Data Wrangling in the Tidyverse Part 2

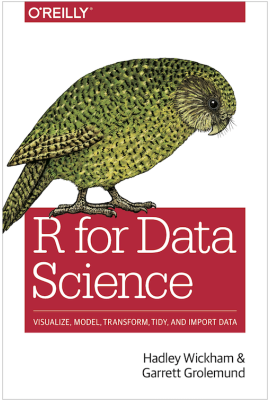
Stats 102A

Miles Chen

Department of Statistics and Data Science

UCLA

Resource: R For Data Science



Portions of this lecture are derived from the book. The book is Free to read:
https://r4ds.had.co.nz/

Section 1

Grouping Data

Summarizing Data with summarize()

The summarize() function (also spelled summarise()) is used to compute summary statistics for a dataset. It reduces multiple values into a single summary value per group or for the entire dataset.

```
starwars |>
  select(height, mass) |>
  summarize(
    avg_height = mean(height, na.rm = TRUE),
    var_height = var(height, na.rm = TRUE),
    avg_mass = mean(mass, na.rm = TRUE),
    min_height = min(height, na.rm = TRUE),
    max_mass = max(mass, na.rm = TRUE),
    count = n()
  )
```

```
# A tibble: 1 x 6
  avg_height var_height avg_mass min_height max_mass count
       <dbl>      <dbl>    <dbl>      <int>    <dbl> <int>
1       175.      1209.     97.3         66     1358    87
```

- mean(), var(), min(), max() compute **summary statistics**.
- n() counts the **number of non-missing rows**.
- na.rm = TRUE ensures that missing values are ignored.

Creating Groups with group_by()

The group_by() function categorizes data into groups, allowing you to perform computations within each group.

```
starwars |>
  group_by(species) |>
  select(name, height, mass, species)
```

```
# A tibble: 87 x 4
# Groups:   species [38]
   name               height  mass species
   <chr>               <int> <dbl> <chr>
 1 Luke Skywalker        172    77 Human
 2 C-3PO                 167    75 Droid
 3 R2-D2                  96    32 Droid
 4 Darth Vader           202   136 Human
 5 Leia Organa           150    49 Human
 6 Owen Lars             178   120 Human
 7 Beru Whitesun Lars    165    75 Human
 8 R5-D4                  97    32 Droid
 9 Biggs Darklighter     183    84 Human
10 Obi-Wan Kenobi        182    77 Human
# i 77 more rows
```

This does **not** change the data directly but sets up groups for further operations.

Combining group_by() and summarize()

The real power of group_by() comes when it is combined with summarize().

The following computes **mean, standard deviation, and count** for each species.

```r
starwars |>
  group_by(species) |>
  summarize(
    mean_ht = mean(height, na.rm = TRUE),
    sd_ht = sd(height, na.rm = TRUE),
    mean_mass = mean(mass, na.rm = TRUE),
    sd_mass = sd(mass, na.rm = TRUE),
    count = n()
  )
```

```
# A tibble: 38 x 6
   species   mean_ht sd_ht mean_mass sd_mass count
   <chr>       <dbl> <dbl>     <dbl>   <dbl> <int>
 1 Aleena        79    NA       15      NA       1
 2 Besalisk     198    NA      102      NA       1
 3 Cerean       198    NA       82      NA       1
 4 Chagrian     196    NA      NaN      NA       1
 5 Clawdite     168    NA       55      NA       1
 6 Droid        131.  49.1      69.8    51.0      6
 7 Dug          112    NA       40      NA       1
```

Filtering and Sorting Grouped Summaries

We can filter and arrange grouped summaries to focus on meaningful results.

```r
starwars |>
  group_by(species) |>
  summarize(
    mean_ht = mean(height, na.rm = TRUE),
    sd_ht = sd(height, na.rm = TRUE),
    mean_mass = mean(mass, na.rm = TRUE),
    sd_mass = sd(mass, na.rm = TRUE),
    count = n()
  ) |>
  filter(count > 1) |>  # Keep species with more than 1 character
  arrange(desc(count)) |> # Sort by count in descending order
  print(n = 6)
```

```
# A tibble: 9 x 6
  species  mean_ht sd_ht mean_mass sd_mass count
  <chr>      <dbl> <dbl>     <dbl>   <dbl> <int>
1 Human       178  12.0       81.3    19.3    35
2 Droid       131. 49.1       69.8    51.0     6
3 <NA>        175  12.4       81      31.2     4
4 Gungan      209. 14.2       74      11.3     3
5 Kaminoan    221  11.3       88      NA       2
6 Mirialan    168   2.83      53.1     4.38    2
```

group_by() + mutate()

mutate() can be combined with group_by() to compute new variables **within each group**.

```
starwars |>
  filter(species %in% c("Human", "Droid") | is.na(species)) |>
  select(name, species, height) |>
  group_by(species) |>
  mutate(z_height = (height - mean(height, na.rm = TRUE)) / sd(height, na.rm = TRUE)) |>
  print(n = 3)
```

```
# A tibble: 45 x 4
# Groups:   species [3]
  name           species height z_height
  <chr>          <chr>    <int>    <dbl>
1 Luke Skywalker Human      172   -0.498
2 C-3PO          Droid      167    0.728
3 R2-D2          Droid       96   -0.716
# i 42 more rows
```

