Introduction to R

Quantitative Methods I

Malo Jan

2024-09-19

Table of contents

Course overview 5

Course structure . . . 5

Course validation . . . 5

Requirements . . . 6

Course material . . . 6

Help . . . 6

Ressources . . . 7

Setting up R and Rstudio 8

Install R . . . 8

Install Rstudio . . . 8

Set preferences and check if everything works . . . 9

Installation issues . . . 10

Mac . . . 10

Windows . . . 10

I Getting started with R and Rstudio 11

1 Rstudio, scripts, quarto 12

1.1 How Rstudio is organized . . . 12

1.2 Scripts . . . 13

1.3 Quarto . . . 13

2 R basics 15

2.1 R as calculator . . . 15

2.2 Objects . . . 16

2.3 Vectors . . . 17

2.3.1 Characters vectors . . . 18

2.4 Functions . . . 19

2.5 Missing values . . . 20

2.6 Packages and libraries . . . 21

2.7 Dataframes and tibbles . . . 22

2.8 Import and write data . . . 24

2.8.1 File formats, paths and R projects . . . 24

2.8.2 Functions to import data in R 26
2.8.3 Export data 27
2.9 Going further 27
2.9.1 Naming things 27
2.9.2 Packages 27
2.9.3 Indexing 28
2.9.4 Logical vectors 29
2.9.5 More on vectors and functions 30

3 Getting help and dealing with errors 34
3.1 R documentation 34
3.2 Cheatsheets 34
3.3 Online ressources 34
3.4 AI is your friend 35
3.5 Most common errors 35

II Manipulating and describing data 39

4 Manipulating data 40
4.1 Introduction to dplyr functions 40
4.1.1 Count the observations of groups with `count()` 41
4.1.2 `filter()` observations 43
4.1.3 Your turn 46
4.2 `select()` variables 46
4.3 Change column names with `rename()` 47
4.4 Calculating statistics by group with `group_by()` and `summarise()` 47
4.5 Your turn 50
4.5.1 Mutate 50
4.6 Recode values with `case_when` 51
4.6.1 Your turn ! 52

5 Reshaping data 53
5.1 Reshaping with `pivot_longer()` and `pivot_wider()` 53
5.2 Combining with bind_rows 55

III Visualizing data 58

IV Testing relationships 59

V Regression analysis 60

VI Multivariate analysis **61**

Course overview

This website contains all the material for the lab sessions of the Quantitative Methods class for the Fall 2025 semester in the research master's program in Political Science at Sciences Po Paris. The class complements Jan Rovny's lecture on Quantitative Methods I.

The course provides you with the fundamentals, resources, and motivation to further your learning independently, prepare for the next semester Luis Sattelmayer's R sessions, and apply quantitative methods in your future research.

Course structure

Session	Description	Date
Session 1	Getting started with R and Rstudio	05/09, 12/09
Session 2	Manipulating and describing data	19/09, 26/09
Session 3	Visualizing data	03/10, 10/10
Session 4	Testing relationships	17/10, 24/10
Session 5	Correlation and simple linear regression	07/11, 14/11
Session 6	Multivariate analysis	21/11, 28/11

Course validation

Learning programming is fundamentally about practice. It involves trying, encountering challenges, and solving them. The course assessment is structured around a series of exercises that will constitute 30% of your final grade in the Quantitative Methods class. You will be required to apply the code and concepts covered in class to new problems and datasets.

Throughout the semester, you will complete four individual exercises between sessions. For each exercise, you will perform a series of operations in R and submit your work. Each exercise will be graded on a scale from 0 to 5 points.

Second, at the end of the semester, you will complete a bigger group exercise with one other person. This exercise will involve integrating content from throughout the semester and will be completed within a two-week timeframe. It will be graded on a scale from 0 to 20 points.

Assignments	Description	Due date	Weight
Exercise 1	Individual	7 days after Session 1	5%
Exercise 2	Individual	7 days after Session 2	5%
Exercise 3	Individual	7 days after Session 3	5%
Exercise 4	Individual	7 days after Session 4	5%
Exercise 5	Group	14 days after session 6	10%

Requirements

This class does not require any prior programming or statistical experience and is designed for complete beginners. However, it does require some basic knowledge of how to use a computer. You should be able to navigate your computer's file system, create and move folders, and download and save files. Specifically, each class's content will be provided as a zip file, which you will need to download, unzip, and move to a folder on your computer. If you are not familiar with these operations, please refer to this tutorial.

This class requires you to bring a laptop to each session with R and RStudio installed by the start of the course. The 'Setting up R and RStudio' page on this website provides a step-by-step guide for installation. If you encounter any issues during the setup or are unable to bring a laptop to class, please let me know.

Course material

Before each session, you will be provided with the class material, which you should download to your laptop to follow along. Additionally, all class activities and resources will be available on this website for future reference. You will also need to submit your exercises on the course's Moodle page.

Help

If you have any questions regarding the course, need help, or are looking for additional resources, please do not hesitate to contact me via email. I will be happy to assist you and will try to reply as quickly as possible.

Ressources

This course does not have a required textbook or mandatory readings. However, if you wish to deepen your understanding of the content covered, I recommend familiarizing yourself with the following resources:

- R for Data Science, this is THE R classic by Hadley Wickham, you should definitely take a look to better understand what we cover in this course.
- Telling stories with data by Rohan Alexander : one of my favorite book on data science with R. (a bit more advanced)
- Computational analysis of communication by van Atteveldt et al.
- Computational Thinking for Social Scientists by Jae Yeon Kim
- Introduction to data science by Rafael Irizarry

There are also many other introductions to R available online, each offering different approaches to teaching the same concepts. I recommend the following:

- Introduction to R, by Felix Lennert

- Introduction to R by Alex Douglas et al.

If you want ressources in french, these are the two most comprehensive introduction you will find :

- Introduction à R et au tidyverse by Julien Barnier

- Guide pour l'analyse de données d'enquêtes avec R by Joseph Larmarange

Setting up R and Rstudio

The goal of this course is to apply what you learn in the introduction to quantitative methods' course by learning **programming**. This involves giving instructions to your computer to perform operations on quantitative data using code. The course focuses on coding with the programming language R, using the software RStudio.

To get the first session off to a good start and avoid wasting too much time setting up, **you must bring a laptop with R and Rstudio installed**. Instructions are provided below. This should be fairly quick. If you have any issues to install these programs correctly, **please contact me before the first class**.

R and RStudio are two different things. R is a programming language and software widely used for statistical analysis. While it is possible to use the default software directly, many (if not most) prefer to use RStudio instead. RStudio is an integrated development environment (IDE) that provides a much more functional and user-friendly interface to interact with R, which is why we choose to use it. However, we still need to install R before installing RStudio, as the latter relies on the former to function properly.

Install R

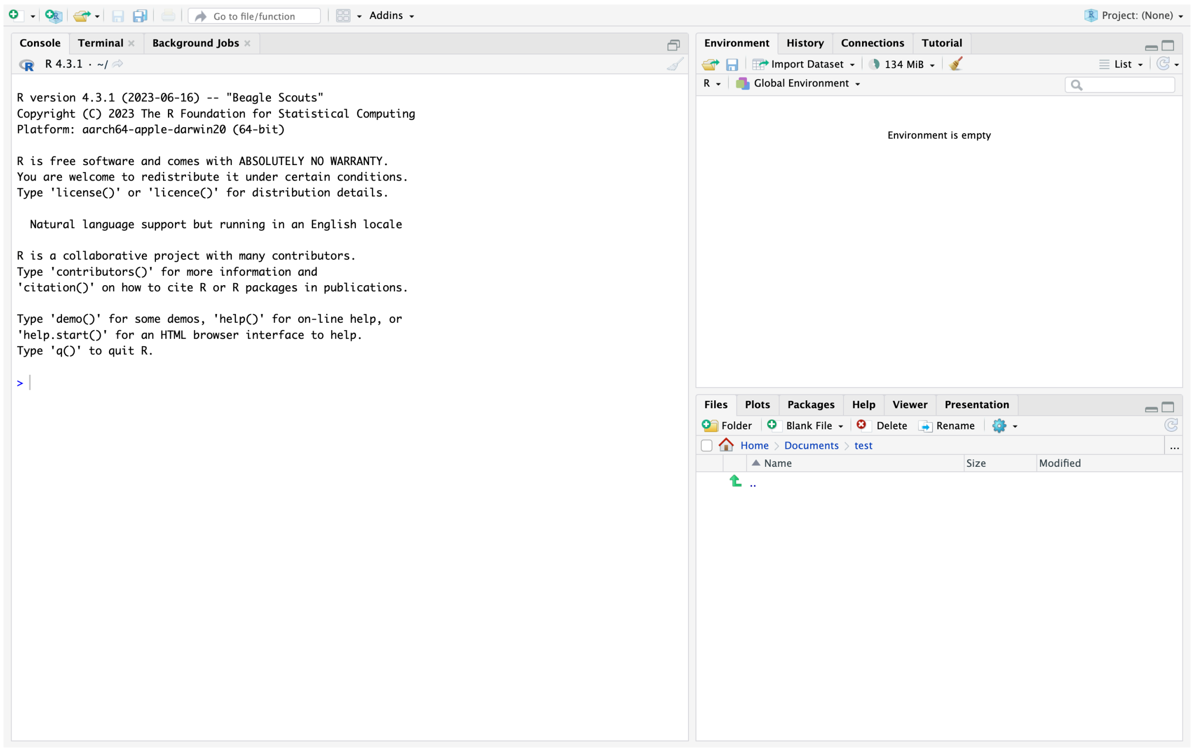
You can install R from CRAN (The comprehensive R Archive Network). On CRAN, you will find links to download the version of R you need for your operating system (Windows, Mac or Linux). Once the download is complete, you need to execute the installer.

Install Rstudio

Once R is set up, you can install Rstudio. For this, go on this webpage, download Rstudio and follow the instructions.

