12 How to improve slicing rows

> **What will this tutorial cover?**
>
> In this tutorial you will learn about the functions `slice`, `slice_head`, `slice_tail`, `slice_max`, `slice_min` and `slice_sample`. All functions allow you to slice specific rows of your data frame.

> **Who do I have to thank?**
>
> Many thanks to akrun and Dan Chaltiel who discussed how to create bootstraps with `slice_sample` in this stackoverflow question.

Both slicing and filtering allow you to remove or keep rows in a data frame. Essentially, you can use both to achieve the same result, but their approaches differ. While `filter` works with conditions (e.g. `displ > 17`), `slice` works with indices.

12.1 Overview of the `slice` function

Suppose we want to remove the first, second and third rows in the economic data frame (574 rows) with `slice`. The time series dataset shows some important economic variables in the US from 1967 to 2015 by month.

```
economics %>%
  slice(1, 2, 3)
```

```
# A tibble: 3 x 6
  date         pce    pop psavert uempmed unemploy
  <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
1 1967-07-01  507. 198712    12.6     4.5     2944
2 1967-08-01  510. 198911    12.6     4.7     2945
3 1967-09-01  516. 199113    11.9     4.6     2958
```

`slice` keeps all rows for which you specify positive indices. Note that in R indexing starts with 1 and not with 0 as in most other programming languages. To make it more clear what rows `slice` keeps, let's add row numbers to our data frame:

```
economics %>%
  rownames_to_column(var = "row_number")
```

```
# A tibble: 574 x 7
   row_number date         pce    pop psavert uempmed unemploy
   <chr>      <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
 1 1          1967-07-01  507. 198712    12.6     4.5     2944
 2 2          1967-08-01  510. 198911    12.6     4.7     2945
 3 3          1967-09-01  516. 199113    11.9     4.6     2958
 4 4          1967-10-01  512. 199311    12.9     4.9     3143
 5 5          1967-11-01  517. 199498    12.8     4.7     3066
 6 6          1967-12-01  525. 199657    11.8     4.8     3018
 7 7          1968-01-01  531. 199808    11.7     5.1     2878
 8 8          1968-02-01  534. 199920    12.3     4.5     3001
 9 9          1968-03-01  544. 200056    11.7     4.1     2877
10 10         1968-04-01  544  200208    12.3     4.6     2709
# ... with 564 more rows
```

Let's then slice some arbitrary rows:

```
economics %>%
  rownames_to_column(var = "row_number") %>%
  slice(c(4, 8, 10))
```

```
# A tibble: 3 x 7
  row_number date         pce    pop psavert uempmed unemploy
  <chr>      <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
1 4          1967-10-01  512. 199311    12.9     4.9     3143
2 8          1968-02-01  534. 199920    12.3     4.5     3001
3 10         1968-04-01  544  200208    12.3     4.6     2709
```

You can see two things: First, you can see that we have retained lines 4, 8, and 10, which map to our provided indices. Second, you can also provide a vector of indices instead of the comma-separated indices in the `slice` function.

To remove specific rows, we can use negative indices. Suppose, we want to remove the first row from our data frame:

```
economics %>%
  slice(-1)
```

```
# A tibble: 573 x 6
   date         pce    pop psavert uempmed unemploy
   <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
 1 1967-08-01  510. 198911    12.6     4.7     2945
 2 1967-09-01  516. 199113    11.9     4.6     2958
 3 1967-10-01  512. 199311    12.9     4.9     3143
 4 1967-11-01  517. 199498    12.8     4.7     3066
 5 1967-12-01  525. 199657    11.8     4.8     3018
 6 1968-01-01  531. 199808    11.7     5.1     2878
 7 1968-02-01  534. 199920    12.3     4.5     3001
 8 1968-03-01  544. 200056    11.7     4.1     2877
 9 1968-04-01  544  200208    12.3     4.6     2709
10 1968-05-01  550. 200361    12       4.4     2740
# ... with 563 more rows
```

