Basic data manipulation

Lecture 3

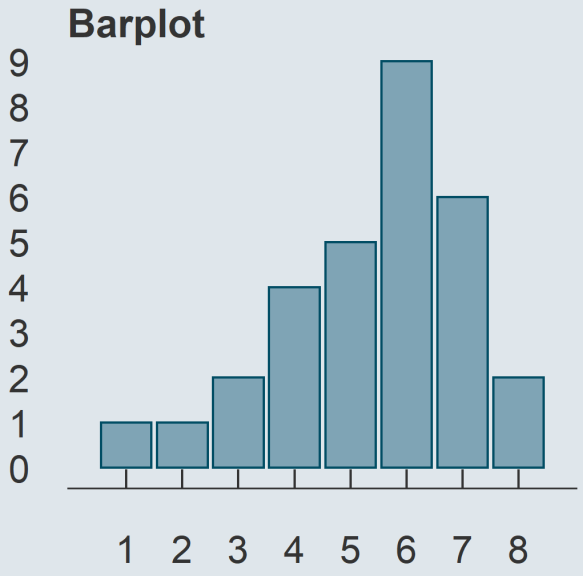
Louis SIRUGUE

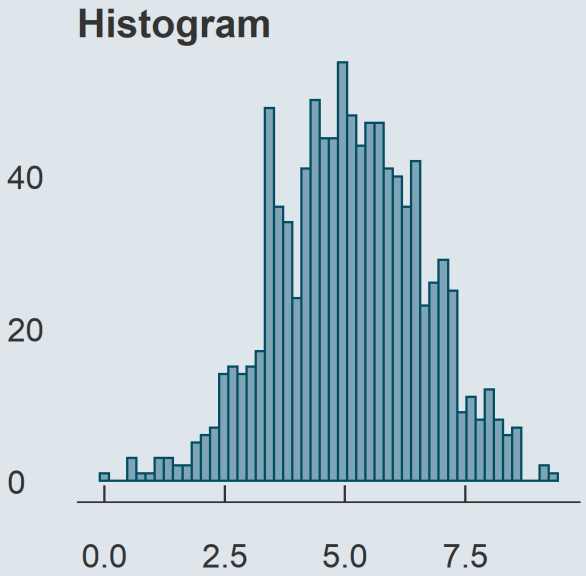
CPES 2 - Fall 2022

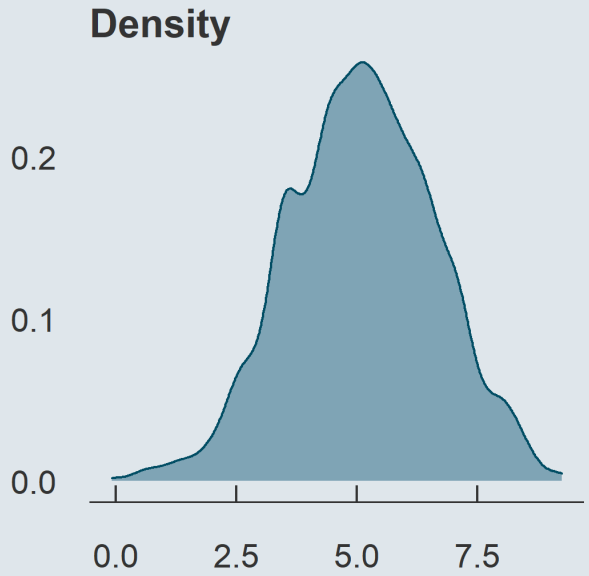
Last time we saw

1. Distributions

- The **distribution** of a variable documents all its possible values and how frequent they are





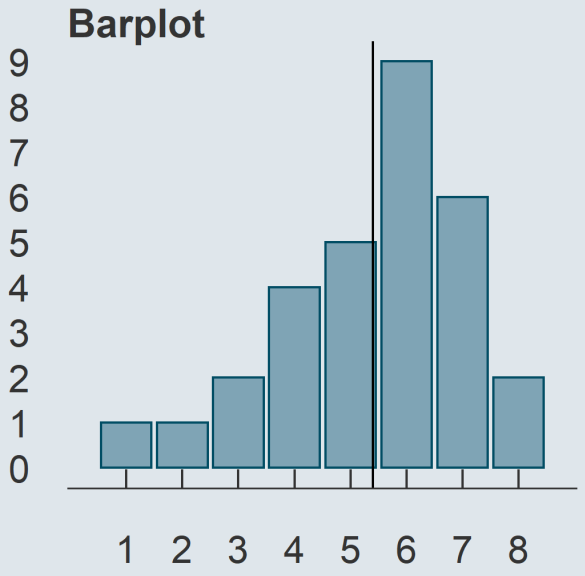


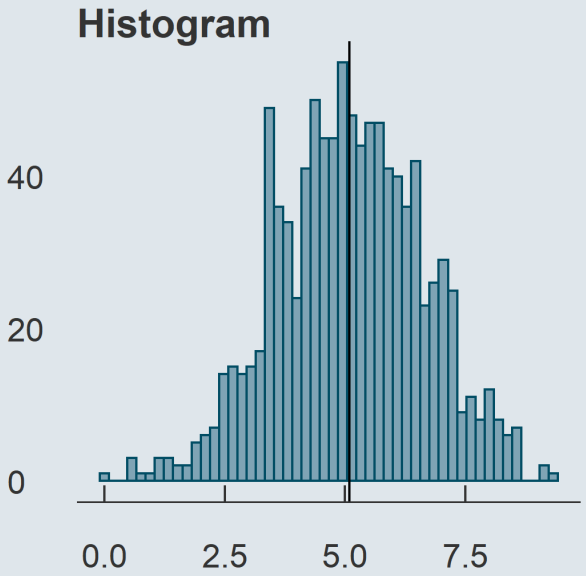
- We can describe a distribution with:

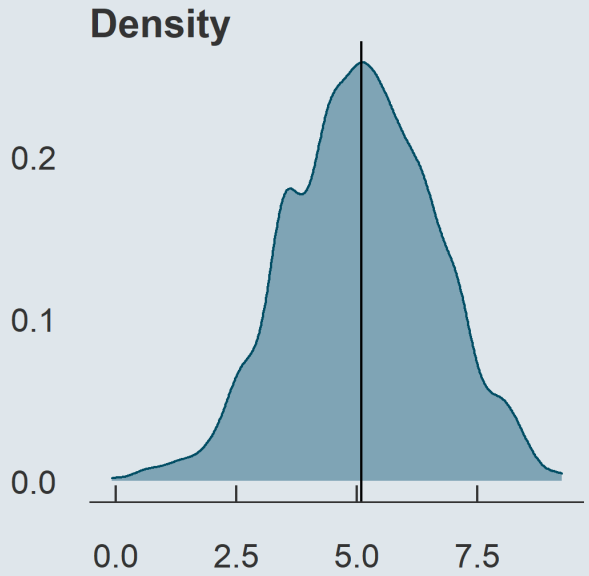
Last time we saw

1. Distributions

- The **distribution** of a variable documents all its possible values and how frequent they are





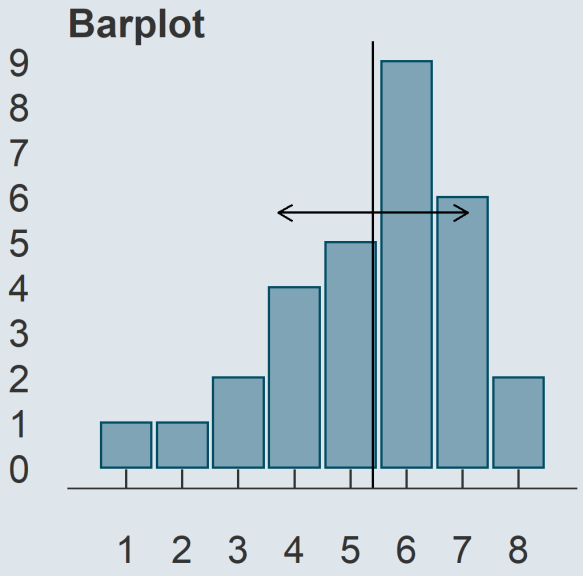


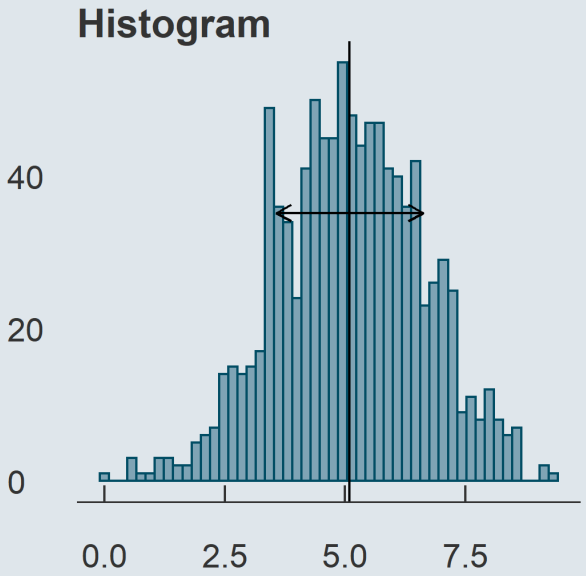
- We can describe a distribution with:
  - Its **central tendency**

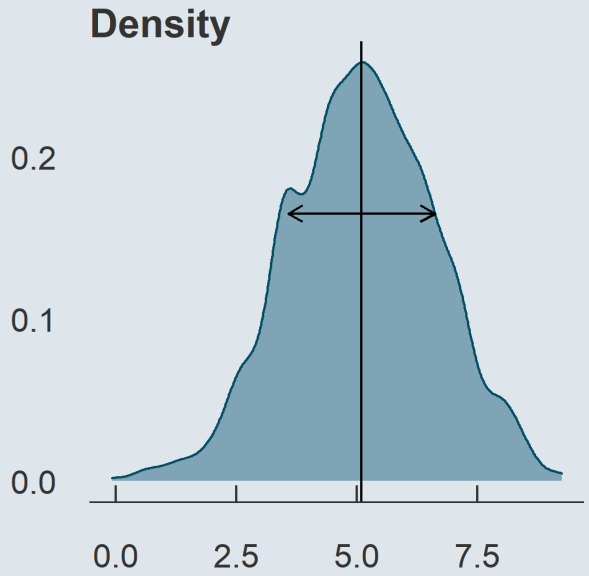
Last time we saw

1. Distributions

- The **distribution** of a variable documents all its possible values and how frequent they are







- We can describe a distribution with:
  - Its **central tendency**
  - And its **spread**

Last time we saw

2. Central tendency

- The **mean** is the sum of all values divided by the number of observations

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

- The **median** is the value that divides the (sorted) distribution into two groups of equal size

$$\text{Med}(x) = \begin{cases} x[\frac{N+1}{2}] & \text{if } N \text{ is odd} \\ \frac{x[\frac{N}{2}]+x[\frac{N}{2}+1]}{2} & \text{if } N \text{ is even} \end{cases}$$

3. Spread

- The **standard deviation** is square root of the average squared deviation from the mean

$$\text{SD}(x) = \sqrt{\text{Var}(x)} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

- The **interquartile range** is the difference between the maximum and the minimum value from the middle half of the distribution

$$\text{IQR} = Q_3 - Q_1$$

Last time we saw

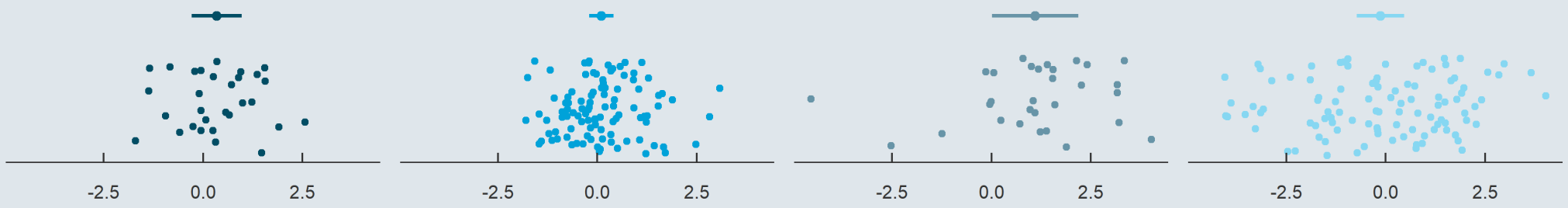
4. Inference

- In Statistics, we view variables as a given realization of a **data generating process**
  - Hence, the **mean** is what we call an **empirical moment**, which is an **estimation**...
  - ... of the **expected value**, the **theoretical moment** of the DGP we're interested in

