521
W	1373	215.1461	134.61	66.11902	36.06	14.66786	90781.41	1373

Introducing collect preview

`collect preview`! That is a new command. We were not changing the layout, so there was no need to specify a new layout. We just asked `collect` to `preview` our existing layout using the style settings currently in effect.

Even so, "preview" seems an odd word. What we see in the Results window is often not our end goal. Often, we are creating a table to be exported to Microsoft Word, HTML, LaTeX, or some other format. Moreover, some of the styles we use cannot be shown in the Results window. So this is just a preview of what you might ultimately obtain when you export your results.

Note that `collect preview` does not display the report about the structure of the table that `collect layout` displays. `collect preview` provides cleaner output—just the table.

With the "new" numeric format, our table shows the numbers we should have been seeing all along.

Reordering columns

Continuing with `collect layout`, you can select the levels of dimension `result` you want, and in any order you want, perhaps,

```
. collect layout (sex#region) (result[mean sd min max N])

Collection: default
      Rows: sex#region
   Columns: result[mean sd min max N]
   Table 1: 10 x 5
```

	Mean	Std. dev.	Minimum	Maximum	N
Male					
NE	78.15295	12.89267	47.17	129.84	1018
MW	78.24791	13.50132	41.5	139.03	1310
S	77.5923	14.27054	30.84	158.53	1332
W	77.98812	13.6871	44.11	175.88	1255
Female					
NE	65.50096	14.0839	39.12	148.21	1078
MW	66.50488	14.7564	34.93	159.44	1464
S	67.16907	15.19103	35.27	138.91	1521
W	66.11902	14.66786	36.06	134.61	1373

Change the order of the levels specified to `collect layout`, and you change the order of the columns on the table.

```
. collect layout (sex#region) (result[N min mean max sd N])
```

You can even repeat levels.

```
. collect layout (sex#region) (result[max max max max max max])
```

(Tabulus maximus?)

Type either command and see.

We could even present just the counts as a frequency cross-tabulation. Feel free to type

```
. collect layout (region) (sex) (result[N])
```

You can also organize the rows and columns differently. You might type any of these layouts or try some of your choosing.

```
. collect layout (sex#result[mean N]) (region)
. collect layout (region#result[mean min max]) (sex)
. collect layout (region#result[mean min max]) (sex)
```

More layout

Our `result` options increase dramatically if we collect `summarize, detail`.

```
. collect clear
. collect: by sex region: summarize weight, detail
```

Let's see what our `result` choices are now.

```
. collect label list result, all

  Collection: default
   Dimension: result
       Label: Result
Level labels:
           N  Number of observations
         Var  Variance
    kurtosis  Kurtosis
         max  Maximum
        mean  Mean
         min  Minimum
          p1  1st percentile
         p10  10th percentile
         p25  25th percentile
          p5  5th percentile
         p50  50th percentile
         p75  75th percentile
         p90  90th percentile
         p95  95th percentile
         p99  99th percentile
          sd  Std. dev.
    skewness  Skewness
         sum  Sum of variable
       sum_w  Sum of the weights
```

We could create a table of whatever percentile distributions interest us, perhaps the quartiles,

```
. collect layout (sex#region) (result[min p25 p50 p75 max])
```

or a finer grain,

```
. collect layout (sex#region) (result[p5 p10 p25 p50 p75 p90 p95])
```

The authors typed that and found that the labels on the percentiles are far too long. So let's shorten them.

```
. collect label levels result p5 "5th" p10 "10th" p25 "25th"
> p50 "50th" p75 "75th" p90 "90th" p95 "95th", modify
. collect preview
```

	5th	10th	25th	50th	75th	90th	95th
Male							
NE	59.42	62.82	69.63	76.89	85.62	95.82	101.61
MW	58.97	62.655	69.17	77.055	85.16	95.2	102.97
S	57.49	60.56	67.19	76.43	85.84	95.03	103.19
W	57.95	62.03	68.49	76.77	85.96	95.03	101.49
Female							
NE	47.51	50.24	55.45	62.88	72.24	84.48	91.74
MW	48.31	50.69	56.59	63.62	73.425	85.39	94.46
S	47.74	50.8	56.36	64.41	75.3	86.98	95.82
W	47.85	50.69	56.25	63.39	72.92	85.96	95.6

We would like to have that %8.2f format back about now.

If you are a fan of third and fourth moments, you could assess and compare all the distributions using skewness and kurtosis.

```
. collect layout (sex#region) (result[mean sd skewness kurtosis])
Collection: default
      Rows: sex#region
   Columns: result[mean sd skewness kurtosis]
   Table 1: 10 x 4
```

	Mean	Std. dev.	Skewness	Kurtosis
Male				
NE	78.15295	12.89267	.5601461	3.705207
MW	78.24791	13.50132	.7798423	4.354643
S	77.5923	14.27054	.6834379	4.384609
W	77.98812	13.6871	.8854262	5.942613
Female				
NE	65.50096	14.0839	1.154802	5.090129
MW	66.50488	14.7564	1.327805	6.098792
S	67.16907	15.19103	1.100521	4.796148
W	66.11902	14.66786	1.231803	5.036233

Introducing collect style autolevels

There is an alternative way to specify the levels on dimension `result` that we used in the last two tables. Instead of specifying them directly in the `collect layout` command, we can preset levels to be used when a dimension name is specified without levels. If you type

```
. collect style autolevels result mean sd skewness kurtosis
```

then whenever `result` appears alone in a `collect layout` command, only levels `mean`, `sd`, `skewness`, and `kurtosis` will be enumerated. We call these levels "automatic levels". It is just as though you typed `result[mean sd skewness kurtosis]`.

So typing

```
. collect style autolevels result mean sd skewness kurtosis
. collect layout (sex#region) (result)
```

produces exactly the same result as

```
. collect layout (sex#region) (result[mean sd skewness kurtosis])
```

Every time you type `collect style autolevels` on the same dimension, it adds whatever levels you type to any existing autolevels for the dimension. So typing

```
. collect style autolevels result p5 p10 p25
. collect style autolevels result p50 p75 p90 p95
```

is equivalent to typing

```
. collect style autolevels result p5 p10 p25 p50 p75 p90 p95
```

Typing

```
. collect style autolevels result, clear
. collect style autolevels result p5 p10 p25 p50 p75 p90 p95
. collect layout (sex#region) (result)
```

produces exactly the same table we created earlier when we typed

```
. collect layout (sex#region) (result[p5 p10 p25 p50 p75 p90 p95])
```

`collect style autolevels` can be particularly convenient when you are exploring several table layouts and you want to use the same `result` levels on all the tables. Or, for that matter, the same levels of any dimension used in the table.

What is in my collection, regression edition

We have already seen one unusual dimension—`result`. The dimensions representing categorical variables, `sex` and `region`, are easy to understand. Anyone who has created a cross-tabulation has used categorical variables as the rows and columns of a table. Dimension `result` was a little bit different. It is just a place where we are keeping related identifiers (levels)—in this case, all the names of results returned in `r()` and `e()`.

We warn you, `collect` uses other unusual dimensions. And it uses a few unusual levels.

Consider the output from a regression.

```
. regress bpsystol age weight i.sex

      Source |       SS           df       MS      Number of obs   =    10,351
-------------+----------------------------------   F(3, 10347)     =   1501.75
       Model |  1709209.9         3  569736.633   Prob > F        =    0.0000
    Residual |  3925460.13    10,347  379.381476   R-squared       =    0.3033
-------------+----------------------------------   Adj R-squared   =    0.3031
       Total |  5634670.03    10,350  544.412563   Root MSE        =    19.478

------------------------------------------------------------------------------
    bpsystol | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
         age |   .6374325   .0111334    57.25   0.000     .6156088    .6592562
      weight |   .4170339    .013474    30.95   0.000     .3906221    .4434456
             |
         sex |
     Female  |   .8244702   .4140342     1.99   0.046     .0128832    1.636057
       _cons |   70.13615   1.187299    59.07   0.000     67.80881    72.46348
------------------------------------------------------------------------------
```

The result levels _r_b, _r_se, . . .