To remove the last row from our data frame, we need to determine the index of the last row. This is nothing else than the total number of rows in our data frame:

```
economics %>%
  slice(-nrow(.))
```

```
# A tibble: 573 x 6
   date         pce    pop psavert uempmed unemploy
   <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
 1 1967-07-01  507. 198712    12.6     4.5     2944
 2 1967-08-01  510. 198911    12.6     4.7     2945
 3 1967-09-01  516. 199113    11.9     4.6     2958
 4 1967-10-01  512. 199311    12.9     4.9     3143
 5 1967-11-01  517. 199498    12.8     4.7     3066
 6 1967-12-01  525. 199657    11.8     4.8     3018
 7 1968-01-01  531. 199808    11.7     5.1     2878
 8 1968-02-01  534. 199920    12.3     4.5     3001
 9 1968-03-01  544. 200056    11.7     4.1     2877
10 1968-04-01  544  200208    12.3     4.6     2709
# ... with 563 more rows
```

The function `slice` is quickly explained. More interesting, however, are the helper functions `slice_head`, `slice_tail`, `slice_max`, `slice_min`, and `slice_sample`, which we will now discuss in more detail. Essentially, all of these functions translate a semantic input ("give me the first 10 lines") into indices.

12.2 How to slice off the top and bottom of a data frame

Imagine you have conducted a survey and the first two rows in your survey were test data.

```
survey_results <- tribble(
  ~id,   ~name,      ~pre,  ~post,
  1,     "Test",     4,     4,
  2,     "Test",     6,     8,
  3,     "Millner",  2,     9,
  4,     "Josh",     4,     7,
  5,     "Bob",      3,     4
)
```

Of course, you don't want to do any calculations with the test data, so you need to get rid of them. `sample_head` does the job for you. The function allows you to slice the top n rows of your data frame.

```
survey_results %>%
  slice_head(
    n = 2
  )
```

```
# A tibble: 2 x 4
     id name    pre  post
  <dbl> <chr> <dbl> <dbl>
1     1 Test      4     4
2     2 Test      6     8
```

Well, that's not what we wanted. Instead of slicing the lines we need for our survey, we sliced the test results. Again, the slice function keeps the rows instead of removing them. One solution to this conundrum is to turn the problem around and slice off the tail of the data frame instead of its head:

```
survey_results %>%
  slice_tail(
```

```
    n = 3
  )
```

```
# A tibble: 3 x 4
     id name      pre  post
  <dbl> <chr>   <dbl> <dbl>
1     3 Millner     2     9
2     4 Josh        4     7
3     5 Bob         3     4
```

This approach however is shaky. How do I know that I need to slice off the last three rows of the data frame? What if the data frame gets larger as the number of participants increases? The better solution is to write code that tells the function to slice off all rows except the first two. This is nothing more than the total number of rows minus 2:

```
survey_results %>%
  slice_tail(
    n = nrow(.) - 2
  )
```

```
# A tibble: 3 x 4
     id name      pre  post
  <dbl> <chr>   <dbl> <dbl>
1     3 Millner     2     9
2     4 Josh        4     7
3     5 Bob         3     4
```

You could have solved the problem with `filter` as well, which might even be the more robust method:

```
survey_results %>%
  filter(name != "Test")
```

```
# A tibble: 3 x 4
     id name      pre  post
  <dbl> <chr>   <dbl> <dbl>
1     3 Millner     2     9
2     4 Josh        4     7
3     5 Bob         3     4
```

12.3 How to slice rows with the highest and lowest values in a given column

A common use case within the slice family is to slice rows that have the highest or lowest value within a column.

Finding these rows with `filter` would be tedious. To see how much, let's give it a try. Suppose we want to find the months in our data frame when unemployment was highest:

```
economics %>%
  filter(unemploy >= sort(.$unemploy, decreasing = TRUE)[10]) %>%
  arrange(desc(unemploy)) %>%
  select(date, unemploy)
```

```
# A tibble: 10 x 2
   date       unemploy
   <date>        <dbl>
 1 2009-10-01    15352
 2 2010-04-01    15325
 3 2009-11-01    15219
 4 2010-03-01    15202
 5 2010-02-01    15113
 6 2009-12-01    15098
 7 2010-11-01    15081
 8 2010-01-01    15046
 9 2009-09-01    15009
10 2010-05-01    14849
```

The code inside the filter function is hard to read. What we do here is pull the `unemploy` column from the data frame, sort the values and get the tenth value of the sorted vector.