- To know how confident we can be in this estimation, we need to compute a **confidence interval**

$$[\bar{x} - t_{n-1,\,97.5\%} \times \frac{\mathrm{SD}(x)}{\sqrt{n}};\ \bar{x} + t_{n-1,\,97.5\%} \times \frac{\mathrm{SD}(x)}{\sqrt{n}}]$$

  - It gets **larger** as the **variance** of the distribution of x increases
  - And gets **smaller** as the **sample size** n increases



Today we learn how to manipulate data

1. Datasets

- 1.1. What is a dataset
- 1.2. Import and eyeball data
- 1.3. Subset data

2. The dplyr grammar

- 2.1. Basic functions
- 2.2. Group by and summarise
- 2.3. Merge and append data
- 2.4. Reshape data

3. Wrap up!

Today we learn how to manipulate data

1. Datasets

- 1.1. What is a dataset
- 1.2. Import and eyeball data
- 1.3. Subset data

1. Datasets

1.1. What is a dataset?

→ **Datasets** are typically tables in which each **row** corresponds to an **observation** and each **column** to a **variable**

Observations can be:

- Individuals
- Countries
- Years

Variables can be:

- Age
- GDP
- Temperature

Excerpt of a dataset on cereals

name	calories	protein	fat	sodium	fiber	vitamins
100% Bran	70	4	1	130	10.0	25
100% Natural Bran	120	3	5	15	2.0	0
All-Bran	70	4	1	260	9.0	25
All-Bran with Extra Fiber	50	4	0	140	14.0	25
Almond Delight	110	2	2	200	1.0	25
Apple Cinnamon Cheerios	110	2	2	180	1.5	25
Apple Jacks	110	2	0	125	1.0	25
Basic 4	130	3	2	210	2.0	25

1. Datasets

1.1. What is a dataset?

- Each column of a dataset can be seen as a **vector** whose **n^{th} element** is about the **n^{th} individual**
  - This is how we can create a dataset: by assigning vectors to variable names in a data object

```
departments <- data.frame(code = 1:4,
                          department = c("Ain", "Aisne", "Allier", "Alpes-de-Haute-Provence"),
                          capital = c("Bourg-en-Bresse", "Laon", "Moulin", "Digne-les-bains"))

print(departments, row.names = F)
```

```
##  code              department         capital
##     1                     Ain Bourg-en-Bresse
##     2                   Aisne            Laon
##     3                  Allier          Moulin
##     4 Alpes-de-Haute-Provence Digne-les-bains
```

- This illustrates what datasets are made of, but the point is not to write datasets ourselves

➜ We need to import data in R

1. Datasets

1.2. Import and eyeball data

- There are **various formats** of datasets, and as many ways to import them in R
  - A very common format is the **.csv** (for *Comma Separated Values*), which basically looks like that:

```
1,Ain,Bourg-en-Bresse
2,Aisne,Laon
3,Allier,Moulin
4,Alpes-de-Haute-Provence,Digne-les-bains
```

To **import** csv data, you can use `read.csv()`. This function has **many parameters** you can change to import the data the way you want. Here are a few of them:

- `skip`: how many **lines to skip** before reading the data
- `header`: whether the first row contains **variables names**
- `sep`: the **character** that **separates** each observations in the csv (usually `","` or `";"`)
- `fileEncoding`: if the data contain **special characters** you should set the right encoding

```
iris <- read.csv("iris.csv", skip = 0, header = TRUE, sep = ",", encoding = "UTF-8")
```

1. Datasets

1.2. Import and eyeball data

- The `head` function prints the **first rows** of a dataset

```
head(iris, 12) # Show the first 12 rows of the data
```

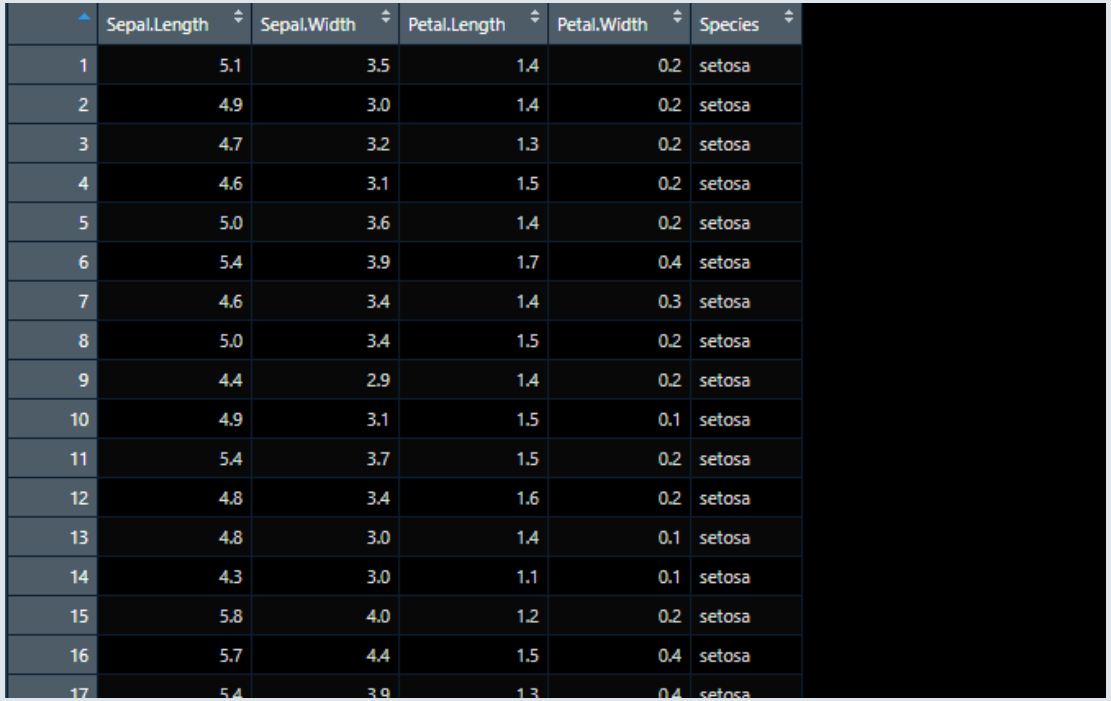
```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          1.4         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 4           4.6         3.1          1.5         0.2  setosa
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
## 11          5.4         3.7          1.5         0.2  setosa
## 12          4.8         3.4          1.6         0.2  setosa
```