The results are already laid out as a table with the coefficient names on the rows and the coefficient statistics on the columns. Neither the rows nor the columns fit into the dimension and level names we have been using.

Let's consider the columns first—the coefficient statistics. We certainly have an appropriate dimension where we can place these: the `result` dimension. What is tricky is how to name their levels. The coefficients themselves are saved as a row vector named `e(b)`, so we could name their level `b` in `result`, as we have all the other stored results. Spoiler alert, we do not.

The problem is we do not store vectors for the standard error, the t statistic, the p-value, or the confidence interval. These are stored in hidden places or can be derived from other results. You do not care about that; you want to use what you see in the `regress` results in your own tables. So we gave these results special level names—`_r_b` for the regression coefficients, `_r_se` for the standard errors, and so on. Here is the full list of special level names for regression and regressionlike results:

Identifier	Result
`_r_b`	coefficients or transformed coefficients reported by *command*
`_r_se`	standard errors of `_r_b`
`_r_z`	test statistics for `_r_b`
`_r_z_abs`	absolute values of `_r_z`
`_r_df`	degrees of freedom for `_r_b`
`_r_p`	p-values for `_r_b`
`_r_lb`	lower bounds of confidence intervals for `_r_b`
`_r_ub`	upper bounds of confidence intervals for `_r_b`
`_r_ci`	confidence intervals for `_r_b`
`_r_cri`	credible interval (CrI) of Bayesian estimates
`_r_crlb`	lower bound of CrI of Bayesian estimates
`_r_crub`	upper bound of CrI of Bayesian estimates

We admit the `_r_` is a bit much to type and requires explanation. There is a reason for the leading underscore. `collect` will collect all the results from `e()` and `r()` for any official command or from any command written by you or by other users. Those results could have any valid name. By convention, we have told users that anything with a leading underscore is reserved for official names. There is also the precedence of `_b[`*coefname*`]` and _se[*coefname*] being supported in expressions to retrieve coefficients and their standard errors.

As an aside, all the `_r_` names you see above are now supported in expressions. After the regression command above, you could type

```
. display _r_b[age] / _r_se[age]
```

to compute the t statistic by hand and display it.

There is also a reason we chose `r`. Consider the `logistic` regression

```
. logistic highbp age weight i.sex
Logistic regression                               Number of obs =  10,351
                                                  LR chi2(3)    = 2326.44
                                                  Prob > chi2   =  0.0000
Log likelihood = -5887.5446                       Pseudo R2     =  0.1650

------------------------------------------------------------------------------
      highbp | Odds ratio   Std. err.      z    P>|z|     [95% conf. interval]
-------------+----------------------------------------------------------------
         age |   1.052054   .0014852    35.95   0.000     1.049147    1.054969
      weight |   1.044683    .001759    25.96   0.000     1.041242    1.048137
             |
         sex |
     Female  |   1.036659   .0498306     0.75   0.454     .9434528    1.139074
       _cons |    .002525   .0004077   -37.05   0.000     .0018401     .003465
------------------------------------------------------------------------------
Note: _cons estimates baseline odds.
```

The default "coefficients" displayed after `logistic` are the odds ratios, not the raw coefficients. You can see the raw coefficients instead by adding the option `coef`. The "`r`" in `_r_b` stands for "reported". After our logistic regression, the odds ratios, not the raw coefficients, are collected. In this case, `result[_r_b]` tags the odds ratios. If we add `coef` to our command, or even if we replay the results with the option `coef`,

```
. logistic, coef
```

the raw coefficients are collected. `_r_b` then stands for the raw coefficient estimates. You can collect whichever transformation you prefer. When transformations are available, whatever you are reporting is what is collected. Type two `collect` commands if you want to collect both transformed and raw coefficients.

There are quite a few commands that report transformations of their coefficients—incidence rate ratios for `poisson`, hazard ratios for `stcox`, standardized coefficients for `sem`, and several others. Many of these estimators also have panel-data and multilevel commands.

The `_r_` results are collected after all regression and regression-like commands. The regression-like commands include `mean`, `proportion`, `ratio`, `bayesmh`, `margins`, `contrast`, and others.

The colname dimension

There is still the issue of what dimension name we should use for the rows of a regression table. They look like variables, so why not `variable`? Because those rows can contain lots of things that are not variables: for example, the ancillary parameters for variance on many regression commands, parameters on latent variables in `sem` and `gsem`, contrasts or expressions in `margins`, and so on.

`collect` uses the dimension `colname` to hold these variable/parameter/estimate tags. There truly is no good meaningful name for all the things this dimension can hold.

There is also a technical reason for using `colname`. The `_r_` results are all related to `e(b)`, and `e(b)` is a row vector. Let's list `e(b)` for our logistic regression.

```
. matrix list e(b)
e(b)[1,5]
       highbp:    highbp:    highbp:    highbp:    highbp:
                                 1b.         2.
          age     weight        sex        sex      _cons
y1  .05074447  .04371396          0  .03600346  -5.981495
```

Those labels immediately above the matrix values are the column names for the matrix. All matrices in Stata have row and column names. That way, you can refer to the rows and columns by name as well as by index number. The matrix's column names collectively are called its `colname`. We can use a macro function to display just the column names.

```
. display "`: colname e(b)'"
age weight 1b.sex 2.sex _cons
```

Considering just `e(b)` (`_r_b`), `collect` is really collecting a matrix. To identify a cell in a matrix `collect` not only needs a tag for the whole matrix, `result[_r_b]`, but also needs tags for the specific row and specific column that identify a particular cell. The column tags are placed in dimension `colname` because that is what Stata calls the column names of a matrix. For our logistic model, the `colname` tags associated with all the `_r_` results are `colname[age]`, `colname[weight]`, `colname[1.sex]`, `colname[2.sex]`, and `colname[_cons]`.

If you guessed from the matrix we listed that there would be a `rowname` tag for the `_r_b` "matrix" that we collected, you would be right. That tag is `rowname[y]`. You won't use the `rowname` dimension nearly so often as you will use the `colname` dimension.

Labels on levels of dimension colname

There is something else special about `colname`. We discussed earlier that the levels of the `result` dimension are labeled using a set of system default labels. `collect` can also automatically label most levels of `colname`. That is because most levels of `colname` are variable names. If a variable is labeled, `collect` picks up that label and uses it to label the level. What is more, if a level represents a factor variable, such as `2.sex`, then `collect` labels that level of the factor variable with the appropriate value label from the dataset. It sounds complicated, but it is really just doing what you want. When we type

```
. quietly collect: mean weight, over(sex)
. collect style autolevels result _r_b _r_se _r_ci
. collect layout (colname) (result)
Collection: default
      Rows: colname
   Columns: result
   Table 1: 2 x 3
```

	Coefficient	Std. error	95% CI	
Weight (kg) @ Male	77.98423	.1945289	77.60292	78.36555
Weight (kg) @ Female	66.39418	.1998523	66.00243	66.78593

we see "`Male`" and "`Female`" as part of our row headers, not "`1`" and "`2`".

Note too that we just used some of the `_r_` levels of result and that we used dimension `colname` too. No need for fanfare. They are just other levels and dimensions that we can use to lay out our tables.

