Social Science Research Methods

Barış Arı

2024-10-01

Table of contents

First Task

The module has two in-person teaching components:

1. Lectures (one-hour on Monday)
2. Applied data analysis: IT lab (two-hours on Wednesday/Friday)

Attendance is mandatory for all teaching sessions. If you cannot attend to any of the sessions, please make sure to submit an extenuating circumstances through eVision.

We will use this website for learning applied data analysis. **The website is not a substitute for module Blackboard.** We will use this site in conjunction with Blackboard.

Our very first task is to install R and R Studio on our laptops.

Please try do this before coming to next lab.

R and R Studio are very powerful tools for analysing data and for creating high-quality documents. I prepared this website using R Studio. It is widely used both in academic research and in commercial enterprise. Learning the fundamentals of these powerful tools gives you an advantage in the job market (or for pursuing further studies such as PhD). They are free and open source.

Make sure to install R first and then the R Studio.

1. R can be installed here: https://cran.r-project.org/
2. R Studio can be install here: https://posit.co/downloads/

Instructions for installing R and R Studio are available in Appendix A of the Online Textbook Hands-on Programming with R.

1 R Basics

R is a free software environment for statistical computing and graphics. It is an extremely powerful tool that we will use for data analysis and visualisations.

R Studio is a customization of R. It runs R in the background and comes up with some additional features such as a very nice text editor.

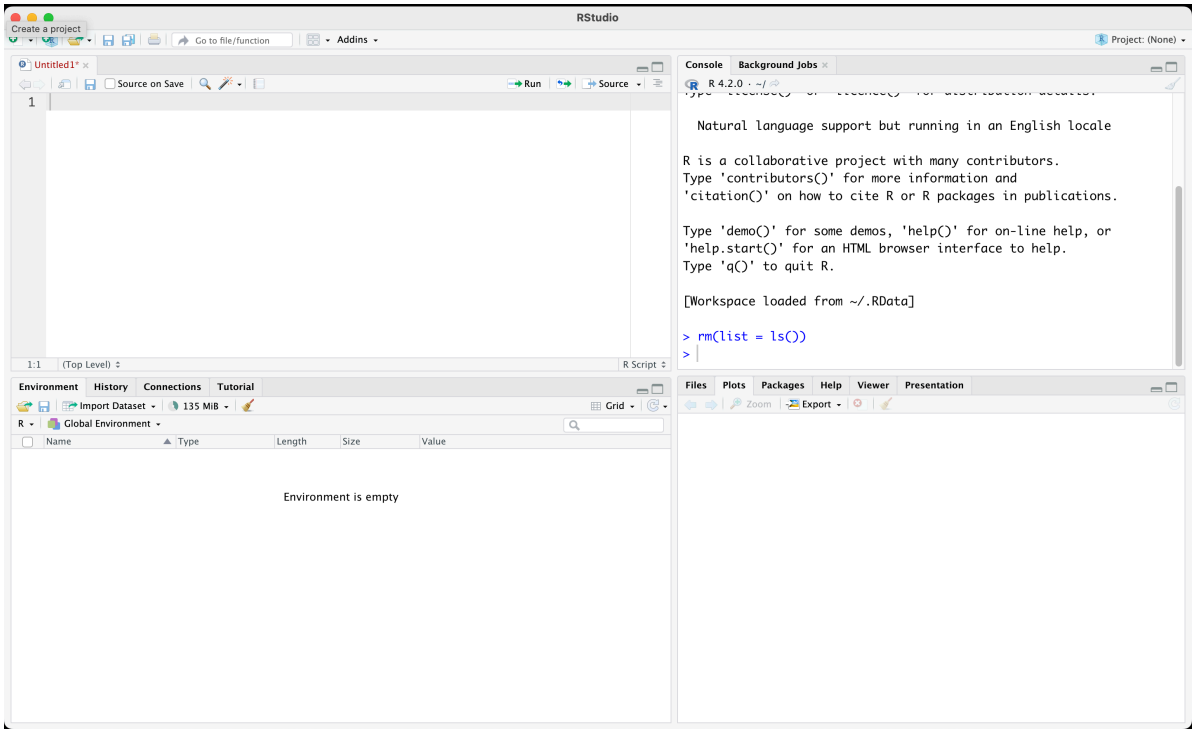


Figure 1.1: R Studio screen with four panes

R Studio has four panes:

1. Console
2. Text editor to work on R scripts
3. Environment window
4. Plot display

As default, the bottom left pane shows the console. This is where you can type your commands to R. However, we will discourage directly typing into the console. Instead, we will use the text editor, which is located above the console pane. This is where we will type our commands (and also comments). We will then run our code from the text editor. This will allow us to track what we have done. We can easily edit the code if we made a mistake. We can also save the commands for future use.

The upper right pane usually shows the environment by default. This is where our objects such as our data will be shown.

Your environment would not include anything when you first open R Studio. It is the case because we have not imported any data or created any objects.

Finally, the lower right pane displays files/folders, plots (that we will prepare) and help files. We will discuss this pane more in the future.

1.1 Objectives for Week 1

1. Use R as a calculator
2. Write and execute a command by using R Studio text editor
3. Save your script
4. Use the assignment operator to create objects
5. Understand the difference between 'string' and 'numerical'
6. Create a simple dataset

1.2 Use R as a calculator

Go to the console pane and type a simple calculation.

```
1 + 3
#> [1] 4
```

As you can see, the output for `1 + 3` is `4`, which is correct. We directly did a calculation using the console.

This would work, but it is not a good approach. **Do not write your code directly to the console.** Instead, go to the top left pane and write your 'code' into the text editor. The calculation 1 + 3 here is your code.

- Save your script by `File >> Save` OR simply by pressing `Ctrl-S` (linux), `Command-S` (mac), `Control-S` (windows). It is a good idea to create a folder/directory for this module and give your script an intuitive name such as *learn_01.R*.