Set preferences and check if everything works

Once the installation of both R and Rstudio is complete, you should open Rstudio. If everything worked, something very similar to the screenshot below should appear. If it doesn't open, restart your latptop and try again.



```
R version 4.3.1 (2023-06-16) -- "Beagle Scouts"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

RStudio has some default settings that are worth changing to adopt best practices. Please do the following:

- Tools /Global options/RGeneral : Save workspace to never and uncheck the box `Restore .Rdata into workspace at startup`.

While R provides a series of basic commands for data manipulation, many of the functionalities we will use come from **packages** that need to be installed and loaded via RStudio. The most well-known package we will use is called `tidyverse`. To check if your installation is successful and you are ready to start the course, copy-paste the following code into the console pane and press enter. It might take a few minutes, and at the end, you should see the same message as the one below.

```
if (!require(tidyverse)) {install.packages("tidyverse")}
```

```
Loading required package: tidyverse
```

```
-- Attaching core tidyverse packages ------------------- tidyverse 2.0.0.9000 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becor
```

Installation issues

If you do not see the message above, welcome in the world of troubleshooting! Depending on your operating system (MacOS, Windows, Linux) and the configuration of your computer, the installation process may not go as smoothly as expected. Here are some common issues and how to solve them:

Mac

If a message appears saying that the package cannot be found, try the following :

```r
if (!require(tidyverse)) {devtools::install_github("hadley/tidyverse")}
```

Windows

If a message appears saying that the package cannot be found, try to install Rtools by copying and pasting the following code into the console pane.

```r
if (Sys.info()["sysname"] == "Windows") {
  install.packages("Rtools")
}
```

If a message appears that Rtools is not available, download it from here. Then restart your computer and try to install the tidyverse package again.

If all of this does not work or you have different types of issues, please contact me.

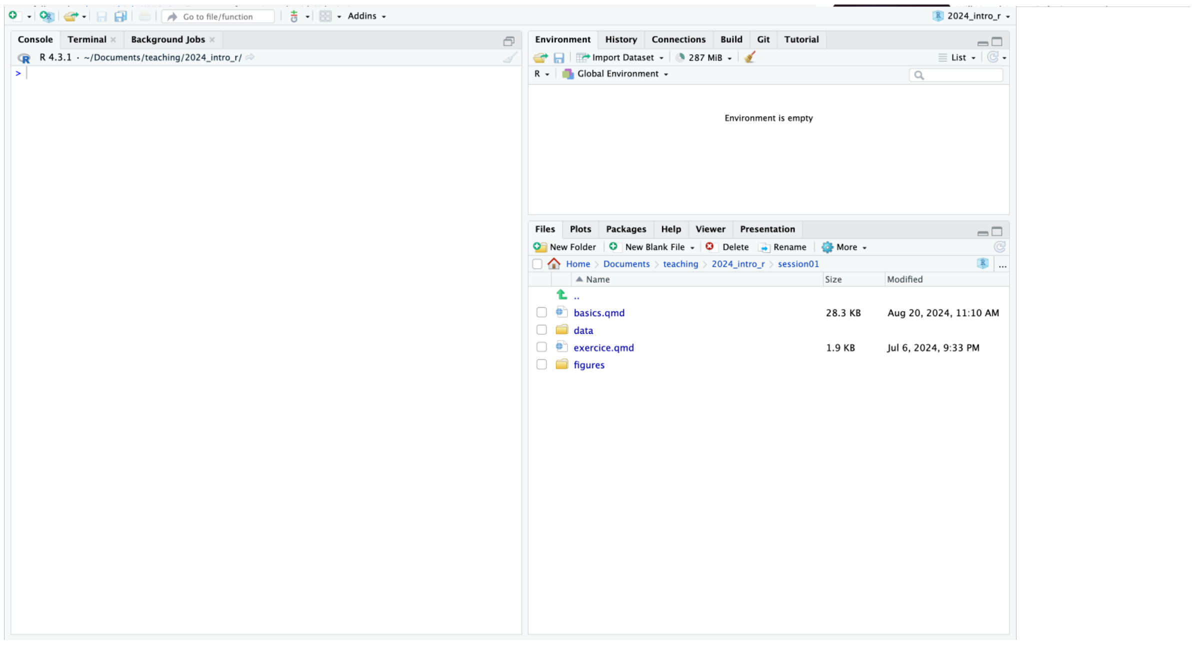
Part I

Getting started with R and Rstudio

1 Rstudio, scripts, quarto

1.1 How Rstudio is organized

To interact with the language R, we use the interface of Rstudio. When you open it on your laptop, you will probably see something similar to the screenshot below.



On the upper right panel, you have what we call the *environment.* At this stage, you should see a message indicating that your environment is empty. However, when we use data in RStudio, what we manipulate will appear there. For instance, if I import a dataset, I will see in that pane that my dataset has been imported.

On the bottom left panel, you see what we call our working directory. This is where the files, such as the data, that we want to use are stored on our laptop. It is essential to be aware that if you remove a file here from RStudio, it will also be deleted from your laptop. Exercise caution when managing files in this directory to avoid unintentional deletions.

On the left side, you have the console where you can type commands, indicated by the `>` sign. If you type something here and then click on `Enter`, you will see the result. So, if you type

2+2, it will produce the output 4 just below. Whatever code you produce, the output will appear in the console.

1.2 Scripts

However, when we interact with R, we rarely use the console to type code. Instead, we mostly use *scripts*, which are specific types of documents where you can write code, save it, and reuse it later on. To open a new script, you can go into the menu bar, click on `File > New File > Rscript` (you can also use the shortcut `Ctrl+Shift+N` on Windows/Linux or `Cmd+Shift+N` on macOS). his will open a new document, known as an R script, in the editor on the upper left side of the interface. Here, you can start writing and saving your R code for analysis and projects. You can type again `2+2`. To run a line on a Rscript, place the cursor on the line of code you want to run and press `Ctrl + Enter` (on Windows/Linux) or `Cmd + Enter` (on macOS). Just above the script in RStudio, you will also find a `Run` button that does the same thing. Once you run that code, you should see the output printed in the console. Note that in R scripts, you can write text and comment your code with the `#`. To save your script, simply click on the small disk icon in the editor's toolbar and choose a name for the file (`Ctrl + S` on Windows/Linux or `Cmd + S` on macOS). You will now see it in the files pane at the bottom left of your screen.

```r
2+2 # This is my first code
```

```
[1] 4
```

1.3 Quarto

While many users opt for R scripts for coding, I won't extensively employ them in this class. Instead, I'll be using Quarto documents. You will also be using Quarto as I'll be requesting you to submit your assignments using this format. Quarto constitutes a method of cohesively editing code and text within a single document, a practice termed literate programming. It enables you to compose code, generate output, conduct analyses in the same document, format the content, and subsequently export it to diverse formats like Word documents, PDFs and html. Additionally, Quarto facilitates the creation of slides, dashboards, books, and websites. Personally, I employ Quarto to craft the course materials.

To create a quarto document, click on `File > New File > Quarto Document` in the menu bar. You will see different options appear but for now, uncheck `use virtual markdown editor` and click on create an empty document. There are three main differences with an R script :

- **YAML** : when you open a quarto document you will have to fill the top matter, called a call a `YAML` which is separated by `---` where you can write different informations and options such as the title, the author, subtitle, abstract, the data and many other things. You can also choose the format of the output by writing `format:` and choosing among `html`, `docx` or `pdf`.
- You can type text without the `#` as if you where typing in any text editor
- To write code, you need to create what we call a chunk. For this, either go on the menu bar : `Code > Insert Chunk` or `option + Cmd + i` in macOs, `Ctrl+Alt+i` on Windows/Linux. You can then directly type code in that chunk.

To generate a pdf/html/word document, you need to click on Render (`Cmd + Shift + K` in macOs, {< kbd win=Shift-Ctrl-P >}}). A new file will appear in your files pane with the output.

2 R basics

2.1 R as calculator

First of all, R is a fancy calculator that can be used to perform fundamental arithmeric operations.

```r
3+7+10 # Addition
```

```
[1] 20
```

```r
4-5 # Substraction
```

```
[1] -1
```

```r
3*9*10 # Multiplication
```

```
[1] 270
```

```r
2/6 # Division
```

```
[1] 0.3333333
```

```r
2^2 # Exponentiation
```

```
[1] 4
```

```r
(2+2)-(4*4)/2^2 # Mix of operations
```

```
[1] 0
```

2.2 Objects

When you run code in R, the results are shown in the console. However, you cannot directly reuse these results in further operations, which is what we want to do. To address this, we use **objects**. Objects in R act as containers that store values, allowing you to keep information for later use. To create an object in R, you use the **assignment operator `<-`** .

```r
my_object <- 2
my_object
```

```
[1] 2
```

Let's consider an example from the most recent French general election. The results led to a highly fragmented parliament, with 11 different parliamentary groups and no single party or coalition able to form an absolute majority on its own. This situation raises the question of which coalitions could be formed to achieve a majority and pass legislation. The code below uses objects to store the number of seats obtained by each parliamentary group.

```r
# Left-wing groups
communists_seats <- 17
lfi_seats <- 72
greens_seats <- 38
socialists_seats <- 66

# Macron's party and allies
renaissance_seats <- 99  # This is Macron's party
modem_seats <- 36        # MODEM, a centrist party
horizon_seats <- 31      # Horizon, party of the former PM Édouard Philippe

# Right-wing groups
conservatives_seats <- 47  # "Les Républicains"

# Far-right groups
rn_seats <- 126            # National Rally (RN)
ciotti_seats <- 16         # Former conservatives who allied with Le Pen's party

# Others
liot_seats <- 22           # A mix of some independent centrists and regionalists
none_seats <- 7            # Non-affiliated MPs
```

After executing these lines, you should be able to see the objects created and their values in the environment pane located in the upper-right section of RStudio. Once these objects created, it is possible to perform operations on them. For instance, it is possible to compute the total number of seats of actual existing coalitions by summing the number of seats of the different parties that compose them and saving them in new objects. Below, I create a new object `left_seats` that stores the total number of seats obtained by the left-wing parties by summing the different objects that store the number of seats obtained by the left-wing parties.