`colname` is not the only dimension that picks up labels from variables. Dimensions `rowname`, `coleq`, `roweq`, `var`, and `across` also fetch variable labels for the levels and factor-variable levels whenever they can.

It turns out the `_r_` levels and the `colname` dimension are not truly unusual. They work just the way any other levels or dimensions work. Their names are just arbitrary.

If you are not liking the row headers in the table above, you can change them. See `collect style row`.

collect layout with regression results

We claimed this subsection was about regression collections, so we should at least create a basic table of regression results from our first regression. First, we type

```
. collect clear
. collect: regress bpsystol age weight i.sex
```

Then, we type

```
. collect style autolevels result _r_b _r_se _r_z _r_p
. collect layout (colname) (result)
Collection: default
      Rows: colname
   Columns: result
   Table 1: 5 x 4
```

	Coefficient	Std. error	t	p-value
Age (years)	.6374325	.0111334	57.25	0.000
Weight (kg)	.4170339	.013474	30.95	0.000
Male	0	0		
Female	.8244702	.4140342	1.99	0.046
Intercept	70.13615	1.187299	59.07	0.000

We used `autolevels` to specify the automatic levels for `result`. That looks a lot like the regression output, except we did not ask for the confidence intervals, there is less column spacing, and this table uses labels rather than variable names on the row headers.

Introducing collect style showbase

There is a lot we could do to make this table prettier, but let's at least get rid of the row for `Male`. `Male` is the base level for the factor variable `i.sex` and we do not need to see its zero coefficient. To turn off displaying base levels for factor variables, we type

```
. collect style showbase off
```

Recall that we do not have to respecify our layout just to see the effect of style changes. We just type

```
. collect preview
```

	Coefficient	Std. error	t	p-value
Age (years)	.6374325	.0111334	57.25	0.000
Weight (kg)	.4170339	.013474	30.95	0.000
Female	.8244702	.4140342	1.99	0.046
Intercept	70.13615	1.187299	59.07	0.000

The base level is gone.

That is all we are going to style on this table. We will have much more to say about styles in section *Let's talk styles*.

At this point, it should come as no surprise that we can transpose the table by swapping the position of `colname` and `result` in our layout.

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

Collection: default
      Rows: result
   Columns: colname
   Table 1: 4 x 4
```

	Age (years)	Weight (kg)	Sex Female	Intercept
Coefficient	.6374325	.4170339	.8244702	70.13615
Std. error	.0111334	.013474	.4140342	1.187299
t	57.25	30.95	1.99	59.07
p-value	0.000	0.000	0.046	0.000

Let's clear the automatic levels so they does not surprise us later.

```
. collect style autolevels result, clear
```

Okay, it did bite the authors when they were writing this entry, and we do not want you to be surprised in the same way. It is pretty easy to convince yourself that collections are broken when you have an `autolevels` set that is at odds with levels you are trying to report.

Tables of model statistics

Before we leave this simple regression, let's look at one more thing. You may think that the regression coefficients are the only "tabular" results we have collected. But there is another set of results lurking in our collection, the model-level statistics. They are all about this one model, so collectively they are a set of one-dimensional results. Even so, a one-dimensional table is still a table.

We can also tell that the model statistics have been collected by listing the labels of dimension `result`.

```
. collect label list result

  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
    _r_z_abs  |t|
        beta  Standardized coefficient
         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
```

It takes a bit of scanning, but about midway down we see the `Model DF`, the `Residual DF`, and the `Log likelihood`. A bit farther down, we see the `R-squared`, the `Adjusted R-squared`, and the `RMSE`.

Do not be distracted by the `__LEVEL__%`; that is just the way labels obtain the confidence level that can be specified using the `level()` option of regression commands.

Previously, we pulled out the coefficient statistics by interacting dimensions `result` and `colname`. How do we ask for just model-level results? They are a one-way table (listing) of results, so we do not need to specify anything for our columns. We just ask for dimension `result` on the rows.

```
. collect layout (result)

Collection: default
      Rows: result
   Table 1: 22 x 1
```

F statistic	1501.751
Number of observations	10351
Command	regress
Command line as typed	regress bpsystol age weight i.sex
Dependent variable	bpsystol
Model DF	3
Residual DF	10347
Program used to implement estat	regress_estat
Log likelihood	-45420.36
Log likelihood, constant-only model	-47291.07
Predictions allowed by margins	XB default
Model	ols
Model sum of squares	1709210
Program used to implement predict	regres_p
Command properties	b V
R-squared	.3033381
Adjusted R-squared	.3031361
Rank of VCE	4
RMSE	19.47772
Residual sum of squares	3925460
Title of output	Linear regression
SE method	ols

Well, we certainly have our model statistics, but we have a lot of other "junk" too—the `Dependent variable`, a flag for `Predictions allowed by margins`, the `Rank of VCE`, and even the `Program used to implement predict` and the `Command line as typed`. We are going to have to be specific with `collect layout` about the levels of `result` we want.

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

Collection: default
      Rows: result[N r2 rmse df_m df_r F]
   Table 1: 6 x 1
```

Number of observations	10351
R-squared	.3033381
RMSE	19.47772
Model DF	3
Residual DF	10347
F statistic	1501.751

In explaining how we ask for the model statistics compared with how we ask for the coefficient statistics, we said, "They are a one-way table (listing) of results, so we do not need to specify anything for our columns." That is true, but it is also a pretty fast explanation. If it seems logical to you, you are good to go. If you would like to understand more fully why it is true, see section *How collect layout processes tag specifications* in [TABLES] **Collection principles**.

What is in my collection, multiple-equation models (dimension coleq)

Another "unusual" dimension that is useful for multivariate models is `coleq`. Let's collect the results from a simple multivariate regression.

```
. collect clear
. collect:  mvreg bpsystol bpdiast = age weight
Equation             Obs   Parms        RMSE    "R-sq"          F      P>F
----------------------------------------------------------------------------
bpsystol          10,351       3    19.48051    0.3031       2250   0.0000
bpdiast           10,351       3    11.51474    0.2067   1348.469   0.0000

------------------------------------------------------------------------------
             | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
bpsystol     |
         age |   .6379892   .0111315    57.31   0.000     .6161692    .6598091
      weight |   .4069041   .0124786    32.61   0.000     .3824435    .4313646
       _cons |   71.27096   1.041742    68.42   0.000     69.22894    73.31297
-------------+----------------------------------------------------------------
bpdiast      |
         age |    .187733   .0065797    28.53   0.000     .1748355    .2006306
      weight |   .3116502    .007376    42.25   0.000     .2971918    .3261086
       _cons |   50.37585    .615764    81.81   0.000     49.16884    51.58287
------------------------------------------------------------------------------
```

What is new about this regression is that it has multiple equations—one for `bpsystol` and one for `bpdiast`. It is sensible to tag each equation in the model and to put those tags into a dimension where they can be referenced together. That is just what `collect` does.

What does it name that dimension? Let's look at the `e(b)` matrix again.

```
. matrix list e(b)
e(b)[1,6]
     bpsystol:  bpsystol:  bpsystol:   bpdiast:   bpdiast:   bpdiast:
          age     weight      _cons        age     weight      _cons
y1  .63798917  .40690407  71.270956  .18773302  .31165024  50.375852
```

We see that there are `colnames` on this matrix, as there were on the simple regression. But we also see `bpsystol:` and `bpdiast:` above the `colnames`. Those are the dependent variables of our equation, and they also label the columns of the matrix. Collectively, we call `bpsystol` and `bpdiast` the matrix's `coleqs`, and there are matrix commands for setting and fetching the `coleq`. So `coleq` is the name `collect` gives to the dimension that holds the tags for the equations. In our model, the levels of those tags are the dependent variable names—`bpsystol` and `bpdiast`. Let's confirm

```
. collect label list coleq, all
  Collection: default
   Dimension: coleq
       Label: Depvars, parameters, and column equations
Level labels:
     bpdiast  Diastolic blood pressure
    bpsystol  Systolic blood pressure
```

Indeed `coleq` is a dimension. It has its own nice, long label—`Depvars, parameters, and column equations`. Its levels are indeed the dependent variable names from our multivariate regression—`bpdiast` and `bpsystol`. And those dimensions have their own nice, long labels—`Diastolic blood pressure` and `Systolic blood pressure`.