1. Datasets

1.2. Import and eyeball data

- The `view` function opens the **data in a new tab** in the code panel. It should look like that:

```
view(iris)
```



	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa
11	5.4	3.7	1.5	0.2	setosa
12	4.8	3.4	1.6	0.2	setosa
13	4.8	3.0	1.4	0.1	setosa
14	4.3	3.0	1.1	0.1	setosa
15	5.8	4.0	1.2	0.2	setosa
16	5.7	4.4	1.5	0.4	setosa
17	5.4	3.9	1.3	0.4	setosa

1. Datasets

1.2. Import and eyeball data

- You can obtain the **dimension** of the data with the `dim()` function

```
dim(iris)
```

```
## [1] 150   5
```

The data has 150 rows and 5 columns

- You can access to the **variable names** with the `names()` function

```
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

1. Datasets

1.2. Import and eyeball data

- The function `summary()` allows you to get a concise **description of each variable** in your data

```
summary(iris, digits = 2) # Choose how to round numeric values with 'digits'
```

```
##   Sepal.Length  Sepal.Width   Petal.Length  Petal.Width    Species
##  Min.   :4.3   Min.   :2.0   Min.   :1.0   Min.   :0.1   Length:150
##  1st Qu.:5.1   1st Qu.:2.8   1st Qu.:1.6   1st Qu.:0.3   Class :character
##  Median :5.8   Median :3.0   Median :4.3   Median :1.3   Mode  :character
##  Mean   :5.8   Mean   :3.1   Mean   :3.8   Mean   :1.2
##  3rd Qu.:6.4   3rd Qu.:3.3   3rd Qu.:5.1   3rd Qu.:1.8
##  Max.   :7.9   Max.   :4.4   Max.   :6.9   Max.   :2.5
```

➜ What we learn:

- `Petal`/`Sepal.Lengths`/`Widths` are **numeric** while `Species` is a **character** variable
- From the **mean** and the **range** of numeric variables:
  - Petals tend to be smaller than sepals (for irises)
  - The range of numeric variables is quite large

1. Datasets

1.2. Import and eyeball data

- In R you can handle datasets in **different formats**:
  - data.frame
  - tibble
  - matrix
  - data.table

- All these formats have their specificities
  - In this course we'll focus on **data.frames** and **tibbles** (very similar)
- One specificity of the **tibble** can be observed when printing the data:
  - It will display its **dimensions** (#rows $\times$ #columns) and the **class** of each variable under its name

```
head(as_tibble(iris), 2)
```

```
## # A tibble: 2 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <chr>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            1.4         0.2 setosa
```

1. Datasets

1.3. Subset data: Extract values

- Just as with vectors, you can **access elements** of the data using **[]**
  - But while vectors have 1 dimension, datasets have **two dimensions**: rows and columns
  - To access specific cells of the data, you must indicate the **row number(s)** and the **column number(s)** *separated by a comma in the brackets*

```
iris[5, 2] # Fifth observation of the second variable
```

```
## [1] 3.6
```

```
iris[2, ] # Second observation of each variable
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 2          4.9           3          1.4         0.2  setosa
```

```
iris[c(3, 6), 4] # Observations 3 and 6 of the fourth variable
```

```
## [1] 0.2 0.4
```

1. Datasets

1.3. Subset data: Get variables

- The `$` allows to **access a variable** of the data

```
iris$Sepal.Length
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
##  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
##  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
##  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
##  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
##  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

- Alternative solutions include using **double brackets** or the **column index**

```
iris[["Sepal.Length"]]
iris[, 1]
```

1. Datasets

1.3. Subset data: Conditional subsetting

- A **logical vector** (`T` and `F` / a condition) **before the comma** will select every **row** that meets this condition

```
iris[iris$Petal.Length > 6.5, ]
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 106          7.6         3.0          6.6         2.1 virginica
## 118          7.7         3.8          6.7         2.2 virginica
## 119          7.7         2.6          6.9         2.3 virginica
## 123          7.7         2.8          6.7         2.0 virginica
```

- A **logical vector** (`T` and `F` / a condition) **after the comma** will select every **column** that meets this condition

```
iris[iris$Petal.Length > 6.5, names(iris) %in% c("Petal.Length", "Species")]
```

```
##     Petal.Length   Species
## 106          6.6 virginica
## 118          6.7 virginica
## 119          6.9 virginica
## 123          6.7 virginica
```

Overview

1. Datasets ✓

- 1.1. What is a dataset
- 1.2. Import and eyeball data
- 1.3. Subset data

2. The dplyr grammar

- 2.1. Basic functions
- 2.2. Group by and summarise
- 2.3. Merge and append data
- 2.4. Reshape data

3. Wrap up!

Overview

1. Datasets ✓

- 1.1. What is a dataset
- 1.2. Import and eyeball data
- 1.3. Subset data

2. The dplyr grammar

- 2.1. Basic functions
- 2.2. Group by and summarise
- 2.3. Merge and append data
- 2.4. Reshape data

2. The `dplyr` grammar

2.1. Basic functions

`dplyr` is a **grammar** of data manipulation, providing very **user-friendly functions** to handle the most common **data manipulation** tasks. The functions you will probably use the most are the following:

- `mutate()`: add/modify variables
- `select()`: keep/drop variables (columns)
- `filter()`: keep/drop observations (rows)
- `arrange()`: sort rows according to the values of given variable(s)
- `summarise()`: aggregate the data into descriptive statistics

- A very handy **operator** to use with the **dplyr** grammar is the **pipe %>%**
  - You can basically read **a %>% b()** as *"apply function b() to object a"*
  - With this operator you can easily **chain the operations** you apply to an object

2. The `dplyr` grammar

2.1. Basic functions

```
iris %>%
  # Generate an new variable, the product of sepal length and width
  mutate(product = Sepal.Length * Sepal.Width) %>%
  # Keep only this variable and the species in the dataset
  select(product, Species) %>%
  # Keep only observations from the virginica species
  filter(Species == "virginica") %>%
  # Arrange rows by decreasing value of the product variable
  arrange(-product) %>%
  # Show the first 5 observations
  head(5)
```

```
##   product   Species
## 1   30.02 virginica
## 2   29.26 virginica
## 3   25.92 virginica
## 4   23.10 virginica
## 5   23.04 virginica
```

2. The `dplyr` grammar

⚠ Be careful when chaining operations with pipes, it's easy to get mixed up! ⚠

- **Don't code blindfolded, `view()` your data at each step** to make sure that it goes the way you intend it to go
- Try to work on the subset of variables you actually need to make things easier

```
iris %>%
  mutate(product = Sepal.Length * Sepal.Width) %>%
  view()
```

```
iris %>%
  mutate(product = Sepal.Length * Sepal.Width) %>%
  select(product, Species) %>%
  view()
```

```
iris %>%
  mutate(product = Sepal.Length * Sepal.Width) %>%
  select(product, Species) %>%
  filter(Species == "virginica") %>%
  view()
```

2. The `dplyr` grammar

2.1. Basic functions: common mutate usage

- Generate a variable as a function of other variables

```
iris <- iris %>% mutate(new_var = (Sepal.Length * Sepal.Width) / 2)
```

- Add a variable from elsewhere (dimensions must match)

```
iris <- iris %>% mutate(new_var = 1:nrow(iris))
```

- Use `ifelse(condition, value if condition is met, value if not)`

```
iris <- iris %>% mutate(new_var = ifelse(Species == "virginica", 1, 0))
```

- Use `case_when()` when there are more than 2 cases

```
iris <- iris %>% mutate(new_var = case_when(Species == "versicolor" ~ "VER",
                                            Species == "virginica" ~ "VIR",
                                            Species == "setosa" ~ "SET"))
```

2. The `dplyr` grammar

2.2. Group by and summarise

The `group_by()` function allows to modify the data **group by group** rather than on all observations

- Imagine that you are interested in **comparing**
  - The average sepal width **by iris species**
  - And the average sepal width **overall**

```
iris_groups <- iris %>% # Let's save our modifications in a new dataset
  mutate(pop_mean = mean(Sepal.Width)) %>% # First compute the mean of all observations
  group_by(Species) %>% # Then group par specie
  mutate(group_mean = mean(Sepal.Width)) # And compute the mean
```

- Let's view our data to see how it went

```
view(iris_groups)
```

2. The `dplyr` grammar

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	pop_mean	group_mean
41	5.0	3.5	1.3	0.3	setosa	3.057333	3.428
42	4.5	2.3	1.3	0.3	setosa	3.057333	3.428
43	4.4	3.2	1.3	0.2	setosa	3.057333	3.428
44	5.0	3.5	1.6	0.6	setosa	3.057333	3.428
45	5.1	3.8	1.9	0.4	setosa	3.057333	3.428
46	4.8	3.0	1.4	0.3	setosa	3.057333	3.428
47	5.1	3.8	1.6	0.2	setosa	3.057333	3.428
48	4.6	3.2	1.4	0.2	setosa	3.057333	3.428
49	5.3	3.7	1.5	0.2	setosa	3.057333	3.428
50	5.0	3.3	1.4	0.2	setosa	3.057333	3.428
51	7.0	3.2	4.7	1.4	versicolor	3.057333	2.770
52	6.4	3.2	4.5	1.5	versicolor	3.057333	2.770
53	6.9	3.1	4.9	1.5	versicolor	3.057333	2.770
54	5.5	2.3	4.0	1.3	versicolor	3.057333	2.770
55	6.5	2.8	4.6	1.5	versicolor	3.057333	2.770
56	5.7	2.8	4.5	1.3	versicolor	3.057333	2.770
57	6.3	3.3	4.7	1.6	versicolor	3.057333	2.770
58	4.9	2.4	3.3	1.0	versicolor	3.057333	2.770
59	6.6	2.9	4.6	1.3	versicolor	3.057333	2.770
60	5.2	2.7	3.9	1.4	versicolor	3.057333	2.770
61	5.0	2.0	3.5	1.0	versicolor	3.057333	2.770
62	5.9	3.0	4.2	1.5	versicolor	3.057333	2.770

2. The `dplyr` grammar

2.2. Group by and summarise

- For each observation we now have the **average** sepal width of its **specie** and that of the whole **population**
  - But these results are **not conveyed clearly** because we still have individual-level info in the data
  - We should keep **only** the **aggregated** variables with one row per specie

```
iris_groups %>%
  select(Species, pop_mean, group_mean) %>% # Select only the variables of interest
  unique() # Drop duplicate rows
```

```
## # A tibble: 3 x 3
## # Groups:   Species [3]
##   Species    pop_mean group_mean
##   <chr>         <dbl>      <dbl>
## 1 setosa         3.06       3.43
## 2 versicolor     3.06       2.77
## 3 virginica      3.06       2.97
```

- We actually summarized the data into **descriptive statistics**
  - This is precisely what the **summarise()** function is made for