```
left_seats <- communists_seats + lfi_seats + greens_seats + socialists_seats
left_seats
```

```
[1] 193
```

Your turn !

Create two new objects `macron_seats` and `far_right_seats` that store the total number of seats obtained by Macron's allies and far-right parties respectively.
Solution. Click to expand!
Solution:

```
macron_seats <- renaissance_seats + modem_seats + horizon_seats
far_right_seats <- rn_seats + ciotti_seats
```

2.3 Vectors

The objects we used so far contained only **one** numeric value. However, what we mostly manipulate in R are **vectors**, which are sequences of different values on which we can perform operations. Vectors can be of different types (eg : numeric, character, logical, date) but they have to be of the same type. For instance, a numeric vector is a sequence of different numbers and a character vector is a sequence of different strings. Vectors are also unidimensionals which mean they contains only one sequence of values and not several such as matrices do.

We can generate vectors with `c()` which stands for "concatenate". For instance, here, I create a vector containing the values of the seats obtained by different coalitions in the election. As a result, the vector `coalition_seats` contains the number of seats obtained by the left, Macron's party and allies, the far-right and the conservatives.

```
coalition_seats <- c(left_seats, macron_seats, far_right_seats, conservatives_seats)
coalition_seats
```

```
[1] 193 166 142  47
```

We use vectors to store different values because it is possible to perform the same operation on all the values of a vector at once. Let's say we want to know the number of seats that an existing coalition would need to have an absolute majority in the parliament. We can do this by creating a new object, `majority`, that stores the number of seats needed for an absolute majority, and then subtracting the number of seats obtained by the different coalitions from this value. The result will be a vector showing the number of seats each coalition needs to reach an absolute majority, which we could also store in a new object if desired.

```
majority <- 577/2 + 0.5 # Half of the number of seats (577) + 0.5 to round up (there is no ha

majority
```

```
[1] 289
```

```
majority - coalition_seats
```

```
[1]  96 123 147 242
```

2.3.1 Characters vectors

So far, we have only used numerical vectors, which consist of numbers. However, we can also create character vectors, which are made up of strings enclosed in quotes (either single ' or double "). For example, we can create a vector containing the names of different parliamentary leaders.

```
left_leaders <- c("Chassaigne", "Chatelain", "Panot", "Vallaud") # Create a vector of left-wi
far_right_leaders <- c("Ciotti", "Le Pen") # Create a vector of far-right leaders
```

As for other vectors, you can combine them in a single vector which will return a vector with all the leaders' names.

```
leaders <- c(left_leaders, far_right_leaders)
leaders
```

```
[1] "Chassaigne" "Chatelain"  "Panot"      "Vallaud"    "Ciotti"
[6] "Le Pen"
```

ℹ Your turn !

Create a vector that contains the names of the leaders of the Macron's party : Attal, Fesneau, Marcangeli. Then, add this vector to the `leaders` vector and store the result in a new object `all_leaders`.
Solution. Click to expand!
Solution:

```
macron_leaders <- c("Attal", "Fesneau", "Marcangeli")
all_leaders <- c(leaders, macron_leaders)
```

2.4 Functions

To manipulate vectors and conduct operations on them, we use **functions**. A function is a reusable block of code that performs a specific task, it takes several input values called *arguments* and produce an *output*. Let's say you want to know how many seats parliamentary groups have on average in the French parliament. You could calculate the sum of the seats and dividing them by their number. But you could also just the `mean()` function that exists in R and that takes a vector of numbers as argument.

```
parl_seats <- c(communists_seats, greens_seats, socialists_seats, lfi_seats, renaissance_sea

mean(parl_seats) # Compute the mean of the vector parl_seats
```

```
[1] 48.08333
```

It is also easy to use functions to compute the maximum, minimum, and sum of a vector of numbers.

```
max(parl_seats) # Compute the maximum of the vector parl_seats
```

```
[1] 126
```

```
min(parl_seats) # Compute the minimum of the vector parl_seats
```

```
[1] 7
```

```r
sum(parl_seats) # Compute the sum of the vector parl_seats
```

```
[1] 577
```

2.5 Missing values

In R, a missing value is represented by the symbol `NA`, which stands for "Not Available." Missing values can arise for a variety of reasons, such as data not being observed or recorded, errors in data collection, or intentional omissions. Understanding and handling missing values is crucial because they can influence the results of your analysis or even cause some functions to return errors. For instance, imagine I haven't found any information about the number of seats one parliamentary group get, but I want to retain this information in my vector. So, I add an NA to it.

```r
parl_seats <- c(parl_seats, NA)
parl_seats
```

```
 [1]  17  38  66  72  99  36  31  47  16 126   7  22  NA
```

When analyzing data, it's not uncommon to encounter NA values, and it's important to be aware of them. To check if a vector contains NA values, you can use the `is.na()` function. This function will returns what is called a *logical vector* that is a sequence of values where each element is either (`TRUE`) or not (`FALSE`). Logical vectors are often used to represent conditions or logical statements.

```r
is.na(parl_seats) # Check which values of a vector are NAs
```

```
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[13]  TRUE
```

This is important because certain functions will not operate properly if there are NA values in your data. For instance, the `mean()` function will return NA if the data contains any NA values

```r
mean(parl_seats)
```

```
[1] NA
```

To deal with NA, the `mean()` function has a `na.rm`

```r
mean(parl_seats, na.rm = TRUE) # Remove NA before computing the mean
```

```
[1] 48.08333
```

2.6 Packages and libraries

The functions we've discussed so far, such as `sum()` and `mean()`, come from base R. These are pre-loaded functions available immediately upon starting R. However, many functions you'll encounter aren't part of base R but instead belong to specific packages that individuals or groups have developed. You can think of packages as collections of functions crafted to simplify certain tasks or to introduce new capabilities to R. For example, there's the `tidyverse` package, which I asked you to install before the class

To install a package in R, you can use the `install.packages()` function, passing the name of the package in quotation marks (either single or double). I recommend doing this installation in the console since you don't need to save this step; it's a one-time action. However, every time you start your script or Quarto document, you'll need to load the package. To do this, use the `library()` function, providing the package name as an argument, but without the quotation marks.

Images sourced from https://www.wikihow.com/Change-a-Light-Bulb

```r
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------- tidyverse 2.0.0.9000 --
v dplyr     1.1.4     v readr     2.1.5
```

```
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
```

2.7 Dataframes and tibbles

When processing data, we primarily interact with vectors through the manipulation of **dataframes** in R. Dataframes are two-dimensional structures that contain rows and columns. Each column is a vector, and each row is an observation. Dataframes are the primary data structure used for data manipulation, computation, and visualization in R. In this class, we'll work with a specific type of dataframe that comes from the `tidyverse` package called a **tibble**. Tibbles are a modern and enhanced version of dataframes that make them easier to print and manipulate.

To understand what dataframes look like, let us continue with the results of the french elections. I manually create a tibble with the `tibble()` function with different variables about parliamentary groups

```r
left_coalition <- tibble(
  party = c("Communists", "Greens", "LFI", "Socialists"),
  leader = c("Chassaigne", "Chatelain", "Panot", "Vallaud"),
  seats_2024 = c(17, 38, 72, 66),
  seats_2022 = c(22, 21, 75, 31)
)

left_coalition
```

```
# A tibble: 4 x 4
  party      leader     seats_2024 seats_2022
  <chr>      <chr>           <dbl>      <dbl>
1 Communists Chassaigne         17         22
2 Greens     Chatelain          38         21
3 LFI        Panot              72         75
4 Socialists Vallaud            66         31
```

You see now that we have a new object in our Environment Pane with 4 observations and 4 variables.

Different functions are available to get an idea of the informations and shape of the dataframe, which are useful when we load an unknown dataset and we want to understand its structure, what are the observations and variables.

```r
glimpse(left_coalition) # Get a glimpse of your data
```

```
Rows: 4
Columns: 4
$ party      <chr> "Communists", "Greens", "LFI", "Socialists"
$ leader     <chr> "Chassaigne", "Chatelain", "Panot", "Vallaud"
$ seats_2024 <dbl> 17, 38, 72, 66
$ seats_2022 <dbl> 22, 21, 75, 31
```

```r
colnames(left_coalition) # Retrieve column names of the dataframe
```

```
[1] "party"      "leader"     "seats_2024" "seats_2022"
```

```r
summary(left_coalition) # Return a summary of the variables
```

```
    party              leader            seats_2024      seats_2022
 Length:4           Length:4           Min.   :17.00   Min.   :21.00
 Class :character   Class :character   1st Qu.:32.75   1st Qu.:21.75
 Mode  :character   Mode  :character   Median :52.00   Median :26.50
                                       Mean   :48.25   Mean   :37.25
                                       3rd Qu.:67.50   3rd Qu.:42.00
                                       Max.   :72.00   Max.   :75.00
```

If we want to access only one variable (one vector) of that dataframe, we use the `$` sign. This will return a vector of the values of this variable.

```r
left_coalition$party # Select the party variable
```

```
[1] "Communists" "Greens"     "LFI"        "Socialists"
```

You can also create new variables based on the existing ones. Here I create a new variable called `seats_change`by calculating the difference of seats between 2024 and 2022 for each party.