`collect label list` can tell us a lot about what is in a dimension, how we might use it in a layout, and whether we are likely to want to change its labels for our table.

We clearly cannot use our univariate regression layout specification.

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

Every cell in that table would have two values, one for the `bpdiast` dependent variable and one for the `bpsystol` dependent variable. That specification does not uniquely identify the cells in the table. We need to add dimension `coleq`. Let's try it in the tables specification first.

```
. collect style autolevels result _r_b _r_ci _r_se _r_z _r_p
. collect layout (colname) (result) (coleq)
Collection: default
      Rows: colname
   Columns: result
    Tables: coleq
   Table 1: 3 x 5
   Table 2: 3 x 5
Systolic blood pressure
```

	Coefficient	95% CI		Std. error	t	p-value
Age (years)	.6379892	.6161692	.6598091	.0111315	57.31	0.000
Weight (kg)	.4069041	.3824435	.4313646	.0124786	32.61	0.000
Intercept	71.27096	69.22894	73.31297	1.041742	68.42	0.000

```
Diastolic blood pressure
```

	Coefficient	95% CI		Std. error	t	p-value
Age (years)	.187733	.1748355	.2006306	.0065797	28.53	0.000
Weight (kg)	.3116502	.2971918	.3261086	.007376	42.25	0.000
Intercept	50.37585	49.16884	51.58287	.615764	81.81	0.000

We have presented our regression results in two tables.

That is not the best arrangement if we want to compare across the two regressions. Let's shuffle the equations onto the columns and put both the `colnames` and the `result` dimensions on the rows.

```
. collect layout (colname#result) (coleq)
Collection: default
      Rows: colname#result
   Columns: coleq
   Table 1: 18 x 2
```

	Systolic blood pressure		Diastolic blood pressure	
Age (years)				
Coefficient		.6379892		.187733
95% CI	.6161692	.6598091	.1748355	.2006306
Std. error		.0111315		.0065797
t		57.31		28.53
p-value		0.000		0.000
Weight (kg)				
Coefficient		.4069041		.3116502
95% CI	.3824435	.4313646	.2971918	.3261086
Std. error		.0124786		.007376
t		32.61		42.25
p-value		0.000		0.000
Intercept				
Coefficient		71.27096		50.37585
95% CI	69.22894	73.31297	49.16884	51.58287
Std. error		1.041742		.615764
t		68.42		81.81
p-value		0.000		0.000

Now it is easy to compare the regression coefficients and their statistics across dependent variables. Again, there is a lot we could do to make this table prettier. The justification makes the CIs jut out. As we predicted, the labels on `bpsystol` and `bpdiast` are too long for column headers. There are too many digits in the results. And more. We will address those types of concerns in *Let's talk styles*.

What is in my collection, collecting results from multiple commands (dimension cmdset)

We have been collecting results from a single command. It is just as easy to collect and tabulate results from several commands.

Let's collect results from two regressions.

```
. collect clear
. collect: regress bpsystol age weight
. collect: regress bpsystol age weight i.hlthstat
```

In the second regression, we added a factor variable that records self-reported health status.

With two regressions in our collection, we have two coefficients for `age` and `weight`. We have two of every statistic associated with those coefficients. That is painfully obvious, but important when specifying a layout. Because we have two of nearly everything, we need another dimension to tell the coefficients in the regression apart.

If only we had a dimension that identified the specific commands from which we collected results. We do, dimension `cmdset`. Let's look at its levels.

```
. collect label list cmdset, all

  Collection: default
   Dimension: cmdset
       Label: Command results index
Level labels:
           1
           2
```

Well, that is minimalist. The levels are `1` and `2` and they are unlabeled. Regardless, `cmdset` is a counter (or index) for each command from which we collected results. That is enough. Let's put that on the columns and put both the `colname` and `result` dimensions on the rows. To keep things short, let's just show the coefficients and their standard errors.

```
. collect style autolevels result _r_b _r_se

. collect layout (colname#result) (cmdset)

Collection: default
      Rows: colname#result
   Columns: cmdset
   Table 1: 24 x 2
```

	1	2
Age (years)		
Coefficient	.6379892	.6071483
Std. error	.0111315	.0119737
Weight (kg)		
Coefficient	.4069041	.4039598
Std. error	.0124786	.012471
Excellent		
Coefficient		0
Std. error		0
Very good		
Coefficient		.715111
Std. error		.5519263
Good		
Coefficient		2.233169
Std. error		.5453581
Fair		
Coefficient		4.133798
Std. error		.6492333
Poor		
Coefficient		3.549244
Std. error		.8558511
Intercept		
Coefficient	71.27096	71.22963
Std. error	1.041742	1.073791

And we need not stop there. We can add the results of as many commands as we like to a collection. Let's add a third regression with one more covariate.

```
. collect: regress bpsystol age weight i.hlthstat i.sex
```

To see those results on our table, we do not have to respecify our layout. We still want the commands on the columns. We have just added one more command. All we need to do is repreview the table.

```
. collect preview
```

	1	2	3
Age (years)			
Coefficient	.6379892	.6071483	.6070032
Std. error	.0111315	.0119737	.011973
Weight (kg)			
Coefficient	.4069041	.4039598	.4122565
Std. error	.0124786	.012471	.0134793
Excellent			
Coefficient		0	0
Std. error		0	0
Very good			
Coefficient		.715111	.6759903
Std. error		.5519263	.5524101
Good			
Coefficient		2.233169	2.184542
Std. error		.5453581	.5461395
Fair			
Coefficient		4.133798	4.062105
Std. error		.6492333	.6506867
Poor			
Coefficient		3.549244	3.537842
Std. error		.8558511	.8558125
Male			
Coefficient			0
Std. error			0
Female			
Coefficient			.6725152
Std. error			.4148375
Intercept			
Coefficient	71.27096	71.22963	70.32292
Std. error	1.041742	1.073791	1.210646

Just what we expected.

We could make this table prettier; see section *Let's talk styles*.

Let's at least get rid of the base levels of the factor variables and make the column headers a bit more informative.

```
. collect style showbase off
. collect label levels cmdset 1 "Base" 2 "Partial" 3 "Full"
. collect preview
```

	Base	Partial	Full
Age (years)			
Coefficient	.6379892	.6071483	.6070032
Std. error	.0111315	.0119737	.011973
Weight (kg)			
Coefficient	.4069041	.4039598	.4122565
Std. error	.0124786	.012471	.0134793
Very good			
Coefficient		.715111	.6759903
Std. error		.5519263	.5524101
Good			
Coefficient		2.233169	2.184542
Std. error		.5453581	.5461395
Fair			
Coefficient		4.133798	4.062105
Std. error		.6492333	.6506867
Poor			
Coefficient		3.549244	3.537842
Std. error		.8558511	.8558125
Female			
Coefficient			.6725152
Std. error			.4148375
Intercept			
Coefficient	71.27096	71.22963	70.32292
Std. error	1.041742	1.073791	1.210646

You cannot only collect from multiple commands but also collect from multiple sets of related commands. In the current example, we could have collected results from `test` commands for the additional covariates in the `Partial` and `Full` models. Or we could have collected the results of `lrtest` for the same purpose. Or we could have collected the results of `margins` commands that might have estimated the effect of dropping weight by 10%. Any or all of these results could have been collected and added below the coefficients in the table above. For an example, see [TABLES] **Example 6**.