- The **z-score** of height is computed **within each species**.
- A character's height is compared **to the mean height of their species**.

C-3PO is **above average** among droids but **below average** when compared to all characters.

Computing Without group_by()

If we **do not** use `group_by()`, `mutate()` computes statistics using **all characters** instead of grouping by species.

```
starwars |>
  filter(species %in% c("Human", "Droid") | is.na(species)) |>
  select(name, species, height) |>
  # group_by(species) |>
  mutate(z_height = (height - mean(height, na.rm = TRUE)) / sd(height, na.rm = TRUE)) |>
  print(n = 3)
```

```
# A tibble: 45 x 4
  name           species height z_height
  <chr>          <chr>    <int>    <dbl>
1 Luke Skywalker Human      172   0.0123
2 C-3PO          Droid      167  -0.188
3 R2-D2          Droid       96  -3.03
# i 42 more rows
```

- Without `group_by()`, **all characters** are used to compute `mean(height)`.
- C-3PO's z-score now compares his height **to all characters**, not just droids.

Grouping by Multiple Variables

You can provide `group_by()` with **multiple variables**, which creates a **hierarchical grouping**. This allows summarization at different levels.

```
toy_cases <- read_csv("https://raw.githubusercontent.com/rstudio/EDAWR/master/data-raw/toyb.csv")
print(toy_cases)
```

```
# A tibble: 12 x 4
   country      year sex    cases
   <chr>       <dbl> <chr>  <dbl>
 1 Afghanistan  1999 female     1
 2 Afghanistan  1999 male       1
 3 Afghanistan  2000 female     1
 4 Afghanistan  2000 male       1
 5 Brazil       1999 female     2
 6 Brazil       1999 male       2
 7 Brazil       2000 female     2
 8 Brazil       2000 male       2
 9 China        1999 female     3
10 China        1999 male       3
11 China        2000 female     3
12 China        2000 male       3
```

Multiple group_by() + summarize()

You can **group by more than one variable** to create hierarchical summaries.

```
summary1 <-
  toy_cases |>
  group_by(country, year) |>
  summarize(cases = sum(cases))
print(summary1)
```

```
# A tibble: 6 x 3
# Groups:   country [3]
  country      year cases
  <chr>       <dbl> <dbl>
1 Afghanistan  1999     2
2 Afghanistan  2000     2
3 Brazil       1999     4
4 Brazil       2000     4
5 China        1999     6
6 China        2000     6
```

Aggregating Across Grouped Summaries

We can further summarize across **higher levels of grouping**.

```
summary2 <- summary1 |> summarize(cases = sum(cases))
summary2
```

```
# A tibble: 3 x 2
  country     cases
  <chr>       <dbl>
1 Afghanistan     4
2 Brazil          8
3 China          12
```

```
summary3 <- summary2 |> summarize(cases = sum(cases))
summary3
```

```
# A tibble: 1 x 1
  cases
  <dbl>
1    24
```

Each summarize operation collapses another level of grouping, aggregating data progressively.

Changing the Order of group_by()

Changing the order of variables in group_by() affects how data is grouped and summarized.

Example: Grouping by year First, Then country

```
summary_a <- toy_cases |> group_by(year, country) |>
  summarize(cases = sum(cases))
print(summary_a)
```

```
# A tibble: 6 x 3
# Groups:   year [2]
   year country     cases
  <dbl> <chr>       <dbl>
1  1999 Afghanistan     2
2  1999 Brazil          4
3  1999 China           6
4  2000 Afghanistan     2
5  2000 Brazil          4
6  2000 China           6
```

Aggregating Across Reordered Groups

```r
summary_b <- summary_a |> summarize(cases = sum(cases))
summary_b
```

```
# A tibble: 2 x 2
   year cases
  <dbl> <dbl>
1  1999    12
2  2000    12
```

```r
summary_c <- summary_b |> summarize(cases = sum(cases))
summary_c
```

```
# A tibble: 1 x 1
  cases
  <dbl>
1    24
```

This progressively collapses the data one level at a time.