```
left_coalition$seats_change <- left_coalition$seats_2024 - left_coalition$seats_2022
```

Your turn !

Create a new variable called **seats_share** that calculates the share of seats in 2024 for each party. The share of seats is calculated as the number of seats of the party divided by the total number of seats in the parliament multiplied by 100.
Solution. Click to expand!
Solution:

```
left_coalition$seats_share <- left_coalition$seats_2024/577*100
left_coalition
```

```
# A tibble: 4 x 6
  party      leader     seats_2024 seats_2022 seats_change seats_share
  <chr>      <chr>           <dbl>      <dbl>        <dbl>       <dbl>
1 Communists Chassaigne         17         22           -5        2.95
2 Greens     Chatelain          38         21           17        6.59
3 LFI        Panot              72         75           -3       12.5
4 Socialists Vallaud            66         31           35       11.4
```

2.8 Import and write data

Up until now, we have been creating data manually for demonstration purposes. As we move forward, we will focus on analyzing real data. Every data analysis project starts with acquiring data. You can generate your own data through surveys, web scraping, or manual data coding (*primary data sources*). However, there are also many pre-existing datasets available for use (*secondary data sources*). These datasets are often provided by researchers, governments, NGOs, companies, international organizations, and more. I highly recommend checking out this list of political datasets curated by Erik Gahner Larsen. Throughout this course, we will use some of these established datasets in political science.

2.8.1 File formats, paths and R projects

To analyze data in R, we first need to import it. While this might sound simple, it can be challenging for beginners. To read a file in R, we need to know two important things: the **file format** and the **path**.

Data comes in various file formats, which are standardized ways of storing and organizing data in digital files. These formats dictate how information is encoded and structured, allowing different software programs to understand and interpret the data correctly. The most common format for data is `.csv` (comma-separated values). In political science, you will also encounter Stata (`.dta`) and SPSS (`.sav`) files. R uses different functions to read files depending on their format.

Secondly, R needs to know where the data is located on your system. A "path" shows the position of a file or folder within your file system, detailing the series of directories and subdirectories leading to the file. There are two types of paths:

- The **absolute path** gives the complete location of a file or directory, beginning at the root of the file system. Examples include: /home/user/documents/myfile.txt for Unix-like systems, and `C:\Users\user\Documents\myfile.txt` for Windows.
- A **relative path** indicates the location of a file or directory in relation to the current working directory. For instance, data/mydata.csv points to a file named mydata.csv in the data subdirectory of the present directory.

The **working directory** refers to the directory where R is currently operating. If you access files in R without providing an absolute path, it defaults to searching within this working directory.

You can see your current working directory in R using the `getwd()` function. To set a new working directory, use the `setwd()` function, specifying the desired path as its argument.

```r
getwd()
```

```
[1] "/Users/malo/Documents/teaching/2024_intro_r/session01"
```

Please note that using absolute paths in your code is considered a bad practice because it can make your code less usable for others. Instead, I recommend using R projects. An R project is a dedicated workspace for your R work, where you keep all your files, data, scripts, and output documents together. When you open an R project, it sets everything up so that your files are easy to find and your work is easier to share and reproduce.

From the top left corner of RStudio, click on `File` and then select `New Project`. You'll then be given three options: to create a project in an existing directory, to create one in a new directory, or to check out a project from a version control repository like Git.

If you choose an existing directory, navigate to that directory. If you opt for a new directory, you'll need to name your project and decide its save location on your computer. Once you finalize your choice by clicking `Create Project`, the working directory in RStudio will automatically be set to your project location. This means that any scripts, data files, or outputs you work on will be saved here by default, making them easier to find and reference later.

Inside your project directory, you'll notice a file with an `.Rproj` extension, such as `MyProject.Rproj`. In the future, you can open this file to launch RStudio directly into this project, ensuring the working directory is already set. It's also advisable to set up specific folders within your project directory for different components like scripts, data, and figures. This keeps everything tidy and organized as your project expands.

2.8.2 Functions to import data in R

There are different functions to import data into R.

- For CSV (Comma Separated Values) files, the base R function `read.csv()` is commonly used. However, within the tidyverse package suite, the readr package provides the `read_csv()` function, which tends to be faster and more versatile. The `read_csv2()` function is designed for CSV files using semicolons ; as field separators and commas , as decimal points, compared to `read_csv()`which assumes commas as field separators.
- Excel files can be read using the readxl package, which provides the `read_excel()` function. This function can read both `.xls` and `.xlsx` files.
- For SPSS data files, you can use the `haven` package. This package contains the function `read_sav()` for `.sav` files.
- Stata data files, or `.dta` files, can also be read using the haven package with the `read_dta()` function.

All these functions enable you to read a file by specifying its path. Here, for example, I import a CSV file containing data on candidates for the 2024 legislative elections in France.

```r
library(tidyverse)

french_candidates <- read_csv("data/2024_french_candidates.csv")
```

```
Rows: 4009 Columns: 9
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (7): code_department, departement, first_name, name, nuance, gender, pro...
dbl (2): exprimes_per, age

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
glimpse(french_candidates)
```

```
Rows: 4,009
Columns: 9
$ code_department <chr> "01", "01", "01", "01", "01", "01", "01", "01", "01", ~
$ departement     <chr> "Ain", "Ain", "Ain", "Ain", "Ain", "Ain", "Ain", "Ain"~
$ first_name      <chr> "Christophe", "Xavier", "Sébastien", "Vincent", "Éric"~
$ name            <chr> "MAÎTRE", "BRETON", "GUERAUD", "GUILLERMIN", "LAHY", "~
$ nuance          <chr> "RN", "LR", "UG", "ENS", "EXG", "DSV", "DSV", "RN", "D~
$ exprimes_per    <dbl> 39.37, 23.96, 23.45, 11.68, 0.69, 0.52, 0.33, 39.20, 2~
$ gender          <chr> "M.", "M.", "M.", "M.", "M.", "M.", "M.", "M.", "M.", ~
$ age             <dbl> 55, 62, 51, 48, 58, 36, 42, 35, 44, 37, 49, 69, 59, 43~
$ profession      <chr> "(22) - Commerçant et assimilé", "(33) - Cadre de la f~
```

2.8.3 Export data

Typically, we might want to save data to our disk after adding information, merging different datasets, and so on. This is useful for later reuse or to share the data with someone else. To achieve this, simply replace 'read' with 'write' in all the functions I've introduced previously. You'll also need to specify the name of the R object containing your data and the path where you wish to export the data.

```r
write_csv(french_candidates, "data/french_candidates2.csv")
```

2.9 Going further

2.9.1 Naming things

Note that I have written the names of objects with underscores. There are different conventions to write object names in R that you can discover here. I personnaly use *snake case* which use lowercase letters and underscores to separate words.

2.9.2 Packages

Rather than importing packages with `library()`, it is also possible to use the `::` operator such as :

```r
dplyr::glimpse(french_candidates)
```

```
Rows: 4,009
Columns: 9
$ code_department <chr> "01", "01", "01", "01", "01", "01", "01", "01", "01", ~
$ departement     <chr> "Ain", "Ain", "Ain", "Ain", "Ain", "Ain", "Ain", "Ain"~
$ first_name      <chr> "Christophe", "Xavier", "Sébastien", "Vincent", "Éric"~
$ name            <chr> "MAÎTRE", "BRETON", "GUERAUD", "GUILLERMIN", "LAHY", "~
$ nuance          <chr> "RN", "LR", "UG", "ENS", "EXG", "DSV", "DSV", "RN", "D~
$ exprimes_per    <dbl> 39.37, 23.96, 23.45, 11.68, 0.69, 0.52, 0.33, 39.20, 2~
$ gender          <chr> "M.", "M.", "M.", "M.", "M.", "M.", "M.", "M.", "M.", ~
$ age             <dbl> 55, 62, 51, 48, 58, 36, 42, 35, 44, 37, 49, 69, 59, 43~
$ profession      <chr> "(22) - Commerçant et assimilé", "(33) - Cadre de la f~
```

It lets you reference a specific function from a package without loading the whole package. This is handy when two packages have functions with the same name, ensuring clarity in your code. It's also useful for one-off function uses, avoiding the need to load an entire package. This approach can make code clearer and sometimes faster by reducing loaded dependencies

2.9.3 Indexing

When we manipulate vectors, we often want to access specific elements of them, which we call **indexing**, which is performed by using square brackets `[]`. You can index either by position or by name.

When I write `leaders[3]`, I want the value of the third element of the `leaders vector`, this is indexing by position. But when I write `leaders[leaders == "Le Pen"]`, I index by name because I want the elements that have Le Pen as value.

```r
leaders
```

```
[1] "Chassaigne" "Chatelain"  "Panot"      "Vallaud"    "Ciotti"    
[6] "Le Pen"    
```

```r
leaders[4] # Get the third element of the vector
```

```
[1] "Vallaud"
```

```r
leaders[-3] # Get everything but the third element of the vector
```

```
[1] "Chassaigne" "Chatelain"  "Vallaud"    "Ciotti"     "Le Pen"    
```

```r
leaders[c(1,4)] # Get the first and the fifth elements of the vector
```

```
[1] "Chassaigne" "Vallaud"
```

```r
leaders[1:3] # Get elements from the first to the third
```

```
[1] "Chassaigne" "Chatelain"  "Panot"
```

```r
leaders[leaders == "Le Pen"] # Which has Le Pen as value
```

```
[1] "Le Pen"
```

```r
leaders[leaders != "Le Pen"] # Which has not Le Pen as value
```

```
[1] "Chassaigne" "Chatelain"  "Panot"      "Vallaud"    "Ciotti"
```

```r
leaders[leaders %in% c("Le Pen", "Ciotti")]# Which has Le Pen or Ciotti
```

```
[1] "Ciotti" "Le Pen"
```

```r
leaders[!leaders %in% c("Le Pen", "Ciotti")]# Which has neither Le Pen nor Ciotti
```

```
[1] "Chassaigne" "Chatelain"  "Panot"      "Vallaud"
```

2.9.4 Logical vectors

Another type of vector in R is the logical vector, which consists of Boolean values: `TRUE` or `FALSE`. Logical vectors are useful for evaluating conditions. They can be used to check for errors in data or to filter variables based on specific criteria.