A much easier way to achieve the same result is to use `slice_max`:

```
economics %>%
  slice_max(
    order_by = unemploy,
    n        = 10) %>%
  select(date, unemploy)
```

```
# A tibble: 10 x 2
   date       unemploy
   <date>        <dbl>
 1 2009-10-01    15352
 2 2010-04-01    15325
 3 2009-11-01    15219
 4 2010-03-01    15202
 5 2010-02-01    15113
 6 2009-12-01    15098
 7 2010-11-01    15081
 8 2010-01-01    15046
 9 2009-09-01    15009
10 2010-05-01    14849
```

For the first argument `order_by` you specify the column for which the highest values should be taken. With `n` you specify how many of the rows with the highest values you want to keep.

If you are more interested in the percentage of rows with the highest value, you can use the argument `prop`. For example, let's slice the 10% of months with the highest unemployment rate:

```
economics %>%
  slice_max(
    order_by = unemploy,
    prop = .1
  )
```

```
# A tibble: 57 x 6
   date          pce     pop psavert uempmed unemploy
   <date>      <dbl>   <dbl>   <dbl>   <dbl>    <dbl>
 1 2009-10-01  9932. 308189      5.4    18.9    15352
 2 2010-04-01 10113. 309191.     6.4    22.1    15325
 3 2009-11-01  9940. 308418      5.9    19.8    15219
 4 2010-03-01 10089. 309212      5.7    20.4    15202
 5 2010-02-01 10031. 309027      5.8    19.9    15113
 6 2009-12-01  9999. 308633      5.9    20.1    15098
 7 2010-11-01 10355. 310596.     6.6    21      15081
 8 2010-01-01 10002. 308833      6.1    20      15046
 9 2009-09-01  9883. 307946      5.9    17.8    15009
10 2010-05-01 10131  309369.     7      22.3    14849
# ... with 47 more rows
```

Similarly, you can keep the rows with the lowest values in a given column. For example, let's find the three months when the unemployment rate was lowest between 1967 and 2015:

```
economics %>%
  slice_min(
    order_by = unemploy,
    n        = 3
  )
```

```
# A tibble: 3 x 6
  date         pce    pop psavert uempmed unemploy
  <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
1 1968-12-01  576. 201621    11.1     4.4     2685
2 1968-09-01  568. 201095    10.6     4.6     2686
3 1968-10-01  572. 201290    10.8     4.8     2689
```

Given the absolute numbers, this was a long time ago.

12.4 How to combine the slice functions with group_by

The `slice` functions become especially powerful when combined with `group_by`. Suppose you want to find each month in the year when the unemployment rate was highest. The trick is that any function called after `group_by` is only applied to the subgroups.

```
library(lubridate)

(highest_unemploy_per_month <- economics %>%
  group_by(year = year(date)) %>%
  slice_max(
    order_by = unemploy,
    n        = 1
  ) %>%
  ungroup())
```

```
# A tibble: 49 x 7
   date         pce    pop psavert uempmed unemploy  year
   <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl> <dbl>
 1 1967-10-01  512. 199311    12.9     4.9     3143  1967
 2 1968-02-01  534. 199920    12.3     4.5     3001  1968
```

```
 3 1969-10-01  618. 203302    11.4     4.5     3049  1969
 4 1970-12-01  666. 206238    13.2     5.9     5076  1970
 5 1971-11-01  721. 208555    13.1     6.4     5161  1971
 6 1972-03-01  749. 209212    11.8     6.6     5038  1972
 7 1973-12-01  877. 212785    14.8     4.7     4489  1973
 8 1974-12-01  962. 214782    14       5.7     6636  1974
 9 1975-05-01 1019. 215523    17.3     9.4     8433  1975
10 1976-11-01 1189  218834    11.4     8.4     7620  1976
# ... with 39 more rows
```

A couple of things happened here. First, I loaded the lubridate package. If you have one of the latest versions of the tidyverse package, lubridate should have already been loaded with `library(tiyverse)` (see this tweet by Hadley Wickham). I then grouped the `economics` data frame in years (`group_by(year = year(date))`). Yes, you can create new columns inside `group_by`. The function `year` from the lubridate package allows me to extract the year from a date column:

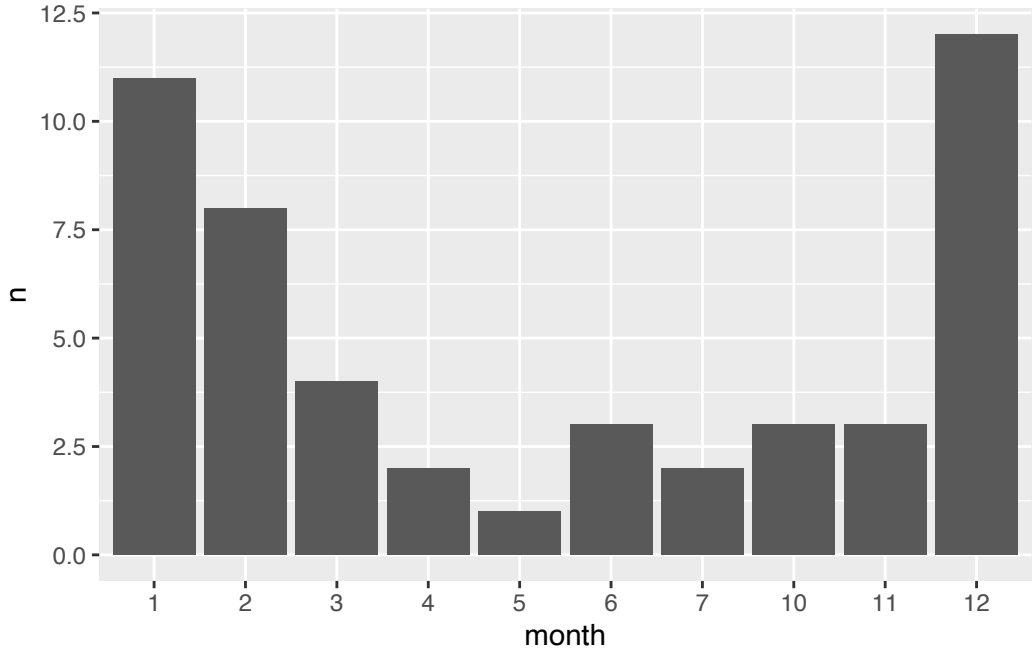
```
year(Sys.Date())
```

```
[1] 2022
```

Now that I have grouped the data by year, I can slice the months with the highest unemployment rate within each year (`slice_max(order_by = unemploy, n = 1)`). Again, this works because each function after `group_by` is applied only to the specific groups. At the end we terminate the grouping function with `ungroup`. Otherwise we would not apply the next functions to the whole data frame, but to the individual groups.

This data could be used, for example, to show in which months the unemployment rate is highest:

```
highest_unemploy_per_month %>%
  mutate(
    month = month(date) %>% as.factor
  ) %>%
  count(month) %>%
  ggplot(aes(x = month, y = n)) +
  geom_col()
```



12.5 How to create bootstraps with `slice_sample`

Another useful function is `slice_sample`. It randomly selects rows from your data frame. You define how many should be selected. Let's, for example, slice 20 rows from our data frame:

```
economics %>%
  slice_sample(n = 20)
```

```
# A tibble: 20 x 6
   date          pce     pop psavert uempmed unemploy
   <date>      <dbl>   <dbl>   <dbl>   <dbl>    <dbl>
 1 2012-04-01 10980. 313667.     8.7    19.1    12646
 2 1996-11-01  5351. 270878      6.4     7.7     7236
 3 1994-08-01  4763. 263724      6.5     8.9     7933
 4 2004-05-01  8163. 292997      5.3     9.9     8212
 5 2001-03-01  6988. 284350      5.3     6.6     6141
 6 2008-12-01  9731. 306004      6.4    10.5    11286
 7 2007-05-01  9651. 301483      4       8.2     6766
 8 1970-03-01   632. 204156     12.4     4.6     3635
 9 1986-06-01  2862. 240459      9.4     7       8508
```

```
10 2001-11-01  7168. 286341      4.1     7.7     8003
11 2003-09-01  7835  291321      5.2    10.2     8921
12 1979-05-01  1559. 224632     10.3     5.6     5840
13 2002-07-01  7380. 288105      5.5     8.9     8390
14 1976-02-01  1108. 217249     12.3     8.2     7326
15 2012-02-01 10954. 313339.     8      19.7    12813
16 1979-11-01  1657. 226027      9.7     5.3     6238
17 1996-08-01  5275  269976      6.6     8.4     6882
18 2003-02-01  7536. 289714      5.6     9.5     8618
19 1978-01-01  1330. 221477     11.9     6.5     6489
20 1997-04-01  5459. 272083      6.5     8.3     6873
```