Section 2

Merging Tables

Two-table verbs

Two-table verbs with dplyr

When working with multiple datasets, you'll often need to combine them to get insights from multiple sources. The **dplyr** package provides several useful functions (verbs) to combine data:

- **Mutating joins**: Add new columns to one table based on matching rows from another.
- **Filtering joins**: Keep rows based on matches found (or not found) in another table.
- **Set operations**: Treat observations as elements of mathematical sets, allowing intersection, union, and differences.

Note: If you're familiar with SQL, these joins will look very similar.

For additional details, see the official documentation:
https://dplyr.tidyverse.org/articles/two-table.html

Vocabulary

- **Fact table**: many rows of events (e.g., `flights`)
- **Dimension table**: small lookup/labels (e.g., `airlines`, `airports`)
- **Primary key**: unique in dimension
- **Foreign key**: repeats in fact
- **Join order mantra**: *Start from the table you want rows from; join in what you need.*

Example Data Tables

We'll use two small "toy" tables to illustrate these concepts clearly:

```r
people <- tibble(
  name = c("Adam", "Betty", "Carl", "Doug"),
  state = c("CA", "CA", "NY", "TX")
  )
states <- tibble(
  abbreviation = c("CA", "NY", "WA"),
  state_name = c("California", "New York", "Washington")
  )
```

left_join()

```
people table                  states table
   name  state                 abbreviation state_name
 1 Adam  CA                   1 CA           California
 2 Betty CA                   2 NY           New York
 3 Carl  NY                   3 WA           Washington
 4 Doug  TX
```

`left_join()` takes all the values in the **left** table and adds columns from the right table by matching values using a column that exists in both tables. Values that do not exist in the other table have `NA` returned.

```r
people |>
  left_join(states, by = join_by(state == abbreviation))
```

```
# A tibble: 4 x 3
  name  state state_name
  <chr> <chr> <chr>
1 Adam  CA    California
2 Betty CA    California
3 Carl  NY    New York
4 Doug  TX    <NA>
```

Result: All names remain, but Doug (TX) has no matching state_name, so gets `NA`.

right_join()

```
people table
   name  state
 1 Adam  CA
 2 Betty CA
 3 Carl  NY
 4 Doug  TX
```

```
states table
  abbreviation state_name
1 CA           California
2 NY           New York
3 WA           Washington
```

`right_join()` is similar to `left_join` except it keeps all the rows in the **right** table, adding matching columns from the left table. Non-matching rows receive `NA`. In general, we recommend using only `left_join()` and switching the order of tables instead.

```r
people |>
  right_join(states, by = join_by(state == abbreviation))
```

```
# A tibble: 4 x 3
  name  state state_name
  <chr> <chr> <chr>
1 Adam  CA    California
2 Betty CA    California
3 Carl  NY    New York
4 <NA>  WA    Washington
```

Result: All states remain, but “Washington” (WA) has no matching person, so gets `NA`.

left_join() with table order switched

```
people table                      states table
   name  state                     abbreviation state_name
 1 Adam  CA                      1 CA           California
 2 Betty CA                      2 NY           New York
 3 Carl  NY                      3 WA           Washington
 4 Doug  TX
```

Note that the rows are the same, but the column orders are different. Also notice that the names in the `by =` argument are swapped.

```r
states |>
  left_join(people, by = join_by(abbreviation == state))
```

```
# A tibble: 4 x 3
  abbreviation state_name name
  <chr>        <chr>      <chr>
1 CA           California Adam
2 CA           California Betty
3 NY           New York   Carl
4 WA           Washington <NA>
```

inner_join()

```
people table
   name  state
 1 Adam  CA
 2 Betty CA
 3 Carl  NY
 4 Doug  TX
```

```
 states table
  abbreviation state_name
1 CA           California
2 NY           New York
3 WA           Washington
```

inner_join() keeps only rows that have values that exist in **both** tables. You can think of this as the intersection.

```
people |>
  inner_join(states, by = join_by(state == abbreviation))
```

```
# A tibble: 3 x 3
  name  state state_name
  <chr> <chr> <chr>
1 Adam  CA    California
2 Betty CA    California
3 Carl  NY    New York
```

Result: Doug (TX) and Washington (WA) are omitted since no matching pair exists.

full_join()

```
people table                        states table
   name  state                       abbreviation state_name
 1 Adam  CA                         1 CA           California
 2 Betty CA                         2 NY           New York
 3 Carl  NY                         3 WA           Washington
 4 Doug  TX
```

`full_join()` keeps **all** rows from both tables. Rows without matches receive `NA`. You can think of this as the union.

```r
people |>
  full_join(states, by = join_by(state == abbreviation))
```

```
# A tibble: 5 x 3
  name  state state_name
  <chr> <chr> <chr>
1 Adam  CA    California
2 Betty CA    California
3 Carl  NY    New York
4 Doug  TX    <NA>
5 <NA>  WA    Washington
```

Result: All rows from both tables appear, with `NA` filling unmatched fields.