Seeing what is my collection

We have been pulling dimension names out of thin air and using them. Let's do more. You can ask your collection about its dimensions at any time.

Introducing collect dims

```
. collect dims

Collection dimensions
Collection: default
──────────────────────────────────────────
                   Dimension   No. levels
──────────────────────────────────────────
Layout, style, header, label
                      cmdset   3
                       coleq   1
                     colname   10
           colname_remainder   1
                    hlthstat   5
                program_class   1
                      result   32
                 result_type   3
                     rowname   1
                         sex   2
Style only
                 border_block   4
                    cell_type   4
──────────────────────────────────────────
```

We read from the output that the current collection is the `default` collection. And we see a list of dimensions in groups.

Header `Layout, style, header, label` is telling you that you can do anything in the collection system with the dimensions in that group. You can lay out tables using `collect layout`. You can set cell styles on specific dimensions and levels using `collect style cell`. (Cell styles are all the styles for how things look—bolding, numeric formats, color, etc.) You can set whether the headers show labels, names or nothing for dimensions, or levels of dimensions, using `collect style header`. You can set the content of the labels used in the row and column headers using `collect label`.

The second grouping reads `Style only`. The only thing you can do with these dimensions and their levels is set cell styles.

A third grouping, not shown here but appearing between the above two, reads `Header, label`. You can do only two things with the dimensions in this group. You can set whether labels or names are shown in the headers, and you can change the content of the labels used in the headers. This group is populated by factor variables found in dimension `coleq`, `roweq`, or `rowname` but not `colname` or `var`.

It is not a syntax error to use any of these dimensions on one of the commands that are not in its usage group. Style and label commands are always allowed so long as their syntax is legal. The dimensions and levels that they reference do not need to exist in the current collection.

Let's return to the output of `collect dims`. In the first grouping of dimensions, we immediately recognize `cmdset`, `colname`, `coleq`, and `result`. They need no further explanation. That leaves three dimensions in the first group that we do not recognize—`colname_remainder`, `program_class`, and `result_type`. Let's list their levels and labels to search for clues.

First, colname_remainder,

```
. collect label list colname_remainder, all

  Collection: default
   Dimension: colname_remainder
       Label: Covariate names with factors removed
Level labels:
       _cons
```

colname_remainder is not interesting in this example. This dimension is created when collect augments the tags on a result with the factor variables from dimensions colname and var already in the tag. colname_remainder is the remaining (nonfactor) elements of interactions or _cons when the colname level is a single factor variable. This dimension might be necessary to help uniquely match items when you specify factor variables directly in collect layout instead of using them as levels within dimension colname or var.

Second, program_class,

```
. collect label list program_class, all

  Collection: default
   Dimension: program_class
       Label: Result program class
Level labels:
      eclass
```

Well, that could not be more boring. The single, unlabeled level is eclass. We collected results from two commands, two regress commands, and regress returns only results in e(). Results returned in e() are called e-class results, ergo, eclass. Had we also collected results from summarize, or even margins, then we would see a second level here—rclass.

We cannot think of a reason to use dimension program_class in the collect system. You could set the background to red for results returned by e-class commands and set the background to blue for results returned by r-class commands. We do not know why you would, but you could. Perhaps you are writing Stata documentation and want to emphasize where the results came from.

Third, result_type,

```
. collect label list result_type, all

  Collection: default
   Dimension: result_type
       Label: Result type
Level labels:
       macro  Macro
      matrix  Matrix
      scalar  Scalar
```

The levels are macro, matrix, and scalar. Those are the types of results that can be returned in e() or r(). Again, not something you would use often in specifying a layout or styling cells. But you could. If you added the interaction #result_type[scalar] to any term in the row, column, or table specification in collect layout, you would limit the table to include only scalar results.

Factor variables in regressions and other commands

In the first group, we see the two dimensions, hlthstat and sex. Those are the two factor variables from our regressions. collect creates dimensions for factor variables from regressions and from other commands that accept factor variables in the *varlist*.

These dimensions are similar to the dimensions that are named after the `by` variables in our very first example in this entry. All of these dimensions can be used to specify rows and columns in `collect layout`. Even dimensions `hlthstat` and `sex` can be used; however, they are usually specified in the `colname` dimension.

One way to tag regression results is `colname[hlthstat]`. So we do get a table by typing

```
. collect layout (colname[hlthstat]#result) (cmdset)
Collection: default
      Rows: colname[hlthstat]#result
   Columns: cmdset
   Table 1: 12 x 2
```

	Partial	Full
Very good		
Coefficient	.715111	.6759903
Std. error	.5519263	.5524101
Good		
Coefficient	2.233169	2.184542
Std. error	.5453581	.5461395
Fair		
Coefficient	4.133798	4.062105
Std. error	.6492333	.6506867
Poor		
Coefficient	3.549244	3.537842
Std. error	.8558511	.8558125

We have selected just the `hlthstat` level of dimension `colname`. Note that the "Base" column is no longer in the table. Variable `hlthstat` was not in the base regression, so there is no "Base" column to report when the table is limited to `colname[hlthstat]`.

We can even limit the table to just some of the levels of the factor variable `hlthstat`. To do that, we use standard factor-variable notation.

```
. collect layout (colname[2.hlthstat 4.hlthstat]#result) (cmdset)
Collection: default
      Rows: colname[2.hlthstat 4.hlthstat]#result
   Columns: cmdset
   Table 1: 6 x 2
```

	Partial	Full
Very good		
Coefficient	.715111	.6759903
Std. error	.5519263	.5524101
Fair		
Coefficient	4.133798	4.062105
Std. error	.6492333	.6506867

You can use full factor-variable notation, so typing

```
. collect layout (colname[i(2 4).hlthstat]#result) (cmdset)
```

would produce the same table.

Another thing we can do with dimensions `hlthstat` and `sex` is change their labels and the labels on their levels. Let's relabel the 4th level of `hlthstat`, and then repreview our most recent table.

```
. collect label levels hlthstat 4 "Between Very good and Poor", modify
. collect preview
```

	Partial	Full
Very good		
Coefficient	.715111	.6759903
Std. error	.5519263	.5524101
Between Very good and Poor		
Coefficient	4.133798	4.062105
Std. error	.6492333	.6506867

That leaves the two dimensions in the `Style only` group of `collect dims`—`border_block` and `cell_type`. These dimensions are for advanced use, but let's list the levels and labels for `cell_type` anyway.

```
. collect label list cell_type, all

   Collection: default
    Dimension: cell_type
        Label: Table cell type
 Level labels:
column-header
       corner
         item
   row-header
```

The levels `row-header`, `column-header`, `item`, and `corner` are referring to the cells in the four parts of a table—the cells in the row headers, the cells in the column headers, the `item` cells in the body of the table, and the no mans land of the upper left corner. When you type

```
. collect style cell cell_type[row-header], shading(background(blue))
```

you are changing the background color of all the cells in the row-header region to blue.

See [TABLES] **Example 4** for an example using dimension `cell_type`.

Surprisingly, the levels of dimension `border_block` are exactly the same as the levels of `cell_type`. Whereas dimension `cell_type` refers to the cells in the table regions, dimension `border_block` refers to the entire block of the region.