Since the lines are randomly selected, you will see different rows. Now what happens when we sample all rows from our data frame:

```
economics %>%
  slice_sample(prop = 1)
```

```
# A tibble: 574 x 6
   date          pce     pop psavert uempmed unemploy
   <date>      <dbl>   <dbl>   <dbl>   <dbl>    <dbl>
 1 1984-09-01  2534. 236760     11.8     7.6     8367
 2 2015-02-01 12082. 320075.     7.9    12.9     8610
 3 1972-04-01   752. 209386     11.5     6.7     4959
 4 1973-03-01   833. 211254     12.7     5.5     4394
 5 1999-11-01  6438. 280471      4.8     6.2     5716
 6 2015-01-01 12046  319929.     7.7    13.2     8903
 7 2002-12-01  7513. 289313      5.5     9.6     8640
 8 2003-12-01  7929. 292008      5.4    10.4     8317
 9 1984-12-01  2583. 237316     11.2     7.3     8358
10 2003-02-01  7536. 289714      5.6     9.5     8618
# ... with 564 more rows
```

Nothing really changes. We will get the same data frame. Why? Because `slice_sample` by default samples without replacement. Once we have selected a row, we cannot select it again. Consequently, there will be no duplicate rows in our data frame. However, if we set the `replace` argument to `TRUE`, we will perform sampling with replacement:

```
set.seed(455)
(sample_with_replacement <- economics %>%
  slice_sample(prop = 1, replace = TRUE))
```

```
# A tibble: 574 x 6
   date          pce     pop psavert uempmed unemploy
   <date>      <dbl>   <dbl>   <dbl>   <dbl>    <dbl>
 1 1968-01-01   531. 199808     11.7     5.1     2878
 2 1968-07-01   563. 200706     10.7     4.5     2883
 3 2010-03-01 10089. 309212      5.7    20.4    15202
 4 1969-06-01   601. 202507     11.1     4.4     2816
 5 1968-09-01   568. 201095     10.6     4.6     2686
 6 1978-09-01  1453. 223053     10.6     5.6     6125
 7 1975-10-01  1061. 216587     13.4     8.6     7897
 8 1980-11-01  1827. 228612     11.6     7.7     8023
 9 1967-08-01   510. 198911     12.6     4.7     2945
10 2012-09-01 11062. 314647.     8.2    18.8    12115
# ... with 564 more rows
```

I set the seed to 455 so you get the same results. We can find the duplicate rows with the function `get_dupes` from the janitor package:

```
sample_with_replacement %>%
  janitor::get_dupes()
```

```
# A tibble: 363 x 7
   date          pce    pop psavert uempmed unemploy dupe_count
   <date>      <dbl>  <dbl>   <dbl>   <dbl>    <dbl>       <int>
 1 1967-08-01  510. 198911    12.6     4.7     2945           2
 2 1967-08-01  510. 198911    12.6     4.7     2945           2
 3 1968-02-01  534. 199920    12.3     4.5     3001           3
 4 1968-02-01  534. 199920    12.3     4.5     3001           3
 5 1968-02-01  534. 199920    12.3     4.5     3001           3
 6 1968-07-01  563. 200706    10.7     4.5     2883           3
 7 1968-07-01  563. 200706    10.7     4.5     2883           3
 8 1968-07-01  563. 200706    10.7     4.5     2883           3
 9 1968-09-01  568. 201095    10.6     4.6     2686           2
10 1968-09-01  568. 201095    10.6     4.6     2686           2
# ... with 353 more rows
```

As you can see, the first line appears twice in the data frame. Now, why would we do this? This functionality allows us to create bootstraps from our data frame. Bootstrapping is a technique where a set of samples of the same size are drawn from a single original sample. For example, if you have a vector `c(1, 4, 5, 6)`, you can create the following bootstraps from this vector: `c(1, 4, 4, 6)`, `c(1, 1, 1, 1)` or `c(5, 5, 1, 6)`. Some values appear

more than once because bootstrapping allows each value to be pulled multiple times from the original data set. Once you have your bootstraps, you can calculate metrics from them. For example, the mean value of each bootstrap. The underlying logic of this technique is that since the sample itself is from a population, the bootstraps act as proxies for other samples from that population.