Controlling Matching Columns (by argument)

Depending on the tables, the join operation can match tables on different variables.

In the previous examples, we used a named character vector `by = join_by(state == abbreviation)` specifying the name in the left table that matches the name in the right table.

Options for joining tables

- `by = join_by(x)` when names match
- `by = join_by(x == y)` when names differ
- Although you can leave the `by =` argument blank, **you should always specify something for `by =`**

Non-unique matches (Cartesian explosions)

If the joining key is not unique, all possible combinations (Cartesian product) of matches will be included. Usually, this indicates an error in the data.

```
people <- tibble(
  name = c("Adam", "Betty", "Carl", "Doug"),
  state = c("CA", "CA", "NY", "TX")
  )
places <- tibble(
  abbreviation = c("CA", "NY", "WA", "CA"),
  state_name = c("California", "New York", "Washington", "Canada")
  )
```


Non-unique matches

```
people table                  places table
   name  state                 abbreviation state_name
 1 Adam   CA                  1 CA          California
 2 Betty  CA                  2 NY          New York
 3 Carl   NY                  3 WA          Washington
 4 Doug   TX                  4 CA          Canada
```

```r
people |> left_join(places, by = join_by(state == abbreviation))
```

```
# A tibble: 6 x 3
  name  state state_name
  <chr> <chr> <chr>
1 Adam  CA    California
2 Adam  CA    Canada
3 Betty CA    California
4 Betty CA    Canada
5 Carl  NY    New York
6 Doug  TX    <NA>
```

- Adam and Betty from CA each appear twice, once for California, once for Canada.
- **Avoiding such issues:** Always ensure keys are unique unless duplicates are intentional.

De-duplicate first

You can use `distinct()` to get rid of duplicates. However, in this case, it means eliminating Canada (California comes before Canada in the alphabet).

```
places_dedup <- places |>
  distinct(abbreviation, .keep_all = TRUE)

people |> left_join(places_dedup, by = join_by(state == abbreviation))
```

```
# A tibble: 4 x 3
  name  state state_name
  <chr> <chr> <chr>
1 Adam  CA    California
2 Betty CA    California
3 Carl  NY    New York
4 Doug  TX    <NA>
```

Additional Join Operations

Filtering joins (`semi_join()` and `anti_join()`):

- `semi_join(x, y)`: Keep rows from `x` if they have matches in `y`. Does not add columns from `y`.
- `anti_join(x, y)`: Keep rows from `x` if they have no matches in `y`.

Example of semi_join():

```r
people |>
  semi_join(states, join_by(state == abbreviation))
```

```
# A tibble: 3 x 2
  name  state
  <chr> <chr>
1 Adam  CA
2 Betty CA
3 Carl  NY
```

- Returns Adam, Betty, Carl (matching states only, no added columns).

Example of anti_join():

```r
people |>
  anti_join(states, join_by(state == abbreviation))
```

```
# A tibble: 1 x 2
  name  state
  <chr> <chr>
1 Doug  TX
```

- Returns Doug (state TX with no match).

Set operations: Intersect / union / setdiff on rows

```r
west_coast <- tibble(state = c("CA","OR","WA"))
people_states <- tibble(state = c("CA","NY","TX"))

intersect(people_states, west_coast)
```

```
# A tibble: 1 x 1
  state
  <chr>
1 CA
```

```r
setdiff(people_states, west_coast)
```

```
# A tibble: 2 x 1
  state
  <chr>
1 NY
2 TX
```

```r
union(people_states, west_coast)
```

```
# A tibble: 5 x 1
  state
  <chr>
1 CA
2 NY
3 TX
4 OR
5 WA
```

Section 3

Reshaping Tables

Section 4

Reshaping Data

Toy data sets

```
storms_url <- "https://raw.githubusercontent.com/rstudio/EDAWR/master/data-raw/storms.csv"
storms <- read_csv(storms_url, show_col_types = FALSE)
cases_url <- "https://raw.githubusercontent.com/rstudio/EDAWR/master/data-raw/cases.csv"
cases <- read_csv(cases_url, show_col_types = FALSE)
pollution_url <- "https://raw.githubusercontent.com/rstudio/EDAWR/master/data-raw/pollution.csv"
pollution <- read_csv(pollution_url, show_col_types = FALSE)
```