2. The `dplyr` grammar

2.2. Group by and summarise

- The **summarise()** function was made to **do it all** that by itself:
  - Step 1: Indicate at which level you want to aggregate the data by putting the **grouping variable** in **group_by()**
  - Step 2: Generate the aggregate variables by specifying **which variable** you want **to aggregate** and what **group-level information** you want about it *(min, max, sum, number of observations, ...)*

```
iris %>% group_by(Species) %>%
  summarise(mean_length = mean(Sepal.Length),
            min_length = min(Sepal.Length),
            max_length = max(Sepal.Length),
            sum_width = sum(Sepal.Width),
            nb_obs = n())
```

```
## # A tibble: 3 x 6
##   Species    mean_length min_length max_length sum_width nb_obs
##   <chr>            <dbl>      <dbl>      <dbl>     <dbl>  <int>
## 1 setosa            5.01        4.3        5.8      171.     50
## 2 versicolor        5.94        4.9        7        138.     50
## 3 virginica         6.59        4.9        7.9      149.     50
```

2. The `dplyr` grammar

2.2. Group by and summarise

Note that **`group_by()`** generally applies to **all operations until** you apply the **`ungroup()`** function

- After computing the average sepal length by species, you may want to get the maximum value of that average:

```
iris %>%
  group_by(Species) %>% # Group by species
  mutate(mean_length = mean(Sepal.Length)) %>% # Compute the mean (by species)
  select(Species, mean_length) %>% # Select the two variables we're interested in
  unique() %>% # Drop duplicate row
  mutate(forgot_ungroup = max(mean_length)) %>% # Compute max without ungroup
  ungroup() %>% # Ungroup
  mutate(did_not_forget = max(mean_length)) # Compute max after ungroup
```

```
## # A tibble: 3 x 4
##   Species    mean_length forgot_ungroup did_not_forget
##   <chr>            <dbl>          <dbl>          <dbl>
## 1 setosa            5.01           5.01           6.59
## 2 versicolor        5.94           5.94           6.59
## 3 virginica         6.59           6.59           6.59
```

If you do not ungroup the data, the maximum value is computed *by* specie and not *across* species.

Practice

1) Import `starbucks.csv` using the following command and **see what's wrong**

```
starbucks <- read.csv("YOUR_DIRECTORY/starbucks.csv")
```

*Your directory must look like "C:/Users/...", **make sure to use / instead of ***

2) Set the `sep` and `encoding` **arguments** of `read.csv` function to import the data **correctly**

3) Use `summarise()` to **compute** for each **beverage category** the average number of **calories** and the number of different **declinations** *(there is 1 row per declination)*

4) Create a **subset** of the data called `maxcal` containing the **variables** `Beverage_category`, `Beverage_prep`, and `Calories`, for the **10 observations** with the **highest calorie** values

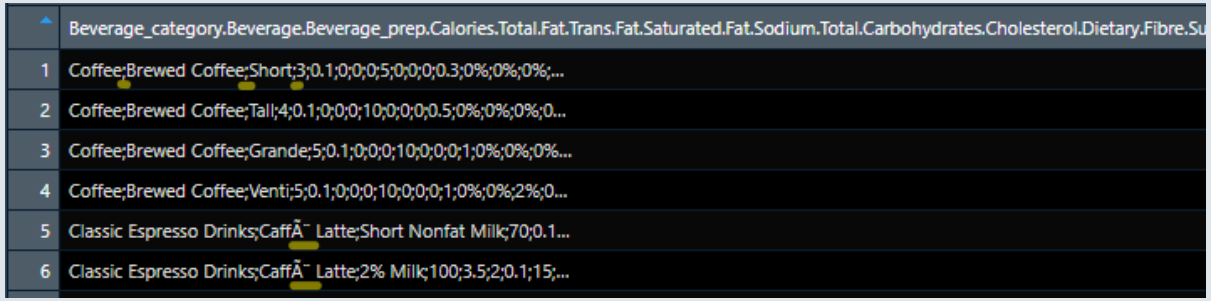
You can use the row_number() function which gives the row numbers as a vector

You've got 10 minutes!

Solution

1) Import `starbucks.csv` using the following command and see what's wrong

```
read.csv("starbucks.csv") %>% view()
```



	Beverage_category.Beverage.Beverage_prep.Calories.Total.Fat.Trans.Fat.Saturated.Fat.Sodium.Total.Carbohydrates.Cholesterol.Dietary.Fibre.Su
1	Coffee;Brewed Coffee;Short;3;0.1;0;0;0;5;0;0;0;0.3;0%;0%;0%;...
2	Coffee;Brewed Coffee;Tall;4;0.1;0;0;0;10;0;0;0;0.5;0%;0%;0%;0...
3	Coffee;Brewed Coffee;Grande;5;0.1;0;0;0;10;0;0;0;1;0%;0%;0%...
4	Coffee;Brewed Coffee;Venti;5;0.1;0;0;0;10;0;0;0;1;0%;0%;2%;0...
5	Classic Espresso Drinks;CaffÃ¨ Latte;Short Nonfat Milk;70;0.1...
6	Classic Espresso Drinks;CaffÃ¨ Latte;2% Milk;100;3.5;2;0.1;15;...

2) Set the `sep` and `encoding` arguments of `read.csv` function to import the data correctly

```
starbucks <- read.csv("starbucks.csv", sep = ";", encoding = "UTF-8")
head(starbucks[4:5, 1:5])
```

```
##         Beverage_category      Beverage     Beverage_prep Calories Total.Fat
## 4                  Coffee Brewed Coffee            Venti        5       0.1
## 5 Classic Espresso Drinks   Caffè Latte Short Nonfat Milk       70       0.1
```

Solution

3) Use `summarise()` to compute for each beverage category the average number of calories and the number of different declinations (there is 1 row per declination)

```
starbucks %>%
  group_by(Beverage_category) %>%
  summarise(Declinations = n(),
            Mean_cal = mean(Calories))
```

```
## # A tibble: 9 x 3
##   Beverage_category                Declinations Mean_cal
##   <chr>                                   <int>    <dbl>
## 1 Classic Espresso Drinks                    58   140.
## 2 Coffee                                      4     4.25
## 3 Frappuccino® Blended Coffee                36   277.
## 4 Frappuccino® Blended Crème                 13   233.
## 5 Frappuccino® Light Blended Coffee          12   162.
## 6 Shaken Iced Beverages                      18   114.
## 7 Signature Espresso Drinks                  40   250
## 8 Smoothies                                   9   282.
## 9 Tazo® Tea Drinks                           52   177.
```