```r
c(TRUE, TRUE, TRUE, FALSE, FALSE, FALSE)
```

```
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

For instance, we could check whether there are no mistakes and confirm that Le Pen is not a leader of the left coalition. This means verifying that no value in the left_leaders vector is equal to “Le Pen”.

```r
left_leaders == "Le Pen"
```

```
[1] FALSE FALSE FALSE FALSE
```

```r
far_right_leaders == "Le Pen"
```

```
[1] FALSE  TRUE
```

The evaluation of conditions can also be used to compare numeric values. For instance, we can check whether the left coalition has more seats than the far-right coalition.

```r
left_seats > far_right_seats
```

```
[1] TRUE
```

2.9.5 More on vectors and functions

In R, functions often expect inputs of specific types. If you pass a character vector containing numeric numbers as strings to a function that expects a numeric vector, it may not behave as expected. As shown below, the function returns a `NA` which means Not available/applicable. When R encounters something it doesn't understand, it returns an error message with indications about the problem.

```r
parl_seats_chr <- c("17", "38", "66", "72", "99", "36", "31", "47", "16", "126", "7", "22")
parl_seats_chr
```

```
 [1] "17"  "38"  "66"  "72"  "99"  "36"  "31"  "47"  "16"  "126" "7"   "22"
```

```r
mean(parl_seats_chr) # This returns NA
```

```
Warning in mean.default(parl_seats_chr): argument is not numeric or logical:
returning NA
```

```
[1] NA
```

Similarly, computing the sum of the `parl_seats_chr` vector will work as expected but trying to calculate the sum of our `leaders` character vector composed of leaders's names will not give a meaningful result.

```r
sum(parl_seats_chr)
```

```
Error in sum(parl_seats_chr): invalid 'type' (character) of argument
```

```r
sum(leaders) # This is an error
```

```
Error in sum(leaders): invalid 'type' (character) of argument
```

If you are not sure about the type of your vectors, you can check with the `class()` function that will give you the answer.

```r
class(parl_seats)
```

```
[1] "numeric"
```

```r
class(parl_seats_chr)
```

```
[1] "character"
```

Sometimes, a vector has not the good type for the operation we want to perform. To check the type of a vector, you can use the family of `is.` functions such as `is.numeric()` and `is.character()` that return a boolean operator. In case the vector is not the right type for our purpose, wan can try to coerce them with the family of `as.` functions such as `as.numeric()` and `as.character()`.

```r
is.numeric(parl_seats_chr) # Check if numeric
```

```
[1] FALSE
```

```r
parl_seats_num <- as.numeric(parl_seats_chr) # Coerce to numeric
is.numeric(parl_seats_num) # Check again if numeric
```

```
[1] TRUE
```

```r
mean(parl_seats_num) # Compute the mean
```

```
[1] 48.08333
```

Functions that you will find in R have been created by someone. You can also create your own functions in R. You usually start doing it when you are more advanced so do not worry it you find it hard, it is just for you to know that it is possible. Here I just create a simplified other function to calculate a mean in R.

```
# Create a function to compute the mean of a vector

compute_mean <- function(x) {
  # Compute the sum of the values in the vector and divide by the number of values (length)
  mean <- sum(x)/length(x)

  # Return the result
  return(mean)
}

compute_mean(parl_seats)
```

```
[1] NA
```

2.9.5.1 More on the tidyverse

The `tidyverse` isn't just a single package but rather a meta-package, meaning it bundles together several other packages, each with its own set of functions. For example, one of these bundled packages is `stringr`, which offers tools for manipulating character vectors. Since `stringr` is part of the `tidyverse`, if you've already loaded the `tidyverse`, there's no need to load `stringr` separately. With it, you can perform tasks like converting strings in a vector to uppercase or lowercase.

```
leaders
```

```
[1] "Chassaigne" "Chatelain"  "Panot"      "Vallaud"    "Ciotti"    
[6] "Le Pen"    
```

```
str_to_lower(leaders) # Change strings to lower class
```

```
[1] "chassaigne" "chatelain"  "panot"      "vallaud"    "ciotti"    
[6] "le pen"    
```

```r
str_to_upper(leaders) # Change strings to upper class
```

```
[1] "CHASSAIGNE" "CHATELAIN"  "PANOT"      "VALLAUD"    "CIOTTI"    
[6] "LE PEN"    
```

```r
str_detect(leaders, "C") # Detect if strings that contains a "C"
```

```
[1]  TRUE  TRUE FALSE FALSE  TRUE FALSE
```

We can also combine characters vectors together with `str_c()`.

```r
parties <- c("Communists", "Greens", "LFI", "Socialists", "Ciotti's party", "National Rally"

str_c(leaders, " is the parliamentary leader of ", parties)
```

```
[1] "Chassaigne is the parliamentary leader of Communists"
[2] "Chatelain is the parliamentary leader of Greens"
[3] "Panot is the parliamentary leader of LFI"
[4] "Vallaud is the parliamentary leader of Socialists"
[5] "Ciotti is the parliamentary leader of Ciotti's party"
[6] "Le Pen is the parliamentary leader of National Rally"
```

3 Getting help and dealing with errors

When learning a new language, mistakes are inevitable, and programming can be particularly frustrating due to the myriad of errors one can encounter. Unlike practicing a foreign language with speakers who may be forgiving of mistakes, R is exceptionally unforgiving. If your query is not structured correctly, it won't be understood at all. Therefore, it's crucial to find ways to pinpoint and understand the root causes of these errors. Overcoming errors is one of the most effective methods for learning R. Here are a few places where you can find the help you need:

3.1 R documentation

Every function or package in R comes with documentation provided by its creators. You can access this documentation directly from RStudio by placing a ? before the function or package name in the console and executing the command. This will open the documentation in the help pane, where you'll find a description of the function, its various arguments, and some usage examples.

```r
?readr::read_csv # Access the documentation of the read_csv function
```

3.2 Cheatsheets

Every package in the tidyverse (and some others) has a cheatsheet that provides information about its various functions. You can find these cheatsheets at this link For example, check out the `readr` cheatsheet.

3.3 Online ressources

First, remember that you're not alone in encountering errors in R; many others have faced similar issues before you. Often, they have sought help online. If R gives you an error you don't understand, it's likely that someone else has encountered the same issue and discussed it online. Start by checking if others have asked similar questions. Websites dedicated to R programming frequently have solutions from experienced users. Additionally, Google can be a valuable resource. Whether you're trying to accomplish a task (e.g., "how to import a `.dta` file in R") or resolve an error, searching the error message or your query can provide valuable insights.

Often, you'll find yourself on a site called Stackoverflow, a community hub for users of various programming languages. You can often copy and paste the code you find there, but it's important to tweak and adapt it to fit your specific needs. Remember, in the world of programming, it's common for everyone to borrow and adapt code from others. Another helpful resource is the Rstudio Commmunity. Additionally, the #rstats hashtag on Twitter can provide insights and discussions from the R programming community.

3.4 AI is your friend

Lastly, we've entered the era of generative AI. When coding, Large Language Models, particularly ChatGPT, can swiftly emerge as invaluable allies. By supplying ChatGPT with your errors or seeking guidance for specific coding tasks, you can obtain outstanding results. I strongly suggest incorporating it into your coding journey. However, exercise caution: ChatGPT might sometimes suggest non-existent functions or present inaccurate information. It isn't a magic bullet, but you can quickly assess the accuracy of its suggestions by testing the code in R. If the code doesn't work correctly, the information may be incorrect. For instance, check out this example example where I asked ChatGPT to explain how to import a Stata file in R.

3.5 Most common errors

Finally, some errors are really common and you will probably face them often. I provide you here a (non exhaustive) list of those to help you troubleshooting[1].

3.5.0.1 Syntax errors

Many errors beginners encounter in R stem from syntax issues: a slight coding mistake can lead RStudio to misunderstand your intentions. Common errors include typos in function names or forgetting symbols like `)`, `,`, or `"`. For example, if you missed a closing `"` when trying to subset the `Le Pen` string from the `leaders` vector: `leaders[leaders == "Le Pen]`, you'd likely see a + in the console. This indicates that R is awaiting further input to process your command.

[1]I rely on many blogposts, here, here, here, here, here and here

3.5.0.2 The "not found" errors

- `Error: function 'x' not found` : mispelling or package not loaded

```
Library(tidyverse)
```

```
Error in Library(tidyverse): could not find function "Library"
```

```
means(c(15,16,19))
```

```
Error in means(c(15, 16, 19)): could not find function "means"
```

```
read_html("https://labour.org.uk/category/latest/press-release/") # Read html code from a wel
```

```
Error in read_html("https://labour.org.uk/category/latest/press-release/"): could not find fu
```

Mistakes related to capitalization or misspelling are common. For instance, attempting to compute the mean of a number vector but mistakenly adding an "s" to the `mean()` function will lead to an error. In another scenario, you might aim to read a webpage's HTML code for web scraping purposes. While the function might be correctly spelled, the error arises if the required rvest package isn't loaded beforehand. When encountering such errors, ensure you've spelled functions correctly and loaded the necessary package (e.g., using `library(rvest)`).

- `Error: object 'x' not found` : typo, forgot to run the line or saving object

```
leaders <- c("Chassaigne", "Vallaud", "Chatelain", "Panot")
leader[1]
```

```
Error in eval(expr, envir, enclos): object 'leader' not found
```

You might alo want to look only at leaders from right-wing parties in the object `right_wing_leaders` Here, the error happens because we did not save any object with this value yet.

```
right_wing_leaders
```

```
Error in eval(expr, envir, enclos): object 'right_wing_leaders' not found
```

```r
right_wing_leaders <- c("Waucquiez")
right_wing_leaders
```

```
[1] "Waucquiez"
```

```
Error in install.packages : object 'x' not found
```

```r
install.packages(rvest)
```

```
Error in eval(expr, envir, enclos): object 'rvest' not found
```

Most of the time, you just forget the `""` and you should write `install.packages("rvest")`. It might also be a typo in the package name (eg. you would have an error with `install.packag("Rvest")`.

- `Error: 'x' does not exist in current working directory`

```r
readr::read_csv("thisdata.csv")
```

```
Error: 'thisdata.csv' does not exist in current working directory ('/Users/malo/Documents/tea
```

This error typically arises when you've given an incorrect path, and R can't find your file. Use `getwd()` to check your current working directory and then adjust the file path as needed.

3.5.0.3 Inconsistent data types

We have seen already that R comes with different data types such as `logical` or `character`. Many functions takes as argument a vector of a specific type and will not work on other. Below an obvious example : if we try to compute the mean of a character vector, this will not work.

```r
leaders
```

```
[1] "Chassaigne" "Vallaud"    "Chatelain"  "Panot"
```

```r
class(leaders)
```

```
[1] "character"
```

```r
mean(leaders)
```

```
Warning in mean.default(leaders): argument is not numeric or logical: returning
NA
```

```
[1] NA
```

Part II

Manipulating and describing data

4 Manipulating data

At this point, you can set up an R project, import a data file into RStudio, and quickly explore the dataset's structure, which is typically the first step in any analysis. If the dataset is well-structured, the next crucial step is to manipulate the data in R to obtain the information we need. Often, we are not interested in the entire dataset; instead, we focus on selecting specific variables, analyzing certain groups, and observing how these variables vary across those groups

With R, there are several ways to perform these operations. Firstly, you can use R's base functions, without having to load a specific package. An alternative approach is based on the `tidyverse`, the package suite developed by Hadley Wickham with a special syntax. Here, we'll concentrate on the `tidyverse` functions, which I feel are easier and more intuitive to use.