Storms Table

```
storms
```

```
# A tibble: 6 x 4
  storm    wind pressure date      
  <chr>   <dbl>    <dbl> <date>    
1 Alberto   110     1007 2000-08-03
2 Alex       45     1009 1998-07-27
3 Allison    65     1005 1995-06-03
4 Ana        40     1013 1997-06-30
5 Arlene     50     1010 1999-06-11
6 Arthur     45     1010 1996-06-17
```

- Each row represents a different tropical storm. The observational unit is a tropical storm.
- The variables are:
  - storm name
  - wind speed
  - air pressure
  - date
- We have one column for each variable
- This data is tidy as is. No reshaping needed.

mutate() works well

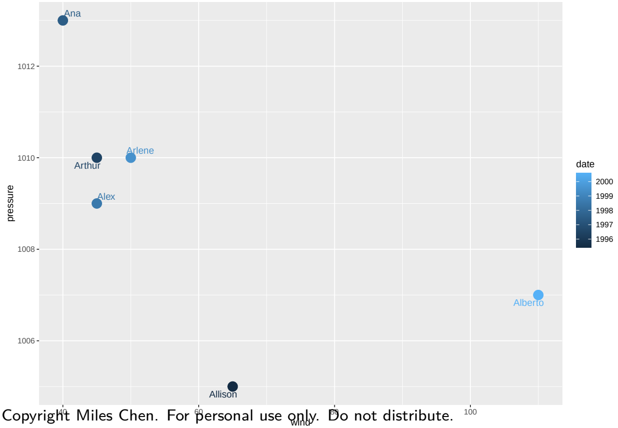
```r
storms |> mutate(ratio = pressure / wind)
```

```
# A tibble: 6 x 5
  storm     wind pressure date       ratio
  <chr>    <dbl>    <dbl> <date>     <dbl>
1 Alberto    110     1007 2000-08-03  9.15
2 Alex        45     1009 1998-07-27 22.4
3 Allison     65     1005 1995-06-03 15.5
4 Ana         40     1013 1997-06-30 25.3
5 Arlene      50     1010 1999-06-11 20.2
6 Arthur      45     1010 1996-06-17 22.4
```

ggplot() works well

With each variable in its own column, we can map the variables to different aesthetics.

```
library(ggrepel)
storms |> ggplot(aes(x = wind, y = pressure, color = date)) +
  geom_point(size = 5) +
  geom_text_repel(aes(label = storm))
```



Cases table

This is a table with fictional data regarding the cases of a certain infection in different years.

```
cases
```

```
# A tibble: 3 x 4
  country `2011` `2012` `2013`
  <chr>    <dbl>  <dbl>  <dbl>
1 FR        7000   6900   7000
2 DE        5800   6000   6200
3 US       15000  14000  13000
```

- Each row is a country. The observational unit is country.
- The variables are:
  - number of cases in 2011
  - number of cases in 2012
  - number of cases in 2013

mutate() works well

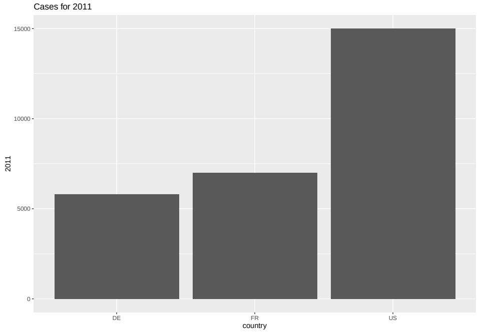
```r
cases |> mutate(change_11_12 = `2012`-`2011`, change_12_13 = `2013`-`2012`, )
```

```
# A tibble: 3 x 6
  country `2011` `2012` `2013` change_11_12 change_12_13
  <chr>    <dbl>  <dbl>  <dbl>        <dbl>        <dbl>
1 FR        7000   6900   7000         -100          100
2 DE        5800   6000   6200          200          200
3 US       15000  14000  13000        -1000        -1000
```

However, plotting is limited

Because each column represents a different year, it is difficult to produce a plot that allows us to compare the different years against each other. ggplot() requires that only one column gets mapped to y. To see the both 2011 and 2012 numbers, I would have to make separate plots.

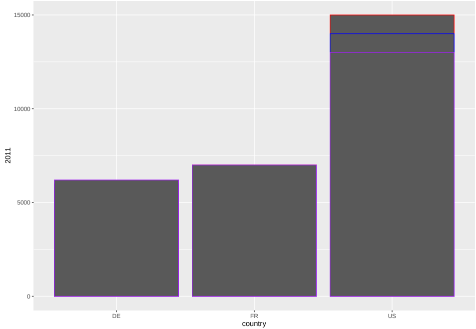
```
cases |> ggplot(aes(x = country, y = `2011`)) +
  geom_col() +
  ggtitle("Cases for 2011")
```



Plotting is limited

Attempting to plot the different years by adding multiple `geom_col()` layers is possible, but results in a plot that is not not very readable without additional work.

```
cases |> ggplot(aes(x = country)) +
  geom_col(aes(y = `2011`), col = "red") +
  geom_col(aes(y = `2012`), col = "blue") +
  geom_col(aes(y = `2013`), col = "purple")
```



Pivoting to the rescue!