Now that we have created one bootstrap from our sample, we can create many. In the following code I have used `map` to create 2000 bootstraps from my original sample.

```
bootstraps <- map(1:2000, ~slice_sample(economics, prop = 1, replace = TRUE))
```

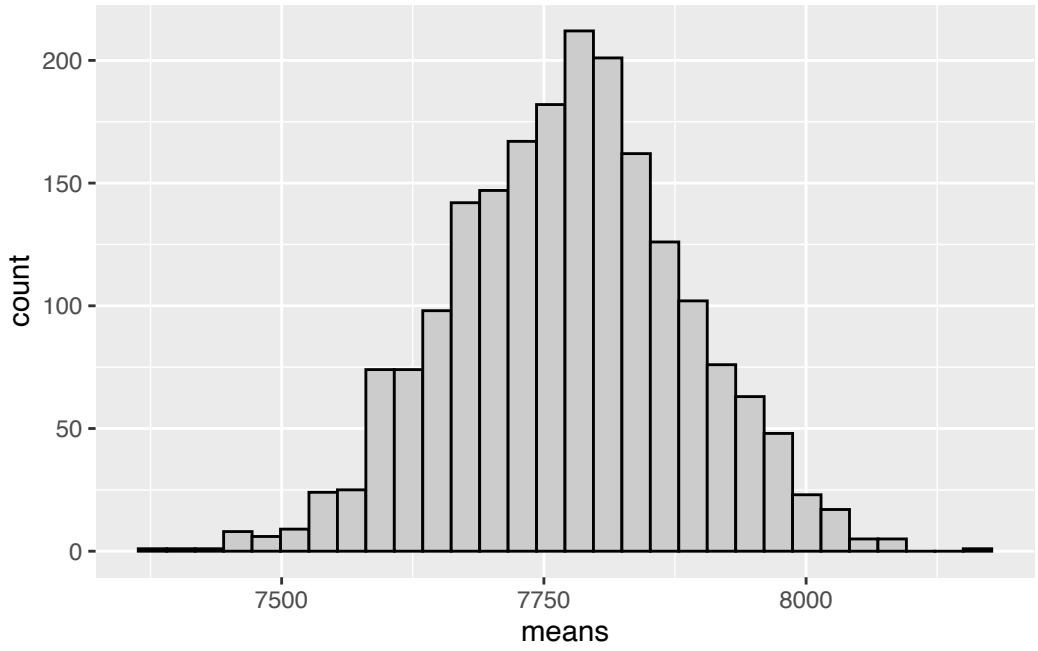
```
bootstraps %>% head(n = 2)
```

```
[[1]]
# A tibble: 574 x 6
   date         pce    pop psavert uempmed unemploy
   <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
 1 1992-04-01 4132. 255992     9.9     8.5     9415
 2 1993-09-01 4512. 260867     6.9     8.3     8714
 3 1999-05-01 6226. 278717     4.9     6.5     5796
 4 1989-07-01 3586. 247342     8.2     5.6     6495
 5 1986-05-01 2858. 240271     9.3     6.8     8439
 6 2006-11-01 9380. 300094     3.9     8.3     6872
 7 2000-05-01 6708. 281877     4.9     5.8     5758
 8 1996-12-01 5379. 271125     6.4     7.8     7253
 9 1992-10-01 4285. 257861     8       9       9398
10 1970-11-01  657. 206024    13.6     5.6     4898
# ... with 564 more rows

[[2]]
# A tibble: 574 x 6
   date          pce     pop psavert uempmed unemploy
   <date>      <dbl>   <dbl>   <dbl>   <dbl>    <dbl>
 1 1979-11-01  1657. 226027      9.7     5.3     6238
 2 1992-07-01  4205. 256894      9.6     8.6     9850
 3 1975-03-01   991. 215198     12.7     7.2     7978
 4 1970-07-01   648. 205052     13.5     5.1     4175
 5 2013-11-01 11488. 317228.     6.2    17.1    10787
 6 1999-12-01  6539. 280716      4.4     5.8     5653
 7 1978-10-01  1467. 223271     10.7     5.9     5947
 8 1980-04-01  1695. 227061     11.3     5.8     7358
```

```
 9 1995-12-01  5098. 267943      6.1     8.3     7423
10 1993-11-01  4554. 261425      6.3     8.3     8542
# ... with 564 more rows
```