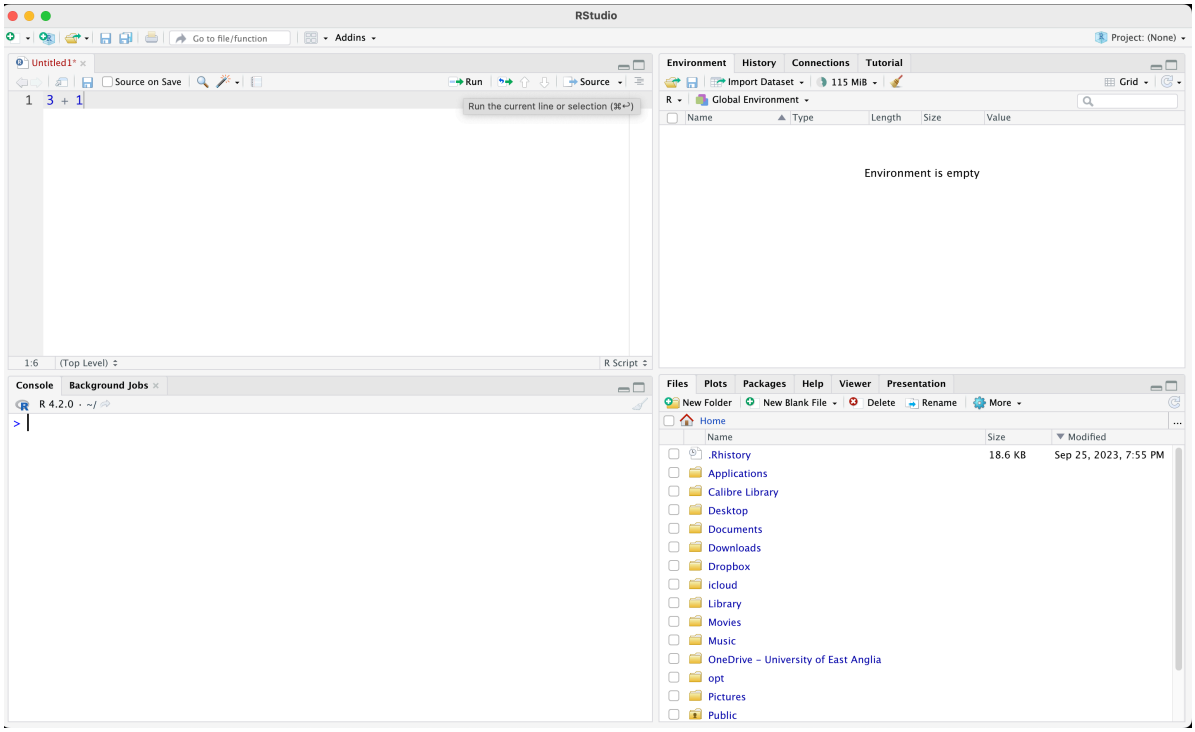


Figure 1.2: Our first calculation

1.3 Assignment operator to create objects

We can create objects in R which store our data. For example, you would like to calculate your age. Current year (i.e., 2024) - your birth year gives your age.

Let's create an object which stores your year of birth. We are going to call it my_birth_year. Each R object must be one-word only, so I use _ instead of space. We could also have used a dot or dash.

```
# This is a comment.
# Characters after a hashtag are considered as comments by R.
# They are not executed.
# Use comments extensively to take notes
# and to remind your future self of the work you did.

# "<-" is the assignment operator
# It basically symbolizes an arrow.

my_birth_year <- 1985
```

Now the Environment should store an object called `my_birth_year`. When I run my_birth_year, R will display the information stored.

```
my_birth_year
#> [1] 1985
```

Note that R is case sensitive. If you mistype, such as `My_birth_year`, it will give you an error message.

```
My_birth_year
#> Error: object 'My_birth_year' not found
```

We can find your age by subtracting current year from `my_birth_year`.

```
2024 - my_birth_year
#> [1] 39
```

We typed 2024 manually. We might want to create another object called `current_year`. Try to do it yourself, as an exercise.

```
current_year <- 2024
```

You can do operations using objects. For example, calculate your age using the objects `current_year` and `my_birth_year`. Store this in another object called `my_age`.

```
my_age <- current_year - my_birth_year
```

Check if you did correctly.

```
my_age
#> [1] 39
```

You can also write over an object.

```
current_year <- 2030
current_year
#> [1] 2030
```

This would not change outputs previously created using the older version of the objects.

```
my_age
#> [1] 39
```

Obviously, current year is not 2030, so let's correct it back.

```
current_year <- 2024
```

1.4 Numerical and String Objects

So far, we stored numerical data. We can also have textual information, such as name of a person, or type of a medicine.

Create an object called `my_name` and store your name there.

```
my_name <- "Baris"
my_name
#> [1] "Baris"
```

As you can see, R displays textual information within quotation (“ “). Any information stored or displayed within ’ ’ is called a string and refers to text.

Create an object called `my_name_last` and store your name there.

```
my_last_name <- "Ari"
```

Obviously, you cannot make a calculation using words. It is nonsensical to subtract two words. You cannot do any calculation with words.

```
my_name_last - my_name
#> Error: object 'my_name_last' not found
```

Sometimes numerical information is stored as text. In that case, R will not consider it as a number. For example, see three objects below.

```
num1 <- 10
num2 <- 100
num3 <- "1000"
```

`num1` and `num2` are numerical values, but `num3` is text. You cannot do any calculation with that.

```
num1
#> [1] 10

num2
#> [1] 100

num3
#> [1] "1000"

num1 + num2
#> [1] 110

num1 + num3
#> Error in num1 + num3: non-numeric argument to binary operator
```

1.5 Create a simple dataset

Imagine that we have the names and birth years of a number of people. We cannot really hold each piece of information in separate objects. We would like to store them altogether in a single object, like a spreadsheet.