What I need to do is reshape or pivot the data so that each row represents the count of cases in a country for a given year.

I want the variables to be:

- country
- year
- count of cases

If the data is structured this way, I can map the `year` column to an aesthetic attribute.

pivot_longer()

To achieve the desired result, we use the function pivot_longer() because we want the resulting data set to be longer than the original data. (Older versions called this gather()). After pivoting the data, notice that we now have 9 rows - one for each country and year.

```
pivot_longer(cases,
             cols = "2011":"2013",
             names_to = "year",
             values_to = "cases")
```

```
# A tibble: 9 x 3
  country year  cases
  <chr>   <chr> <dbl>
1 FR      2011   7000
2 FR      2012   6900
3 FR      2013   7000
4 DE      2011   5800
5 DE      2012   6000
6 DE      2013   6200
7 US      2011  15000
8 US      2012  14000
9 US      2013  13000
```

The pivot_longer() function

```
pivot_longer(data = cases,
             cols = "2011":"2013",
             names_to = "year",
             values_to = "cases")
```

The pivot_longer() function takes in a few arguments:

- data is the name of the data.frame or tibble that we will pivot
- cols are the names of the columns that will be pivoted. In this case, we want the columns named "2011" through "2013". You can specify a range of column names with the : operator. Otherwise, you can provide a vector of column names
- names_to is a character string with what you want to call the resulting column of names. The former column names will be put into this column.
- values_to is a character string with what you want to call the resulting column of values. The former cell values will be put into this column.

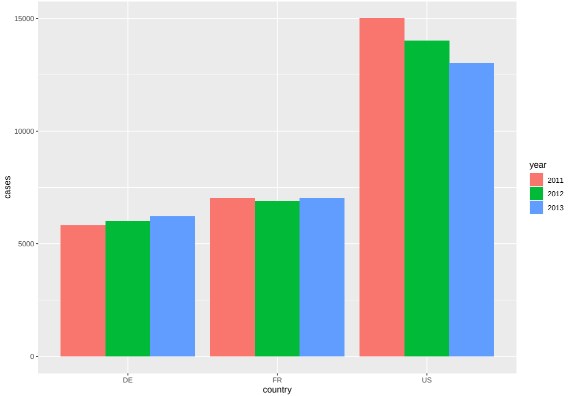
The names in pivot_longer() are arbitrary

```
pivot_longer(cases,
             cols = "2011":"2013",
             names_to = "when it happened",
             values_to = "how many")
```

```
# A tibble: 9 x 3
  country `when it happened` `how many`
  <chr>   <chr>                   <dbl>
1 FR      2011                     7000
2 FR      2012                     6900
3 FR      2013                     7000
4 DE      2011                     5800
5 DE      2012                     6000
6 DE      2013                     6200
7 US      2011                    15000
8 US      2012                    14000
9 US      2013                    13000
```

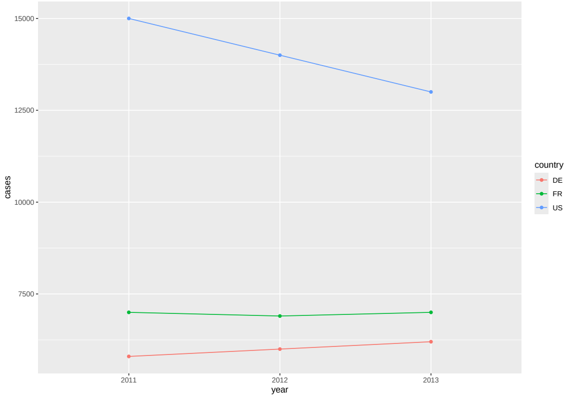
Plotting is better with the data in this form

```r
cases |>
  pivot_longer(cols = "2011":"2013",
               names_to = "year",
               values_to = "cases") |>
  ggplot(aes(x = country, y = cases, fill = year)) +
  geom_col(position = "dodge")
```



Another plot example

```r
cases |>
  pivot_longer(cols = "2011":"2013",
               names_to = "year",
               values_to = "cases") |>
  ggplot(aes(x = year, y = cases, color = country)) +
  geom_point() + geom_path(aes(group = country))
```