4.1 Introduction to dplyr functions

To manipulate data, we will use a package from the tidyverse called `dplyr`, comprising a comprehensive suite of exceptionally useful functions. To familiarize ourselves with its usage, we will explore the Quality of Government Environmental Indicators dataset. To do this, we will need first to load the tidyverse and import the data with the `haven` package.

```
# Load the tidyverse

library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------- tidyverse 2.0.0.9000 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becor
```

```r
library(haven)

qog <- read_dta("data/qog_env.dta")

head(qog)
```

```
# A tibble: 6 x 414
  cname       ccode  year cname_qog   ccode_qog ccodealp ccodealp_year ccodecow
  <chr>       <dbl> <dbl> <chr>           <dbl> <chr>    <chr>            <dbl>
1 Afghanistan     4  1946 Afghanistan         4 AFG      AFG46              700
2 Afghanistan     4  1947 Afghanistan         4 AFG      AFG47              700
3 Afghanistan     4  1948 Afghanistan         4 AFG      AFG48              700
4 Afghanistan     4  1949 Afghanistan         4 AFG      AFG49              700
5 Afghanistan     4  1950 Afghanistan         4 AFG      AFG50              700
6 Afghanistan     4  1951 Afghanistan         4 AFG      AFG51              700
# i 406 more variables: ccodevdem <dbl>, cname_year <chr>, version <chr>,
#   act_act <dbl>, as_rifr <dbl>, as_ws <dbl>, bti_envc <dbl>, ccci_coop <dbl>,
#   ccci_em <dbl>, ccci_fin <dbl>, ccci_kyoto <dbl>, ccci_rep <dbl>,
#   ccci_unfccc <dbl>, cckp_rain <dbl>, cckp_temp <dbl>, ccl_exepp <dbl>,
#   ccl_leglp <dbl>, ccl_lpp <dbl>, ccl_mitlpp <dbl>, ccl_nexep <dbl>,
#   ccl_nlegl <dbl>, ccl_nlp <dbl>, ccl_nmitlp <dbl>, edgar_bc <dbl>,
#   edgar_ch4 <dbl>, edgar_co <dbl>, edgar_co2gdp <dbl>, edgar_co2pc <dbl>, ...
```

4.1.1 Count the observations of groups with `count()`

One basic operation in data manipulation is to count the number of observations of different values in a variable. The `dplyr::count()` function is used for this purpose.

In Exercise 1, we observed that the qog dataset has a country-year structure. For each country, there is a series of indicators, with one value per year. The dataset includes a `year` variable and a `cname` variable representing the country.

We might want to know how many country observations we have for each year. To do this, we can use `count()` with the `year` variable, that will give us the number of observations for each year. If we look at the results, we see that the dataset has observations from 1946 to 2020 with an increasing number of countries in the dataset over time.

```r
count(qog, year)
```

```
# A tibble: 75 x 2
   year     n
   <dbl> <int>
 1  1946    75
 2  1947    76
 3  1948    82
 4  1949    85
 5  1950    88
 6  1951    88
 7  1952    89
 8  1953    89
 9  1954    91
10  1955    94
# i 65 more rows
```

```
table(qog$year) # Alternative way to do it
```

```

1946 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961
  75   76   82   85   88   88   89   89   91   94   96   97   97   98  103  118
1962 1963 1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977
 119  125  127  131  135  138  141  143  144  146  149  149  151  154  160  160
1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993
 160  165  166  167  168  168  170  170  170  172  172  172  172  173  189  192
1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009
 192  193  193  193  193  193  193  193  194  194  194  194  195  195  195  195
2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020
 195  195  195  195  195  194  194  194  194  194  194
```

If we want to cross the `year` and `cname` variables, we can use the `count()` function with both variables. This will give us the number of observations for each year and country. In this case, we see that we have only one observation for each country-year combination.

```
count(qog, year, cname)
```

```
# A tibble: 11,722 x 3
    year cname                                      n
   <dbl> <chr>                                  <int>
 1  1946 Afghanistan                                1
 2  1946 Albania                                    1
 3  1946 Andorra                                    1
 4  1946 Antigua and Barbuda                        1
 5  1946 Argentina                                  1
 6  1946 Australia                           1
 7  1946 Belgium                             1
 8  1946 Bhutan                              1
 9  1946 Bolivia (Plurinational State of)    1
10  1946 Brazil                              1
# i 11,712 more rows
```

4.1.2 `filter()` observations

We may want to use only a subset of countries and get a smaller versions of the dataset. And the tidyverse has one for this, which is called `filter()` We want to select specific rows/observations of the dataset based on a specific conditions. To filter, we will use extensively boolean operators.

Operator	Description
`==`	equal
`!=`	not equal
`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal
`&`	and
`|`	or
`!`	not
`%in%`	in the set

For instance, I might be only interested in environmental indicators for France. In that case I could filter the whole dataset with only the observations that has France as country variable.

```r
filter(qog, cname == "France")
```

```
# A tibble: 75 x 414
   cname  ccode  year cname_qog      ccode_qog ccodealp ccodealp_year ccodecow
   <chr>  <dbl> <dbl> <chr>              <dbl> <chr>    <chr>            <dbl>
 1 France   250  1963 France (1963-)       250 FRA      FRA63              220
 2 France   250  1964 France (1963-)       250 FRA      FRA64              220
 3 France   250  1965 France (1963-)       250 FRA      FRA65              220
 4 France   250  1966 France (1963-)       250 FRA      FRA66              220
 5 France   250  1967 France (1963-)       250 FRA      FRA67              220
 6 France   250  1968 France (1963-)       250 FRA      FRA68              220
 7 France    250  1969 France (1963-)       250 FRA      FRA69              220
 8 France    250  1970 France (1963-)       250 FRA      FRA70              220
 9 France    250  1971 France (1963-)       250 FRA      FRA71              220
10 France    250  1972 France (1963-)       250 FRA      FRA72              220
# i 65 more rows
# i 406 more variables: ccodevdem <dbl>, cname_year <chr>, version <chr>,
#   act_act <dbl>, as_rifr <dbl>, as_ws <dbl>, bti_envc <dbl>, ccci_coop <dbl>,
#   ccci_em <dbl>, ccci_fin <dbl>, ccci_kyoto <dbl>, ccci_rep <dbl>,
#   ccci_unfccc <dbl>, cckp_rain <dbl>, cckp_temp <dbl>, ccl_exepp <dbl>,
#   ccl_leglp <dbl>, ccl_lpp <dbl>, ccl_mitlpp <dbl>, ccl_nexep <dbl>,
#   ccl_nlegl <dbl>, ccl_nlp <dbl>, ccl_nmitlp <dbl>, edgar_bc <dbl>, ...
```

```
# If I want to keep all the countries except France
filter(qog, cname != "France")
```

```
# A tibble: 11,647 x 414
   cname       ccode  year cname_qog   ccode_qog ccodealp ccodealp_year ccodecow
   <chr>       <dbl> <dbl> <chr>           <dbl> <chr>    <chr>            <dbl>
 1 Afghanistan     4  1946 Afghanistan         4 AFG      AFG46              700
 2 Afghanistan     4  1947 Afghanistan         4 AFG      AFG47              700
 3 Afghanistan     4  1948 Afghanistan         4 AFG      AFG48              700
 4 Afghanistan     4  1949 Afghanistan         4 AFG      AFG49              700
 5 Afghanistan     4  1950 Afghanistan         4 AFG      AFG50              700
 6 Afghanistan     4  1951 Afghanistan         4 AFG      AFG51              700
 7 Afghanistan     4  1952 Afghanistan         4 AFG      AFG52              700
 8 Afghanistan     4  1953 Afghanistan         4 AFG      AFG53              700
 9 Afghanistan     4  1954 Afghanistan         4 AFG      AFG54              700
10 Afghanistan     4  1955 Afghanistan         4 AFG      AFG55              700
# i 11,637 more rows
# i 406 more variables: ccodevdem <dbl>, cname_year <chr>, version <chr>,
#   act_act <dbl>, as_rifr <dbl>, as_ws <dbl>, bti_envc <dbl>, ccci_coop <dbl>,
#   ccci_em <dbl>, ccci_fin <dbl>, ccci_kyoto <dbl>, ccci_rep <dbl>,
#   ccci_unfccc <dbl>, cckp_rain <dbl>, cckp_temp <dbl>, ccl_exepp <dbl>,
#   ccl_leglp <dbl>, ccl_lpp <dbl>, ccl_mitlpp <dbl>, ccl_nexep <dbl>,
#   ccl_nlegl <dbl>, ccl_nlp <dbl>, ccl_nmitlp <dbl>, edgar_bc <dbl>, ...
```

```
# Filter for multiple countries
filter(qog, cname %in% c("France", "Norway", "Spain", "Denmark"))
```

```
# A tibble: 300 x 414
   cname   ccode  year cname_qog ccode_qog ccodealp ccodealp_year ccodecow
   <chr>    <dbl> <dbl> <chr>          <dbl> <chr>    <chr>             <dbl>
 1 Denmark    208  1946 Denmark          208 DNK      DNK46               390
 2 Denmark    208  1947 Denmark          208 DNK      DNK47               390
 3 Denmark    208  1948 Denmark          208 DNK      DNK48               390
 4 Denmark    208  1949 Denmark          208 DNK      DNK49               390
 5 Denmark    208  1950 Denmark          208 DNK      DNK50               390
 6 Denmark    208  1951 Denmark          208 DNK      DNK51               390
 7 Denmark    208  1952 Denmark          208 DNK      DNK52               390
 8 Denmark    208  1953 Denmark          208 DNK      DNK53               390
 9 Denmark    208  1954 Denmark          208 DNK      DNK54               390
10 Denmark    208  1955 Denmark          208 DNK      DNK55               390
# i 290 more rows
# i 406 more variables: ccodevdem <dbl>, cname_year <chr>, version <chr>,
#   act_act <dbl>, as_rifr <dbl>, as_ws <dbl>, bti_envc <dbl>, ccci_coop <dbl>,
#   ccci_em <dbl>, ccci_fin <dbl>, ccci_kyoto <dbl>, ccci_rep <dbl>,
#   ccci_unfccc <dbl>, cckp_rain <dbl>, cckp_temp <dbl>, ccl_exepp <dbl>,
#   ccl_leglp <dbl>, ccl_lpp <dbl>, ccl_mitlpp <dbl>, ccl_nexep <dbl>,
#   ccl_nlegl <dbl>, ccl_nlp <dbl>, ccl_nmitlp <dbl>, edgar_bc <dbl>, ...
```

```
# Filter for countries and year and save in a new object !
qog_subset <- filter(qog, cname %in% c("France", "Norway", "Spain", "Denmark"), year > 1990)
```

Often however, we need several operations to be done together. To do so with the tidyverse functions, we can use something that is called the pipe : `|>`. You will quickly understand what the pipe is, we use it to chain instructions just as in a recipe. Depending on the version of R, you might also use/see this `%>%`. The shortcut for it is `Ctrl + Shift + M` or `Cmd + Shift + M` on Mac.