Special dimensions created by table

We have covered the most important special dimensions that can be created when you collect results. There may be others if your collection was created by `table`. The nomenclature is familiar now, so let's cover these dimensions quickly. Not because they are unimportant but because you are now ready to drink from the fire hose. Our examples will be terse and intended solely to demonstrate features, not to be interesting or meaningful.

The `table` command is built on top of the collection system. The `table` command builds a collection to hold all the results you request, customizes some styles, creates a layout, and then previews the table.

`table` names the collection it creates `Table`. If you run another `table` command, the collection `Table` is replaced with the collection created by the new `table` command. Collection `Table`, when it exists, always contains the collection for the most recent `table` command.

Dimension variables

We mentioned much earlier that there is not much difference in the collection created by a command like `collect: by region: summarize ...` and a command like `table region ...`. Both create a dimension named `region`, and its levels are the distinct values that the variable `region` takes on in the dataset. We discussed this type of dimension at length in *Tags, dimensions, and levels* through *Interactions in collect layout* and will say no more here.

Variables from statistic() option—dimension var

When you specify statistics using the `statistic()` option of `table`, `table` creates a dimension named `var` whose levels are the names of the variables for which statistics were computed. Take the simple table,

```
. table region, statistic(mean age lead) statistic(sd age lead)
```

	Mean		Standard deviation	
	Age (years)	Lead (mcg/dL)	Age (years)	Lead (mcg/dL)
Region				
NE	47.81584	14.83784	17.01692	5.782612
MW	46.52776	14.78544	17.37627	6.698146
S	48.19068	13.29985	16.86443	6.200866
W	47.83828	14.52686	17.53498	5.704972
Total	47.57965	14.32033	17.21483	6.166468

We can learn more about this table by typing `collect layout`:

```
. collect layout

Collection: Table
      Rows: region
   Columns: result#var
   Table 1: 6 x 4
```

	Mean		Standard deviation	
	Age (years)	Lead (mcg/dL)	Age (years)	Lead (mcg/dL)
Region				
NE	47.81584	14.83784	17.01692	5.782612
MW	46.52776	14.78544	17.37627	6.698146
S	48.19068	13.29985	16.86443	6.200866
W	47.83828	14.52686	17.53498	5.704972
Total	47.57965	14.32033	17.21483	6.166468

When specified without arguments, `collect layout` redisplays the most recent table it created, and yes, `table` used `collect layout` to create its table. Let's focus on the header that we have heretofore ignored. It tells us what the row specification was—`region`. And it tells us what the column specification was—`result#var`. Knowing those specifications can be truly convenient. If we want to rearrange the table rows and columns, we know which dimensions to use.

Dimension `var` is the new player in that specification. Let's look at `var` a little more closely.

```
. collect label list var

  Collection: Table
   Dimension: var
       Label: Statistic option variable
Level labels:
         age  Age (years)
        lead  Lead (mcg/dL)
```

We see levels `age` and `lead`. Those are the names of the variables we specified in the `statistic()` option. Dimension `var` looks a lot like the dimension `colname`, which we saw when collecting regression results. Great we know how to use dimensions like that. Let's shuffle our table so that the means and standard deviations are near each other.

```
. collect layout (var#result) (region)

Collection: Table
      Rows: var#result
   Columns: region
   Table 1: 6 x 5
```

	Region				
	NE	MW	S	W	Total
Age (years)					
Mean	47.81584	46.52776	48.19068	47.83828	47.57965
Standard deviation	17.01692	17.37627	16.86443	17.53498	17.21483
Lead (mcg/dL)					
Mean	14.83784	14.78544	13.29985	14.52686	14.32033
Standard deviation	5.782612	6.698146	6.200866	5.704972	6.166468

Dimension colname and matching to regressions

We said that dimension `var` looked a lot like dimension `colname`. In fact, they serve exactly the same purpose. So much so that `table` also creates dimension `colname`, which is identical to dimension `var`. This can be useful if you are trying to put results from `table` on the same rows or columns as results from regressions or regressionlike commands. Recall that `collect` puts covariate names into dimension `colname`.

Here is a silly example using `colname` to align the results from `table` and `regress`.

First, we type the `table` command.

```
. table, statistic(mean age lead) statistic(sd age lead)
```

Mean	
Age (years)	47.57965
Lead (mcg/dL)	14.32033
Standard deviation	
Age (years)	17.21483
Lead (mcg/dL)	6.166468

Then, we add our regression results to the `table` results.

```
. collect, name(Table): regress bpsystol age lead

      Source |       SS           df       MS      Number of obs   =     4,948
-------------+----------------------------------   F(2, 4945)      =    775.82
       Model |  640033.944         2  320016.972   Prob > F        =    0.0000
    Residual |  2039746.25     4,945  412.486602   R-squared       =    0.2388
-------------+----------------------------------   Adj R-squared   =    0.2385
       Total |  2679780.19     4,947  541.698038   Root MSE        =     20.31

------------------------------------------------------------------------------
    bpsystol | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
         age |   .6517974   .0168645    38.65   0.000     .6187355    .6848593
        lead |   .2680019   .0468828     5.72   0.000     .1760907     .359913
       _cons |    96.0544   1.057516    90.83   0.000      93.9812     98.1276
------------------------------------------------------------------------------
```

Note that we used the `collect` option `name()`, which we used to place our results into collection `Table`—the collection produced by the `table` command.

Behind the scenes, `table` sets the automatic levels of results to be only the results you have specified on the `table` command or what `table` thinks are sensible results to show if you have included a `command()` option. We need to add the regression results we wanted displayed to the automatic levels. Let's add coefficients and their standard errors.

```
. collect style autolevels result _r_b _r_se
```

All that is left is to specify how we want our table to look.

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

Collection: Table
      Rows: colname
   Columns: result
   Table 1: 3 x 4
```

	Mean	Standard deviation	Coefficient	Std. error
Age (years)	47.57965	17.21483	.6517974	.0168645
Lead (mcg/dL)	14.32033	6.166468	.2680019	.0468828
Intercept			96.0544	1.057516

We have both our `table` and `regress` results in one table.

We could organize the table as one column.

```
. collect layout (colname#result)
Collection: Table
      Rows: colname#result
   Table 1: 13 x 1
```

Age (years)	
Mean	47.57965
Standard deviation	17.21483
Coefficient	.6517974
Std. error	.0168645
Lead (mcg/dL)	
Mean	14.32033
Standard deviation	6.166468
Coefficient	.2680019
Std. error	.0468828
Intercept	
Coefficient	96.0544
Std. error	1.057516

Why would we want the results in one column? Perhaps we would like to compare the results across groups.

If we just add the `region` variable as the row specification to our `table` command, we will compute the means by the levels of `region`.

```
. table region, statistic(mean age lead) statistic(sd age lead) nototal
```

If we insert `by region:` into the command that collects regression results, the regression results will also be computed by the levels of `region`.

```
. collect, name(Table): by region, sort: regress bpsystol age lead
```

We still need to add to the automatic levels.

```
. collect style autolevels result _r_b _r_se
```

All that is left is to add dimension `region` as our column specification.

```
. collect layout (colname#result) (region)
Collection: Table
      Rows: colname#result
   Columns: region
   Table 1: 13 x 4
```

	Region			
	NE	MW	S	W
Age (years)				
Mean	47.81584	46.52776	48.19068	47.83828
Standard deviation	17.01692	17.37627	16.86443	17.53498
Coefficient	.6819023	.6143461	.6761958	.6459431
Std. error	.0390149	.0305406	.034739	.0319899
Lead (mcg/dL)				
Mean	14.83784	14.78544	13.29985	14.52686
Standard deviation	5.782612	6.698146	6.200866	5.704972
Coefficient	.3411097	.26796	.3455647	.1104092
Std. error	.1148679	.0796156	.0934401	.0969654
Intercept				
Coefficient	93.83657	97.91132	93.83902	98.2411
Std. error	2.50846	1.837282	2.150045	2.09812

Index of command() options—dimension command

The `table` command itself can collect results from multiple commands. Here is an example of two nested regressions.