Additional methods for selecting columns to pivot

```r
cases |>
  pivot_longer(
    cols = starts_with("20"), # any column that starts with "20"
    names_to = "year",
    values_to = "cases"
)
```

```r
cases |>
  pivot_longer(
    cols = -country, # everything except the country column
    names_to = "year",
    values_to = "cases"
)
```

The pollution table

Here is another table with fictional data regarding polution measurements for different cities.

```
pollution
```

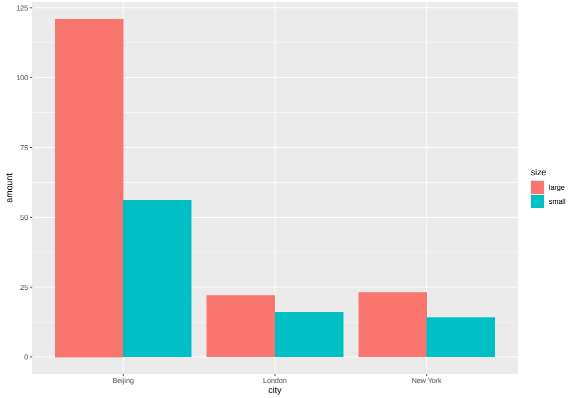
```
# A tibble: 6 x 3
  city     size  amount
  <chr>    <chr>  <dbl>
1 New York large     23
2 New York small     14
3 London   large     22
4 London   small     16
5 Beijing  large    121
6 Beijing  small     56
```

- Each row is a city-particle size combination. The observational unit is a pollution measurement for a specific city and particle size
- The variables are:
  - city
  - size
  - amount

Plotting is good with the current structure

I can map each column to an aesthetic attribute.

```
pollution |>
  ggplot(aes(x = city, y = amount, fill = size)) +
  geom_col(position = "dodge")
```



However, we can't do calculations with a single column

```
pollution
```

```
# A tibble: 6 x 3
  city     size  amount
  <chr>    <chr>  <dbl>
1 New York large     23
2 New York small     14
3 London   large     22
4 London   small     16
5 Beijing  large    121
6 Beijing  small     56
```

With the current structure, it's not possible (without a lot trouble) to calculate the difference between large and small particles in each city.

pivot_wider()

For this data, we need to use the function `pivot_wider()`. This function takes values in one column and pivots them across multiple columns (making the data wider). (Older versions called this `spread()`)

```
pollution |> pivot_wider(id_cols = city,
                         names_from = size,
                         values_from = amount)
```

```
# A tibble: 3 x 3
  city     large small
  <chr>    <dbl> <dbl>
1 New York    23    14
2 London      22    16
3 Beijing    121    56
```

pivot_wider()

```
pollution |> pivot_wider(id_cols = city,
              names_from = "size",
              values_from = "amount")
```

The pivot_wider() function takes in a few arguments:

- data is the name of the data.frame or tibble that we will pivot
- id_cols (optional) is the name of the column that identifies each observational unit
- names_from is the name of the column that has the names that will become column headers
- values_from is the name of the column that has the values

Calculations with mutate() for each city are now possible

```
pollution |>
  pivot_wider(id_cols = city,
              names_from = "size",
              values_from = "amount") |>
  mutate(difference = large - small,
         ratio = large / small)
```

```
# A tibble: 3 x 5
  city     large small difference ratio
  <chr>    <dbl> <dbl>      <dbl> <dbl>
1 New York    23    14          9  1.64
2 London      22    16          6  1.38
3 Beijing    121    56         65  2.16
```

Section 5

Warnings and Things to avoid

pivot_longer() duplicates any column that isn't pivoted

```
# A tibble: 3 x 4
  country `2011` `2012` `2013`
  <chr>    <dbl>  <dbl>  <dbl>
1 FR        7000   6900   7000
2 DE        5800   6000   6200
3 US       15000  14000  13000
```

Watch what happens if the columns I pivot are only "2012" and "2013".

```r
pivot_longer(cases,
             cols = "2012":"2013",
             names_to = "year",
             values_to = "cases")
```

pivot_longer() duplicates any column that isn't pivoted

The columns that are not pivoted are duplicated for the new rows created.

```
pivot_longer(cases,
             cols = "2012":"2013",
             names_to = "year",
             values_to = "cases")
```

```
# A tibble: 6 x 4
  country `2011` year  cases
  <chr>    <dbl> <chr> <dbl>
1 FR        7000 2012   6900
2 FR        7000 2013   7000
3 DE        5800 2012   6000
4 DE        5800 2013   6200
5 US       15000 2012  14000
6 US       15000 2013  13000
```