Let's say we want to count the number of observations we have on the variable **`wvs_pedp`** that measure the share of the population participating in environmental protests for Sweden. We can use the pipe to do it by chaining the `filter()` and `count()` functions rather than doing it separately in different steps.

```
qog |>
  # Keep only countries
  filter(cname == "Sweden") |>
  # Count the number of observations for each country
  count(wvs_pedp) # Use Sort = TRUE if you want to sort the results
```

```
# A tibble: 2 x 2
  wvs_pedp     n
     <dbl> <int>
```

```
1      5.22     1
2    NA        74
```

4.1.3 Your turn

Count the countries and year where the share of the population participating in environmental protests was greater than 10 (measured by the `wvs_pedp` variable).

4.2 `select()` variables

While `filter()` is useful for keeping only certain groups of rows, `select()` is used, as its name suggests, to select certain variables (columns) from our dataframe.

Let's say I'm interested in CO2 emissions per capita, their overall level and how they vary over time and by country. For this, I would need only three variables that are present in the dataset: `year`, `cname` and the variable `wdi_co2` which comes from the World Bank's World Development Indicators. To do this, I use `select()` and just specify which variables of the dataset I want to select.

```r
qog_subset |>
  select(cname, year,  wdi_co2)
```

```
# A tibble: 120 x 3
   cname    year wdi_co2
   <chr>   <dbl>   <dbl>
 1 Denmark  1991   11.7
 2 Denmark  1992   10.5
 3 Denmark  1993   11.0
 4 Denmark  1994   11.7
 5 Denmark  1995   10.9
 6 Denmark  1996   13.7
 7 Denmark  1997   11.6
 8 Denmark  1998   11.2
 9 Denmark  1999   10.4
10 Denmark  2000    9.61
# i 110 more rows
```

4.3 Change column names with `rename()`

Occasionally, you may encounter datasets with columns named in a non-meaningful manner, such as v5, x45, or Q234, often representing variable identifiers from surveys. To enhance the understanding of the variables' meanings, it can prove beneficial to rename these columns using the rename() function. In our current dataset, the variable names are already quite informative. However, for the purpose of this illustration, let's rename the 'cname' variable as 'country', offering a more explicit label.

```r
qog_renamed <- qog |>
  # Rename the variable cname as country
  rename(country = cname)

qog_renamed |>
  head()
```

```
# A tibble: 6 x 414
  country     ccode  year cname_qog   ccode_qog ccodealp ccodealp_year ccodecow
  <chr>       <dbl> <dbl> <chr>           <dbl> <chr>    <chr>            <dbl>
1 Afghanistan     4  1946 Afghanistan         4 AFG      AFG46              700
2 Afghanistan     4  1947 Afghanistan         4 AFG      AFG47              700
3 Afghanistan     4  1948 Afghanistan         4 AFG      AFG48              700
4 Afghanistan     4  1949 Afghanistan         4 AFG      AFG49              700
5 Afghanistan     4  1950 Afghanistan         4 AFG      AFG50              700
6 Afghanistan     4  1951 Afghanistan         4 AFG      AFG51              700
# i 406 more variables: ccodevdem <dbl>, cname_year <chr>, version <chr>,
#   act_act <dbl>, as_rifr <dbl>, as_ws <dbl>, bti_envc <dbl>, ccci_coop <dbl>,
#   ccci_em <dbl>, ccci_fin <dbl>, ccci_kyoto <dbl>, ccci_rep <dbl>,
#   ccci_unfccc <dbl>, cckp_rain <dbl>, cckp_temp <dbl>, ccl_exepp <dbl>,
#   ccl_leglp <dbl>, ccl_lpp <dbl>, ccl_mitlpp <dbl>, ccl_nexep <dbl>,
#   ccl_nlegl <dbl>, ccl_nlp <dbl>, ccl_nmitlp <dbl>, edgar_bc <dbl>,
#   edgar_ch4 <dbl>, edgar_co <dbl>, edgar_co2gdp <dbl>, edgar_co2pc <dbl>, ...
```

4.4 Calculating statistics by group with `group_by()` and `summarise()`

To compute a first set of descriptive statistics, we could look at the central indicators of the co2 emissions variable (`wdi_co2`).

```
mean(qog_subset$wdi_co2, na.rm = TRUE) # Compute the mean
```

```
[1] 7.666003
```

```
median(qog_subset$wdi_co2, na.rm = TRUE) # Compute the median
```

```
[1] 7.496984
```

```
sd(qog_subset$wdi_co2, na.rm = TRUE) # Compute the standard deviation
```

```
[1] 2.003383
```

However, we are often interested in how these indicators vary across groups such as country or year. To do so, use the `summarise()` function in R that allows you to compute new variables by groups. To choose grouping variable, we use first `group_by()` where we specify for which group we want to compute something. Here I use this function to compute the mean, the median, the standard deviation and the first and third quartile of co2 emissions per capita for each country. This gives us a new tibble with all of the summary information.

```
qog_stats <- qog |>
   group_by(cname) |>
  summarise(mean_co2 = mean(wdi_co2, na.rm = TRUE),
            median_co2 = median(wdi_co2, na.rm = TRUE),
            sd_co2 = sd(wdi_co2, na.rm = TRUE))

qog_stats
```

```
# A tibble: 204 x 4
   cname               mean_co2 median_co2 sd_co2
   <chr>                  <dbl>      <dbl>  <dbl>
 1 Afghanistan            0.148      0.135 0.0892
 2 Albania                1.65       1.56  0.649
 3 Algeria                2.62       2.98  0.981
 4 Andorra                6.82       6.75  0.700
 5 Angola                 0.754      0.582 0.354
 6 Antigua and Barbuda    4.71       4.61  0.986
 7 Argentina              3.69       3.67  0.615
 8 Armenia                1.37       1.46  0.386
 9 Australia             14.6       15.4   2.71
10 Austria                7.19       7.39  1.08
# i 194 more rows
```

It is also possible to sort the results to gain a clearer idea of which countries have the highest average CO2 emissions per capita. To achieve this, we utilize the `arrange()` function, specifying the variable by which we intend to sort.

```r
qog_stats |>
  arrange(-median_co2)
```

```
# A tibble: 204 x 4
   cname                     mean_co2 median_co2 sd_co2
   <chr>                        <dbl>      <dbl>  <dbl>
 1 Qatar                         51.5       54.6  14.8
 2 United Arab Emirates          33.6       29.2  14.0
 3 Kuwait                        28.1       27.4  12.9
 4 Luxembourg                    27.3       25.0   7.41
 5 Bahrain                       23.9       23.7   2.71
 6 United States of America      19.0       19.3   1.76
 7 Brunei Darussalam             17.4       16.5   4.65
 8 Canada                        15.8       16.2   1.84
 9 Australia                     14.6       15.4   2.71
10 Trinidad and Tobago           17.8       15.3   9.86
# i 194 more rows
```

```r
qog_stats |>
  arrange(mean_co2)
```

```
# A tibble: 204 x 4
   cname                     mean_co2 median_co2 sd_co2
   <chr>                        <dbl>      <dbl>  <dbl>
 1 Burundi                     0.0324     0.0354 0.0112
 2 Chad                        0.0516     0.0464 0.0186
 3 Ethiopia                    0.0574     0.0546 0.0264
 4 Rwanda                      0.0624     0.0645 0.0298
 5 Central African Republic    0.0678     0.0643 0.0154
 6 Mali                        0.0679     0.0523 0.0402
 7 Niger                       0.0722     0.0637 0.0339
 8 Burkina Faso                0.0725     0.0662 0.0447
 9 Somalia                     0.0762     0.0620 0.0346
10 Uganda                      0.0797     0.0632 0.0353
# i 194 more rows
```

4.5 Your turn

Now I want to know when emissions per capita levels were the lowest in Saudia Arabia. I can filter only this country, select the variables of interest and sort the dataframe. We observe that the CO2 emissions per capita were at their minimum towards the late 1990s (which is not good).