Solution

4) Create a subset of the data called `maxcal` containing the variables `Beverage_category`, `Beverage_prep`, and `Calories`, for the 10 observation with the highest calorie values

```
maxcal <- starbucks %>%
  arrange(-Calories) %>%
  select(Beverage_category, Beverage_prep, Calories) %>%
  filter(row_number() <= 10)

maxcal
```

```
##              Beverage_category     Beverage_prep Calories
## 1    Signature Espresso Drinks           2% Milk      510
## 2    Signature Espresso Drinks           Soymilk      460
## 3  Frappuccino® Blended Coffee        Whole Milk      460
## 4    Signature Espresso Drinks Venti Nonfat Milk      450
## 5              Tazo® Tea Drinks           2% Milk      450
## 6  Frappuccino® Blended Coffee           Soymilk      430
## 7  Frappuccino® Blended Coffee Venti Nonfat Milk      420
## 8    Signature Espresso Drinks           2% Milk      400
## 9              Tazo® Tea Drinks           Soymilk      390
## 10 Frappuccino® Blended Coffee        Whole Milk      390
```

2. The `dplyr` grammar

2.3. Merge and append data

- Research projects often imply to **combine data** from different sources
  - To **add observations** (append rows)
  - To **add variables** (merge columns)

Dataset 1 on attainment

country	year	share_tertiary
FRA	2015	44.68760
GBR	2015	49.94341
USA	2015	46.51771

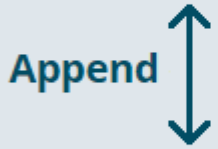
2. The `dplyr` grammar

2.3. Merge and append data

- Research projects often imply to **combine data** from different sources
  - Either to **add observations** (append rows)
  - Either to **add variables** (merge columns)

Dataset 1 on attainment

country	year	share_tertiary
FRA	2015	44.68760
GBR	2015	49.94341
USA	2015	46.51771



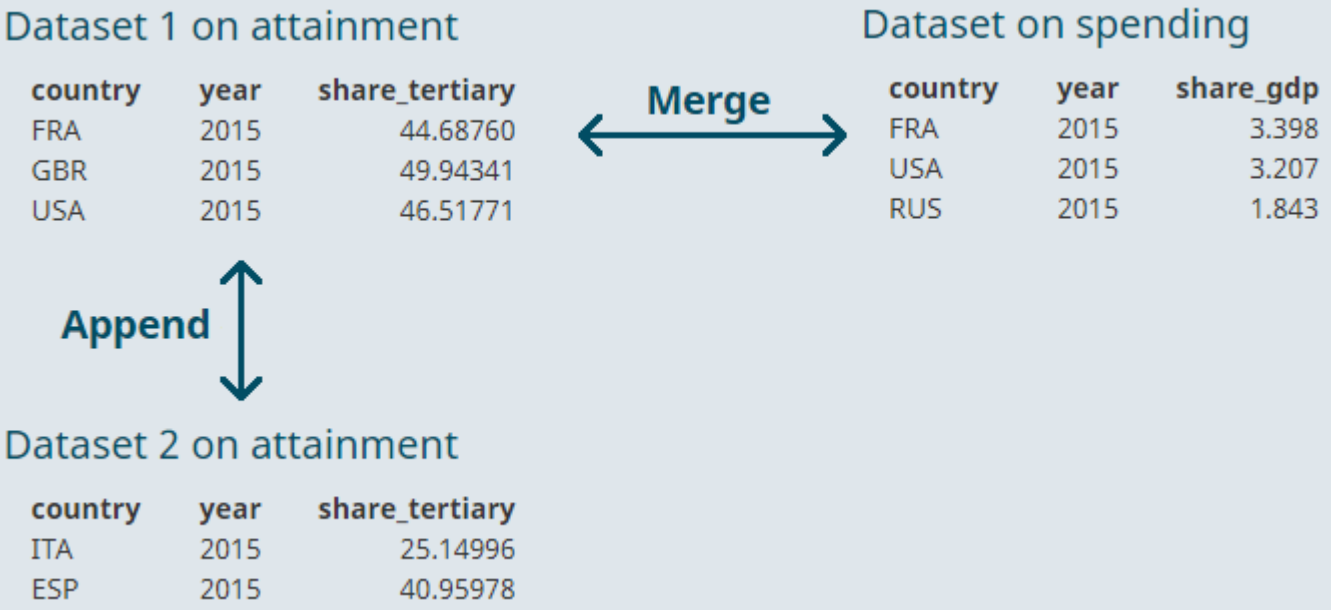
Dataset 2 on attainment

country	year	share_tertiary
ITA	2015	25.14996
ESP	2015	40.95978

2. The `dplyr` grammar

2.3. Merge and append data

- Research projects often imply to **combine data** from different sources
  - Either to **add observations** (append rows)
  - Either to **add variables** (merge columns)



2. The `dplyr` grammar

2.3. Append data: The `bind_rows()` function

```
read.csv("attainment_FR_UK_US.csv")
```

```
##   country year share_tertiary
## 1     FRA 2015       44.68760
## 2     GBR 2015       49.94341
## 3     USA 2015       46.51771
```

```
read.csv("attainment_IT_SP.csv")
```

```
##   country year share_tertiary
## 1     ITA 2015       25.14996
## 2     ESP 2015       40.95978
```

```
attainment <- read.csv("attainment_FR_UK_US.csv") %>%
  bind_rows(read.csv("attainment_IT_SP.csv"))
attainment
```

```
##   country year share_tertiary
## 1     FRA 2015       44.68760
## 2     GBR 2015       49.94341
## 3     USA 2015       46.51771
## 4     ITA 2015       25.14996
## 5     ESP 2015       40.95978
```

Variables in the two datasets should be the same:

- **Same name**
- **Same class**

2. The `dplyr` grammar

2.3. Join data: `*_join()` functions

- Join functions all work the same way:
  - A **dataset A** with a **variable X** and other variables
  - A **dataset B** with a **variable X** and other variables
  - X is the common variable, so datasets will be **joined *by* X**