`pivot_wider()` is sensitive to spelling differences

If we have multiple spellings for New York (say New York and NYC), then each unique value gets its own row.

```
pollution2 <- pollution
pollution2[1,1] <- "NYC"
pollution2
```

```
# A tibble: 6 x 3
  city     size  amount
  <chr>    <chr>  <dbl>
1 NYC      large     23
2 New York small     14
3 London   large     22
4 London   small     16
5 Beijing  large    121
6 Beijing  small     56
```

Result

```
pivot_wider(pollution2,
            id_cols = city,
            names_from = "size",
            values_from = "amount")
```

```
# A tibble: 4 x 3
  city     large small
  <chr>    <dbl> <dbl>
1 NYC         23    NA
2 New York    NA    14
3 London      22    16
4 Beijing    121    56
```

Fill values

Sometimes you truly do have a scenario where you want to pivot wider and some entries do not exist. If you don't want `NA`s to show, you can specify a fill value.

Note: only use 0 if "missing" truly means zero. Otherwise leave NA or annotate the imputation.

```
pivot_wider(pollution2,
             id_cols = city,
             names_from = "size",
             values_from = "amount",
             values_fill = 0)
```

```
# A tibble: 4 x 3
  city     large small
  <chr>    <dbl> <dbl>
1 NYC         23     0
2 New York     0    14
3 London      22    16
4 Beijing    121    56
```

Another example of spelling differences

What will happen?

```
pollution2 <- pollution
pollution2[1,2] <- "LARGE"
pollution2
```

```
# A tibble: 6 x 3
  city     size  amount
  <chr>    <chr>  <dbl>
1 New York LARGE     23
2 New York small     14
3 London   large     22
4 London   small     16
5 Beijing  large    121
6 Beijing  small     56
```

Result

```
pivot_wider(pollution2,
             id_cols = city,
             names_from = "size",
             values_from = "amount")
```

```
# A tibble: 3 x 4
  city     LARGE small large
  <chr>    <dbl> <dbl> <dbl>
1 New York    23    14    NA
2 London      NA    16    22
3 Beijing     NA    56   121
```

pivot_longer() and pivot_wider() are inverse operations

```r
pollution
```

```
# A tibble: 6 x 3
  city     size  amount
  <chr>    <chr>  <dbl>
1 New York large     23
2 New York small     14
3 London   large     22
4 London   small     16
5 Beijing  large    121
6 Beijing  small     56
```

```r
w <- pivot_wider(pollution, id_cols = city, names_from = "size", values_from = "amount")
w
```

```
# A tibble: 3 x 3
  city     large small
  <chr>    <dbl> <dbl>
1 New York    23    14
2 London      22    16
3 Beijing    121    56
```

pivot_longer() and pivot_wider() are inverse operations

```r
w
```

```
# A tibble: 3 x 3
  city     large small
  <chr>    <dbl> <dbl>
1 New York    23    14
2 London      22    16
3 Beijing    121    56
```

```r
pivot_longer(w, cols = "large":"small", names_to = "size", values_to = "amount")
```

```
# A tibble: 6 x 3
  city     size  amount
  <chr>    <chr>  <dbl>
1 New York large     23
2 New York small     14
3 London   large     22
4 London   small     16
5 Beijing  large    121
6 Beijing  small     56
```

pivot_longer() and pivot_wider() are inverse operations

```
cases
```

```
# A tibble: 3 x 4
  country `2011` `2012` `2013`
  <chr>    <dbl>  <dbl>  <dbl>
1 FR        7000   6900   7000
2 DE        5800   6000   6200
3 US       15000  14000  13000
```

```
l <- pivot_longer(cases, cols = "2011":"2013", names_to = "year", values_to = "count")
l
```

```
# A tibble: 9 x 3
  country year  count
  <chr>   <chr> <dbl>
1 FR      2011   7000
2 FR      2012   6900
3 FR      2013   7000
4 DE      2011   5800
5 DE      2012   6000
6 DE      2013   6200
7 US      2011  15000
8 US      2012  14000
9 US      2013  13000
```

pivot_longer() and pivot_wider() are inverse operations

```
l
```

```
# A tibble: 9 x 3
  country year  count
  <chr>   <chr> <dbl>
1 FR      2011   7000
2 FR      2012   6900
3 FR      2013   7000
4 DE      2011   5800
5 DE      2012   6000
6 DE      2013   6200
7 US      2011  15000
8 US      2012  14000
9 US      2013  13000
```

```
pivot_wider(l, names_from = "year", values_from = "count")
```

```
# A tibble: 3 x 4
  country `2011` `2012` `2013`
  <chr>    <dbl>  <dbl>  <dbl>
1 FR        7000   6900   7000
2 DE        5800   6000   6200
3 US       15000  14000  13000
```