4.5.1 Mutate

We can also create new variables based on other ones with `mutate()`. Let's say I want to compute the growth rate of co2 emissions per capita every year. For this, I will use `mutate()` to create new variables.

```
qog |>
  select(cname, year, wdi_fossil) |>
  mutate(dataset = "QOG dataset")
```

```
# A tibble: 11,722 x 4
   cname        year wdi_fossil dataset
   <chr>       <dbl>      <dbl> <chr>
 1 Afghanistan  1946         NA QOG dataset
 2 Afghanistan  1947         NA QOG dataset
 3 Afghanistan  1948         NA QOG dataset
 4 Afghanistan  1949         NA QOG dataset
 5 Afghanistan  1950         NA QOG dataset
 6 Afghanistan  1951         NA QOG dataset
 7 Afghanistan  1952         NA QOG dataset
 8 Afghanistan  1953         NA QOG dataset
 9 Afghanistan  1954         NA QOG dataset
10 Afghanistan  1955         NA QOG dataset
# i 11,712 more rows
```

To compute the growth rate, we need for each year the emissions per capita of that year and those from the previous year, which I access with `lag()`.

```
qog_co2_growth <- qog |>
  filter(cname == "France") |>
  select(cname, year, wdi_co2) |>
  mutate(co2_lag = lag(wdi_co2),
         co2_growth = (wdi_co2 - co2_lag) / co2_lag*100)
```

```r
qog_co2_growth
```

```
# A tibble: 75 x 5
   cname   year wdi_co2 co2_lag co2_growth
   <chr>  <dbl>   <dbl>   <dbl>      <dbl>
 1 France  1963    6.88   NA         NA
 2 France  1964    7.01    6.88       2.02
 3 France  1965    7.06    7.01       0.695
 4 France  1966    6.90    7.06      -2.32
 5 France  1967    7.33    6.90       6.19
 6 France  1968    7.52    7.33       2.59
 7 France  1969    8.01    7.52       6.64
 8 France  1970    8.45    8.01       5.42
 9 France  1971    8.83    8.45       4.50
10 France  1972    9.11    8.83       3.13
# i 65 more rows
```

4.6 Recode values with `case_when`

`Mutate()` is also widely used to recode variable. Here we create a new `trajectory` variable and we use `case_when()` to recode whether there co2_emissions are growing or not for a given year.

```r
qog_co2_growth <- qog_co2_growth |>
  filter(cname == "France") |>
  mutate(
    trajectory = case_when(
      co2_growth > 0 ~ "Bad",
      co2_growth < 0 ~ "Good"
    )
  )
qog_co2_growth
```

```
# A tibble: 75 x 6
   cname   year wdi_co2 co2_lag co2_growth trajectory
   <chr>  <dbl>   <dbl>   <dbl>      <dbl> <chr>
 1 France  1963    6.88   NA         NA     <NA>
 2 France  1964    7.01    6.88       2.02  Bad
 3 France  1965    7.06    7.01       0.695 Bad
 4 France  1966    6.90    7.06     -2.32  Good
 5 France  1967    7.33    6.90      6.19  Bad
 6 France  1968    7.52    7.33      2.59  Bad
 7 France  1969    8.01    7.52      6.64  Bad
 8 France  1970    8.45    8.01      5.42  Bad
 9 France  1971    8.83    8.45      4.50  Bad
10 France  1972    9.11    8.83      3.13  Bad
# i 65 more rows
```

```r
qog_co2_growth |>
  count(trajectory)
```

```
# A tibble: 3 x 2
  trajectory     n
  <chr>      <int>
1 Bad           22
2 Good          31
3 <NA>          22
```

4.6.1 Your turn !

Create a century variable that indicates whether the year is in the 20th or 21st century. Find the countries that have the highest total number of deaths due to natural disasters for each century, as measured by the `emdat_ndeath` variable.

5 Reshaping data

The analysis of your data is usually not the most time-consuming part of your project; data wrangling is. Your data must be cleaned before it can be analyzed. This means having a *tidy* dataset, where each variable is a column, each observation is a row, and each cell contains a single value. Sometimes, the datasets you find are not tidy, and you will need to reshape them. In other cases, you may encounter datasets organized differently, representing the same data in a less useful format for analysis. Here, we focus on how to combine and reshape different datasets to use them properly. I will use polling data from various political parties.

```r
library(tidyverse)

german_polls <- read_csv("data/german_polls.csv")
```

5.1 Reshaping with `pivot_longer()` and `pivot_wider()`

Datasets can be in long (many rows, few columns) or wide formats (few rows, many columns). Depending on our unit of analysis, we can reshape our datasets into different formats. In the `german_polls dataset`, the vote intentions for each party are stored in different columns. However, we could also structure the data with one column for parties and another for voting intention.

```r
german_long <- german_polls |>
  pivot_longer(
    # Select which columns to pivot
    cols = c(Union, FDP, LINKE, SPD, PIRATEN, AfD, GRUENE),
    # Choose a name for the new column with all of the parties
    names_to = "party",
    # Choose a name for the new column with the vote intentions
    values_to = "intention"
  )

german_long
```

```
# A tibble: 24,115 x 6
   date       firm      date_from  sample_size party   intention
   <date>     <chr>     <date>           <dbl> <chr>       <dbl>
 1 2005-09-22 FG Wahlen 2005-09-20        1345 Union          37
 2 2005-09-22 FG Wahlen 2005-09-20        1345 FDP             8
 3 2005-09-22 FG Wahlen 2005-09-20        1345 LINKE           8
 4 2005-09-22 FG Wahlen 2005-09-20        1345 SPD            35
 5 2005-09-22 FG Wahlen 2005-09-20        1345 PIRATEN        NA
 6 2005-09-22 FG Wahlen 2005-09-20        1345 AfD            NA
 7 2005-09-22 FG Wahlen 2005-09-20        1345 GRUENE          8
 8 2005-10-06 FG Wahlen 2005-10-04        1259 Union          39
 9 2005-10-06 FG Wahlen 2005-10-04        1259 FDP             7
10 2005-10-06 FG Wahlen 2005-10-04        1259 LINKE           9
# i 24,105 more rows
```

```
german_long |>
  group_by(date, party) |>
  summarise(intention = mean(intention, na.rm = T))
```

```
`summarise()` has grouped output by 'date'. You can override using the
`.groups` argument.
```

```
# A tibble: 18,788 x 3
# Groups:   date [2,684]
   date       party   intention
   <date>     <chr>       <dbl>
 1 2005-09-22 AfD           NaN
 2 2005-09-22 FDP             8
 3 2005-09-22 GRUENE          8
 4 2005-09-22 LINKE           8
 5 2005-09-22 PIRATEN       NaN
 6 2005-09-22 SPD            35
 7 2005-09-22 Union          37
 8 2005-10-06 AfD           NaN
 9 2005-10-06 FDP             7
10 2005-10-06 GRUENE          7
# i 18,778 more rows
```

We can also reshape the dataset to put it back on wide format.

```r
german_wide <- german_long |>
  pivot_wider(
    names_from = party,
    values_from = intention
  ) |>
  unnest()

german_wide
```

```
# A tibble: 3,445 x 11
   date       firm  date_from  sample_size Union   FDP LINKE   SPD PIRATEN   AfD
   <date>     <chr> <date>           <dbl> <dbl> <dbl> <dbl> <dbl>   <dbl> <dbl>
 1 2005-09-22 FG W~ 2005-09-20        1345    37     8     8    35      NA    NA
 2 2005-10-06 FG W~ 2005-10-04        1259    39     7     9    34      NA    NA
 3 2005-10-13 FG W~ 2005-10-11        1280    38     8     8    35      NA    NA
 4 2005-10-27 FG W~ 2005-10-25        1269    37     9     8    35      NA    NA
 5 2005-11-10 FG W~ 2005-11-08        1230    36     9     9    33      NA    NA
 6 2005-11-24 FG W~ 2005-11-22        1298    37     9     8    34      NA    NA
 7 2005-12-08 FG W~ 2005-12-06        1237    38     9     8    34      NA    NA
 8 2006-01-12 FG W~ 2006-01-10        1249    39     9     8    33      NA    NA
 9 2006-01-26 FG W~ 2006-01-24        1279    41     8     8    33      NA    NA
10 2006-02-16 FG W~ 2006-02-14        1260    41     8     7    32      NA    NA
# i 3,435 more rows
# i 1 more variable: GRUENE <dbl>
```

5.2 Combining with bind_rows

In many cases, you want to combine datasets having the same structure and mostly the same variables but different observations (rows). It may be the same data from different years, different countries etc. Here, I want to combine the data I have on Germany with similar data on spanish polls. To do so, I first, import this dataset and convert it also to a long format.

```r
spain_polls <- read_csv("data/spanish_polls.csv")
```

```
Rows: 1414 Columns: 22
-- Column specification --------------------------------------------------------
Delimiter: ","
chr   (1): firm
dbl  (19): PP, PSOE, ERC, PNVEAJ, CC, BNG, Cs, VOX, Podemos, EHBildu, PACMA,...
date  (2): date, date_from
```

```
i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
spain_long <- spain_polls |>
  pivot_longer(
    # Select which columns NOT to pivot
    cols = -c(date, firm, date_from, sample_size),
    # Choose a name for the new column with all of the parties
    names_to = "party",
    # Choose a name for the new column with the vote intentions
    values_to = "intention"
  )

spain_long
```

```
# A tibble: 25,452 x 6
   date       firm        date_from  sample_size party   intention
   <date>     <chr>       <date>           <dbl> <chr>       <dbl>
 1 2011-12-15 Metroscopia 2011-12-14        1000 PP           44.9
 2 2011-12-15 Metroscopia 2011-12-14        1000 PSOE         28.4
 3 2011-12-15 Metroscopia 2011-12-14        1000 ERC          NA
 4 2011-12-15 Metroscopia 2011-12-14        1000 PNVEAJ       NA
 5 2011-12-15 Metroscopia 2011-12-14        1000 CC           NA
 6 2011-12-15 Metroscopia 2011-12-14        1000 BNG          NA
 7 2011-12-15 Metroscopia 2011-12-14        1000 Cs           NA
 8 2011-12-15 Metroscopia 2011-12-14        1000 VOX          NA
 9 2011-12-15 Metroscopia 2011-12-14        1000 Podemos      NA
10 2011-12-15 Metroscopia 2011-12-14        1000 EHBildu      NA
# i 25,442 more rows
```

My goal now is to combine them together. But before, I add a country variable on the two datasets to keep track of which country the data is from.

```r
spain_long <- spain_long |> mutate(country = "Spain")
german_long <- german_long |> mutate(country = "Germany")
```

Now, I combine the two datasets with the bind_rows function that comes from the dplyr package. The functions allows you to combine two datasets with the same structure by stacking them on top of each other. The resulting dataset will have all the rows from the first dataset, followed by all the rows from the second dataset. The datasets will be combined by columns, so the columns must have the same names and types.

```r
polls <- bind_rows(spain_long, german_long)
```

Now I can for instance plot the evolution of the vote intentions for the mainstream right and radical right party in both countries with the following code. We will see more on visualization in the next session.

Part III

Visualizing data

Part IV

Testing relationships

Part V

Regression analysis

Part VI

Multivariate analysis