```
. table, command(regress bpsystol age lead)
>        command(regress bpsystol age lead weight)

----------------------------------------|----------
regress bpsystol age lead               |
  Age (years)                           |  .6517974
  Lead (mcg/dL)                         |  .2680019
  Intercept                             |   96.0544
regress bpsystol age lead weight        |
  Age (years)                           |  .6373174
  Lead (mcg/dL)                         |  .1183383
  Weight (kg)                           |  .3998766
  Intercept                             |  70.08091
----------------------------------------|----------
```

Clearly, `table` is keeping track of the commands we typed; the full commands are shown right there on the table. The commands are the super rows, and the regression coefficients from the `result` dimension are the rows. `table` creates the dimension `command` and uses it to hold a level for each `command()` option.

```
. collect label list command

  Collection: Table
   Dimension: command
       Label: Command option index
Level labels:
           1  regress bpsystol age lead
           2  regress bpsystol age lead weight
```

We can put the commands on the columns for a more conventional regression comparison table.

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

Collection: Table
      Rows: colname#result
   Columns: command
   Table 1: 4 x 2

-------------------------------------------------------------------------------
                 | regress bpsystol age lead   regress bpsystol age lead weight
-------------------------------------------------------------------------------
Age (years)      |                  .6517974                           .6373174
Lead (mcg/dL)    |                  .2680019                           .1183383
Weight (kg)      |                                                     .3998766
Intercept        |                   96.0544                           70.08091
-------------------------------------------------------------------------------
```

We should clearly shorten the labels on the levels of `command` using the `collect label levels` command. We might also want to add the standard errors of the coefficients or other coefficient statistics using `collect style autolevels result`. We leave that as an exercise.

Index of command() and statistic() options—dimension statcmd

What if our `table` command has both `command()` and `statistic()` options?

```
. table region, statistic(mean age lead) statistic(sd age lead) ///
    command(regress bpsystol age lead) nototal
```

We are not going to show the output from that command because it would wrap on this page. Let's instead see how the table was laid out.

```
. collect layout

Collection: Table
      Rows: region
   Columns: statcmd#result#colname
   Table 1: 5 x 7
```

(*output omitted*)

We again omit the table from the output because it would wrap. Let's focus on the header. The only dimension we do not recognize is `statcmd` in the `Columns:` listing. Let's look at `statcmd`.

```
. collect label list statcmd

  Collection: Table
   Dimension: statcmd
       Label: Statistic/command option index
Level labels:
           1  Mean
           2  Standard deviation
           3  regress bpsystol age lead
```

So each level of `statcmd` represents one of our `statistic()` or `command()` option. Let's transpose our row and column specifications so we can finally see a table.

```
. collect layout (statcmd#result#colname) (region)

Collection: Table
      Rows: statcmd#result#colname
   Columns: region
   Table 1: 13 x 4
```

	Region			
	NE	MW	S	W
Mean				
Mean				
Age (years)	47.81584	46.52776	48.19068	47.83828
Lead (mcg/dL)	14.83784	14.78544	13.29985	14.52686
Standard deviation				
Standard deviation				
Age (years)	17.01692	17.37627	16.86443	17.53498
Lead (mcg/dL)	5.782612	6.698146	6.200866	5.704972
regress bpsystol age lead				
Coefficient				
Age (years)	.6819023	.6143461	.6761958	.6459431
Lead (mcg/dL)	.3411097	.26796	.3455647	.1104092
Intercept	93.83657	97.91132	93.83902	98.2411

Other dimensions

One other dimension that `table` sometimes creates automatically is `across()`. That dimension holds all the combinations of any `across()` options that are specified to determine over which groups percentages and proportions are computed. You will not use this dimension often.

`table` also creates any dimensions that `collect` would create for any commands that appear in `command()` options. Which is to say, any of the dimensions we have discussed in this entry and more. We already saw such dimensions when we included `command(regress ...)` in some of our examples above.

Let's talk styles

Overview

Styles affect how almost everything on your table looks, is organized, or composed. Even so, we are not going to categorize all the styles or even discuss what you can do with styles. That is done in the individual style entries. This entry is about concepts and how you use those concepts. For a categorization of styles with links to their entries, go to [TABLES] **Intro 4** and see these sections:

Change styles—formats, bolding, colors, and more

Control display of zero coefficients in regression results

Modify labels in row and column headers

There is a bit of labeling in that last section, but it also links to styles. In row and column headers, both content and format matter.

Basic targeting

What is common to all styles is changing what you want changed and not changing what you do not want changed. You may want to make all coefficients italicized but not any of the other results. You may want to emphasize all the statistics on the coefficient `age` by making them bold but not change the rest of the covariates. Hitting your target is what matters. So we will call this targeting.

We are going to use numeric format to demonstrate. Changes to numeric format can be seen in all export formats and in the Results window. Changes to numeric formats can even be seen in the Linux console version of Stata.

Let's use a table created from one of our simple regressions from earlier. We will not show the regression results,

```
. collect clear
. collect: regress bpsystol age weight lead
```

but we will show the table we lay out.

```
. collect layout (colname) (result[_r_b _r_ci _r_se _r_z _r_p])
Collection: default
      Rows: colname
   Columns: result[_r_b _r_ci _r_se _r_z _r_p]
   Table 1: 4 x 5
```

	Coefficient	95% CI		Std. error	t	p-value
Age (years)	.6373174	.6057546	.6688803	.0160998	39.59	0.000
Weight (kg)	.3998766	.3644918	.4352614	.0180494	22.15	0.000
Lead (mcg/dL)	.1183383	.0296721	.2070044	.0452276	2.62	0.009
Intercept	70.08091	67.04886	73.11296	1.546613	45.31	0.000

Command `collect style cell` has option `nformat()`, which lets us set the numeric format. Let's change all numeric formats on the entire table to %7.4f.

```
. collect style cell, nformat(%7.4f)
```

We did not specify anything after `cell`, so we are changing the format for everything. Let's see the effect of that change.

```
. collect preview
```

	Coefficient	95% CI		Std. error	t	p-value
Age (years)	0.6373	0.6058	0.6689	0.0161	39.5853	0.0000
Weight (kg)	0.3999	0.3645	0.4353	0.0180	22.1546	0.0000
Lead (mcg/dL)	0.1183	0.0297	0.2070	0.0452	2.6165	0.0089
Intercept	70.0809	67.0489	73.1130	1.5466	45.3125	0.0000

Everything has four decimals. What if we want to change the format of only the coefficients? Recall that the coefficients are level `_r_b` in dimension `result`. We simply specify the tag `result[_r_b]` as the only value for which we want to change the format.

```
. collect style cell result[_r_b], nformat(%7.2f)
. collect preview
```

	Coefficient	95% CI		Std. error	t	p-value
Age (years)	0.64	0.6058	0.6689	0.0161	39.5853	0.0000
Weight (kg)	0.40	0.3645	0.4353	0.0180	22.1546	0.0000
Lead (mcg/dL)	0.12	0.0297	0.2070	0.0452	2.6165	0.0089
Intercept	70.08	67.0489	73.1130	1.5466	45.3125	0.0000

Only the coefficients have two decimal places.