`map` returns a list of data frames. Once we have the bootstraps, we can calculate any metric from them. Usually one calculates confidence intervals, standard deviations, but also measures of center like the mean from the bootstraps. Let's do the latter:

```
means <- map_dbl(bootstraps, ~ mean(.$unemploy))

ggplot(NULL, aes(x = means)) +
  geom_histogram(fill = "grey80", color = "black")
```



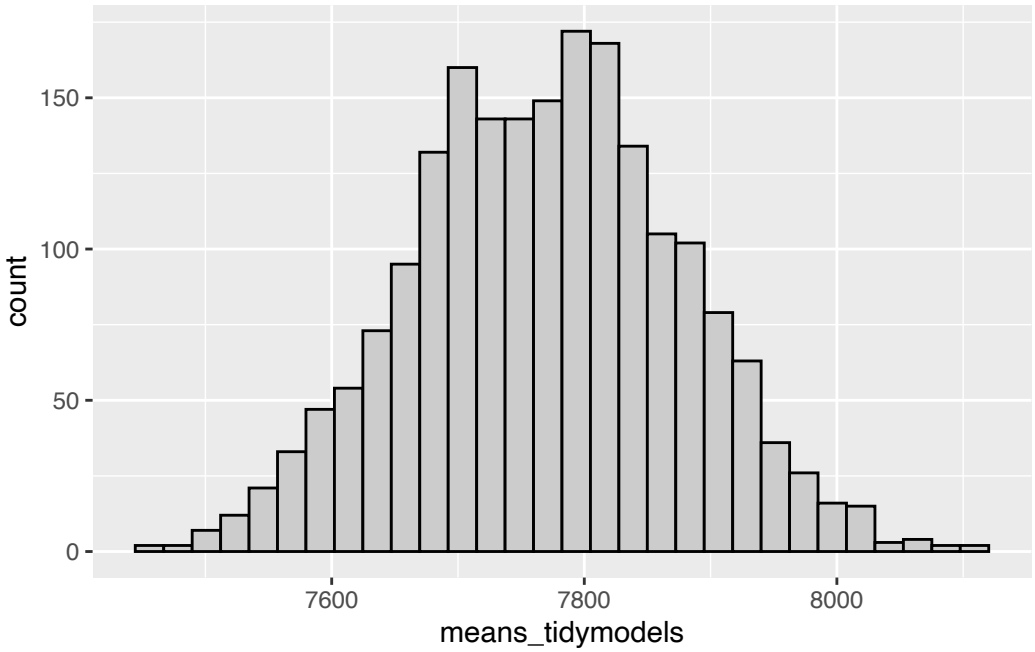
As you can see, the distribution of the mean is normally distributed. Most of the mean values are around 7750, which is pretty close to the mean value of our sample:

```
economics$unemploy %>% mean
```

```
[1] 7771.31
```

We can compare this result with the `bootstraps` function from the tidymodels package, which is more rigorous than our approach (It is not too important that you understand the code here. Basically, we use the `bootstraps` function to create a similar distribution of the mean values).

```
boostraps_tidymodels <- rsample::bootstraps(economics, times = 2000)

means_tidymodels <- map_dbl(boostraps_tidymodels$splits,
                            ~ mean(rsample::analysis(.)$unemploy))

ggplot(NULL, aes(x = means_tidymodels)) +
  geom_histogram(fill = "grey80", color = "black")
```



This distribution is very similar to our distribution we created with `slice_sample`.

Summary

Here's what you can take away from this tutorial.

- The slice functions slice rows based on their indices. Positive indices are kept, negative indices are removed.
- `group_by` and `slice_max` / `slice_min` is a powerful combination to reduce the size

of your data frame by finding the rows within groups with the highest or lowest values.

- `slice_sample` can be used to create bootstraps from your data frame