The 4 main join functions

Function	For X in A & B	For X in A only	For X in B only	Summary
A %>% left_join(B, by = "X")	Kept	Kept	Dropped	Only keeps what's in A
A %>% right_join(B, by = "X")	Kept	Dropped	Kept	Only keeps what's in B
A %>% inner_join(B, by = "X")	Kept	Dropped	Dropped	Only keeps what's common
A %>% full_join(B, by = "X")	Kept	Kept	Kept	Keeps everything

2. The `dplyr` grammar

⚠ Beware of NAs! ⚠

- When you have **values** of X that are **not common** to both datasets
  - Any other join than the inner_join() will **generate NAs**

```
attainment %>% full_join(read.csv("spending.csv"), by = "country")
```

```
##   country year.x share_tertiary year.y share_gdp
## 1     FRA   2015       44.68760   2015     3.398
## 2     GBR   2015       49.94341     NA        NA
## 3     USA   2015       46.51771   2015     3.207
## 4     ITA   2015       25.14996     NA        NA
## 5     ESP   2015       40.95978     NA        NA
## 6     RUS     NA             NA   2015     1.843
```

- Any variable from `A` (`B`) other than those stated in `by=` will be `NA` for observations that are only in `B` (`A`)
- This holds when a variable that is not mentioned in the `by=` argument appears in both datasets:
  - In that case, R adds a data-specific suffix to the names and keeps them both
  - The variable from `B` (here `year.y`) will be `NA` for observations that are only in `A` only (here `GBR`, `ITA`, `ESP`)

2. The `dplyr` grammar

2.3. Join data: example

```
attainment %>% left_join(read.csv("spending.csv"), by = "country")
```

```
##   country year.x share_tertiary year.y share_gdp
## 1     FRA   2015       44.68760   2015     3.398
## 2     GBR   2015       49.94341     NA        NA
## 3     USA   2015       46.51771   2015     3.207
## 4     ITA   2015       25.14996     NA        NA
## 5     ESP   2015       40.95978     NA        NA
```

```
attainment %>% right_join(read.csv("spending.csv"), by = "country")
```

```
##   country year.x share_tertiary year.y share_gdp
## 1     FRA   2015       44.68760   2015     3.398
## 2     USA   2015       46.51771   2015     3.207
## 3     RUS     NA             NA   2015     1.843
```

➜ What would be the result of an inner_join() here?

2. The `dplyr` grammar

2.4. Reshape data

- It is important to be able to **switch from** the ***long*** **to** the ***wide*** format and conversely
  - Some computations should be done in one format or the other

Wide format

country	year	share_tertiary	share_gdp
FRA	2015	44.69	3.40
USA	2015	46.52	3.21

Long format

country	year	Variable	Value
FRA	2015	share_tertiary	44.69
FRA	2015	share_gdp	3.40
USA	2015	share_tertiary	46.52
USA	2015	share_gdp	3.21

2. The `dplyr` grammar

2.4. Reshape data: From wide to long with pivot_longer()

```
wide <- attainment %>%
  inner_join(read.csv("spending.csv") %>% select(-year),
             by = "country")
wide
```

```
##   country year share_tertiary share_gdp
## 1     FRA 2015       44.68760     3.398
## 2     USA 2015       46.51771     3.207
```

➜ *Pivoting to **long format** can be seen as putting **variables on top of each other** rather side to side*

- We need to indicate:
  - **Which variables to stack**
  - The **name of** the variable in which we want the **values** of the stacked variables to be stored
  - The **name of** the variable that will indicate to which **variable** corresponds each value

2. The `dplyr` grammar

2.4. Reshape data: From wide to long with pivot_longer()

```
long <- wide %>%
                # Which variable to should be stacked
  pivot_longer(c(share_tertiary, share_gdp),
                # Where their values should be stored
                values_to = "Value",
                # Where to store which variable corresponds each value
                names_to = "Variable")
long
```

```
## # A tibble: 4 x 4
##   country  year Variable       Value
##   <chr>   <int> <chr>          <dbl>
## 1 FRA      2015 share_tertiary 44.7
## 2 FRA      2015 share_gdp       3.40
## 3 USA      2015 share_tertiary 46.5
## 4 USA      2015 share_gdp       3.21
```

2. The `dplyr` grammar

2.4. Reshape data: From long to wide with pivot_wider()

- To **pivot in a wide** format we need to indicate:
  - **Which variable** contains **values** of the variables we want to put side to side
  - **Which variable** indicates which **variable** correspond to each value

```
wide <- long %>%
               # Where the values are
  pivot_wider(values_from = "Value",
               # Where the corresponding variable names are
               names_from = "Variable")
wide
```

```
## # A tibble: 2 x 4
##   country  year share_tertiary share_gdp
##   <chr>   <int>          <dbl>     <dbl>
## 1 FRA      2015           44.7      3.40
## 2 USA      2015           46.5      3.21
```

Overview

1. Datasets ✓

- 1.1. What is a dataset
- 1.2. Import and eyeball data
- 1.3. Subset data

2. The dplyr grammar ✓

- 2.1. Basic functions
- 2.2. Group by and summarise
- 2.3. Merge and append data
- 2.4. Reshape data

3. Wrap up!

3. Wrap up!

Read data

```
starbucks <- read.csv("C:/User/Documents/folder/starbucks.csv", sep = ";", encoding = "UTF-8")
```

***➜ Make sure to use / and not ***

Chaining operations

```
starbucks %>%
  arrange(-Calories) %>%
  select(Beverage_category, Beverage_prep, Calories) %>%
  filter(row_number() <= 3)
```

```
##             Beverage_category Beverage_prep Calories
## 1   Signature Espresso Drinks       2% Milk      510
## 2   Signature Espresso Drinks       Soymilk      460
## 3 Frappuccino® Blended Coffee    Whole Milk      460
```

➜ Make sure to view your data at each step

3. Wrap up!

Important functions of the dplyr grammar

Function	Meaning
mutate()	Modify or create a variable
select()	Keep a subset of variables
filter()	Keep a subset of observations
arrange()	Sort the data
group_by()	Group the data
summarise()	Summarizes variables into 1 observation per group
bind_rows()	Append data
left/right/inner/full_join()	Merge data
pivot_longer/wider()	Reshape data