The format for the coefficients, their confidence intervals, and their standard errors is usually the same. Here is how we specify all of those results to have two decimal places.

```
. collect style cell result[_r_b _r_ci _r_se], nformat(%7.2f)
. collect preview
```

	Coefficient	95% CI		Std. error	t	p-value
Age (years)	0.64	0.61	0.67	0.02	39.5853	0.0000
Weight (kg)	0.40	0.36	0.44	0.02	22.1546	0.0000
Lead (mcg/dL)	0.12	0.03	0.21	0.05	2.6165	0.0089
Intercept	70.08	67.05	73.11	1.55	45.3125	0.0000

We typed `result[_r_b _r_ci _r_se]` to target all three of the results, just as we would type `result[_r_b _r_ci _r_se]` on `collect layout` to select the three results for rows or columns. Styles are yet another reason why tags, dimensions, and levels are so important in the collection system.

We could go on formatting results, but you get the idea.

We can target any dimension that tags any value or label on our table. If we wanted to draw our reader's attention to the results for covariate `lead`, we might change the color of its row to red, or we might bold the text. Instead, we will change the numeric format as a proxy for one of those more reasonable changes.

```
. collect style cell colname[lead], nformat(%7.5f)
. collect preview
```

	Coefficient	95% CI		Std. error	t	p-value
Age (years)	0.64	0.61	0.67	0.02	39.5853	0.0000
Weight (kg)	0.40	0.36	0.44	0.02	22.1546	0.0000
Lead (mcg/dL)	0.11834	0.02967	0.20700	0.04523	2.61651	0.00891
Intercept	70.08	67.05	73.11	1.55	45.3125	0.0000

And now the results for `lead` are "emphasized".

Let's fit this same regression on males, females, and all data. The `table` command makes that easy. We will not show the results of `table`.

```
. table sex, command(regress bpsystol age weight lead)
```

Instead, we will show some tidier results.

```
. collect layout (colname#result[_r_b _r_se]) (sex)

Collection: Table
      Rows: colname#result[_r_b _r_se]
   Columns: sex
   Table 1: 8 x 3
```

	Sex		
	Male	Female	Total
Age (years)	.4756206	.783255	.6373174
	.0221995	.023314	.0160998
Weight (kg)	.3499395	.440647	.3998766
	.0281172	.0262451	.0180494
Lead (mcg/dL)	.1154999	.1008595	.1183383
	.0580126	.0850915	.0452276
Intercept	81.09842	61.13921	70.08091
	2.700181	2.133394	1.546613

It is hard to tell the standard errors from the coefficients on that table. We could use a header style to add row labels for the coefficient and standard error, but let's instead put parentheses around the standard errors. That can be done using the `sformat()` option of `collect style cell`.

```
. collect style cell result[_r_se], sformat((%s))

. collect preview
```

	Sex		
	Male	Female	Total
Age (years)	.4756206	.783255	.6373174
	(.0221995)	(.023314)	(.0160998)
Weight (kg)	.3499395	.440647	.3998766
	(.0281172)	(.0262451)	(.0180494)
Lead (mcg/dL)	.1154999	.1008595	.1183383
	(.0580126)	(.0850915)	(.0452276)
Intercept	81.09842	61.13921	70.08091
	(2.700181)	(2.133394)	(1.546613)

Yes, somewhat surprisingly, you can apply both a numeric and a string format to a value. Once the value is numerically formatted, it is then passed through a string format. For numeric values, that string format is primarily used just as we used it here—to adorn the result.

Advanced targeting

What if we want to emphasize just one result in this whole table? What if the `age` coefficient for females was of particular import to our research? We saw just above that we could specify multiple tags by including multiple levels in a dimension using styles. We can also use tag interactions when applying styles. It takes three tags to identify the result we described—`result[_r_b]`, `colname[age]`, and `sex[2]`. The way we specify that all of those tags are required is to interact them—`result[_r_b]#colname[age]#sex[2]`. The translation of that interaction term into English is literally result must be coefficient and covariate must be `age` and `sex` must be female. We put that term as the argument to `collect style cell` and type the command.

```
. collect style cell result[_r_b]#colname[age]#sex[2], nformat(%7.2f)
```

Previewing our table gives

```
. collect preview
```

	Sex		
	Male	Female	Total
Age (years)	.4756206	0.78	.6373174
	(.0221995)	(.023314)	(.0160998)
Weight (kg)	.3499395	.440647	.3998766
	(.0281172)	(.0262451)	(.0180494)
Lead (mcg/dL)	.1154999	.1008595	.1183383
	(.0580126)	(.0850915)	(.0452276)
Intercept	81.09842	61.13921	70.08091
	(2.700181)	(2.133394)	(1.546613)

Our desired coefficient has been "highlighted".

More likely, we want to "highlight" both the coefficient and its standard error. That just requires that we specify the tags for both coefficient and standard error, rather than just for the coefficient.

```
. collect style cell result[_r_b _r_se]#colname[age]#sex[2], nformat(%7.2f)
. collect preview
```

	Sex		
	Male	Female	Total
Age (years)	.4756206	0.78	.6373174
	(.0221995)	(0.02)	(.0160998)
Weight (kg)	.3499395	.440647	.3998766
	(.0281172)	(.0262451)	(.0180494)
Lead (mcg/dL)	.1154999	.1008595	.1183383
	(.0580126)	(.0850915)	(.0452276)
Intercept	81.09842	61.13921	70.08091
	(2.700181)	(2.133394)	(1.546613)

Okay, we will do one thing just for looks. Let's get rid of that obnoxious vertical rule. You never see those in publications.

```
. collect style cell border_block, border(right, pattern(nil))
. collect preview
```

	Sex		
	Male	Female	Total
Age (years)	.4756206	0.78	.6373174
	(.0221995)	(0.02)	(.0160998)
Weight (kg)	.3499395	.440647	.3998766
	(.0281172)	(.0262451)	(.0180494)
Lead (mcg/dL)	.1154999	.1008595	.1183383
	(.0580126)	(.0850915)	(.0452276)
Intercept	81.09842	61.13921	70.08091
	(2.700181)	(2.133394)	(1.546613)

We specified the `border_block` dimension, but we did not need to target a specific level. We turned off right borders on every block in the table, which includes those that were creating that vertical rule. `pattern(nil)` is a programmery way of saying no line.

Saving and using

Do not forget you can save and use styles; see [TABLES] **collect style save**.

If you get a table styled just the way you want, you can save its style and apply that style to other similar tables. There is also nothing wrong with keeping all your style commands in their own do-file and running that do-file before you preview a similar table.

Either way works fine. The advantage of keeping your style commands in a do-file is that you can review and change them in the do-file. Keeping a do-file is more challenging if you are using the Table Builder to style your table.

Exporting

We are not going to say much about exporting, which seems odd given that exporting will be the end goal for many tables. There just is not much to say. You type `collect export`, followed by a filename with the format you want as the file suffix. That's about it. This is an entry about concepts, and exporting does not have many concepts to explain.

What we will tell you is that not all styles export to all export formats. If you are exporting to Microsoft Word or to HTML, you are in luck. Almost all styles export to those formats. If you are exporting to plain text (`.txt`), you are out of luck. Aside from numeric formats and some text positioning, almost no styles export to plain text.

To learn more about exporting tables from a collection, see `collect export`.

Saving collections

You can save and restore collections. There is not anything conceptually interesting to add to that.

We do recommend that if you are typing `collect` commands interactively that you do save your work by saving your collection.

Managing collections

You can list the collections in memory, set the current collection, copy collections, combine collections, rename collections, and drop collections. All of those operations can be useful. None of those operations is fraught with conceptual challenges.

Just to be clear: combining collections is no different from adding to an existing collection using repeated `collect` prefixes or `collect get` commands without `collect clearing`.

Also see

[TABLES] **Intro 3** — Workflow outline

[TABLES] **Intro 4** — Overview of commands