Let's start with names. We have eight people:

1. Keir Starmer
2. Rishi Sunak
3. Liz Truss
4. Boris Johnson
5. Theresa May
6. David Cameron
7. Gordon Brown
8. Tony Blair

We can store their full names in a single object using the combine function `c()`.

```
names_pm <- c("Keir Starmer",
              "Rishi Sunak",
              "Liz Truss",
              "Boris Johnson",
              "Theresa May",
              "David Cameron",
              "Gordon Brown",
              "Tony Blair")
```

Note that each PMs name is written within quotation and they are combined together with the function `c()`. Each item within `c()` is separated with a comma. Let's see the object:

```
names_pm
#> [1] "Keir Starmer"  "Rishi Sunak"   "Liz Truss"     "Boris Johnson"
#> [5] "Theresa May"   "David Cameron" "Gordon Brown"  "Tony Blair"
```

Great! We have the names of the last six UK PMs.

You may have realized that there are numbers in squared brackets in the beginning of each line.

These numbers refer to the order in the sequence. For example, "Keir Starmer" is the first item whereas "Theresa May" is the fifth.

You can recall a particular item in the object using square brackets. Let's print the first item in `names_pm`.

```
names_pm[1]
#> [1] "Keir Starmer"
```

Similarly, for the third item, you would use [3]:

```
names_pm[3]
#> [1] "Liz Truss"
```

Find the fifth name in the object.

```
names_pm[5]
#> [1] "Theresa May"
```

You can add more than one number into the square brackets using the `c()` function. For example, who are the second and fourth names?

```
names_pm[c(2,4)]
#> [1] "Rishi Sunak"   "Boris Johnson"
```

Next, let's write down their birth year. The order is important! You need to keep the same order with PMs.

```
birth_years <- c(1962, # Keir Starmer
                 1980, # Rishi Sunak
                 1975, # Liz Truss
                 1964, # Boris Johnson
                 1956, # Theresa May
                 1966, # David Cameron
                 1951, # Gordon Brown
                 1953  # Tony Blair)
                 )
```

Check the object we just created.

```
birth_years
#> [1] 1962 1980 1975 1964 1956 1966 1951 1953
```

Let's put them together in a spreadsheet. What we would like to do is to vertically bind the two objects, which is called column bind and denoted with `cbind()`.

```
cbind(names_pm, birth_years)
#>      names_pm        birth_years
#> [1,] "Keir Starmer"  "1962"
#> [2,] "Rishi Sunak"   "1980"
#> [3,] "Liz Truss"     "1975"
#> [4,] "Boris Johnson" "1964"
#> [5,] "Theresa May"   "1956"
#> [6,] "David Cameron" "1966"
#> [7,] "Gordon Brown"  "1951"
#> [8,] "Tony Blair"    "1953"
```

So far, we just printed this on our screen but we have not stored it in an object. Put this into an object.

```
my_data <- cbind(names_pm, birth_years)
```

Check `my_data`.

```
my_data
#>      names_pm        birth_years
#> [1,] "Keir Starmer"  "1962"
#> [2,] "Rishi Sunak"   "1980"
#> [3,] "Liz Truss"     "1975"
#> [4,] "Boris Johnson" "1964"
#> [5,] "Theresa May"   "1956"
#> [6,] "David Cameron" "1966"
#> [7,] "Gordon Brown"  "1951"
#> [8,] "Tony Blair"    "1953"
```

Note that birth_years are stored as text, not numbers. I know this because they are within quotation marks.

It is customary to keep spreadsheets as something called "data frames" in R. This will not change our data, but makes further operations easier by unlocking some of the features of R.

```
my_data <- as.data.frame(my_data)
```

We can take a better look at the dataset using `View()` function.

```
# View(my_data)
```

Let's save our script.

1.5.1 Variables in a data frame

Columns in a data frame are also called variables. We have two variables in the dataset:

- names_pm : Name of the UK PM
- birth_years: Birth year of the PM

There are a few ways to access a variable. A straightforward approach is to use the `$` notation:

```
# 'name of the data frame'$'name of the variable'
my_data$names_pm
#> [1] "Keir Starmer"  "Rishi Sunak"   "Liz Truss"     "Boris Johnson"
#> [5] "Theresa May"   "David Cameron" "Gordon Brown"  "Tony Blair"
```

Now it is your turn. Display the `birth_years` variable.

```
my_data$birth_years
#> [1] "1962" "1980" "1975" "1964" "1956" "1966" "1951" "1953"
```

You can think this expression as a sentence in R. In plain English, this expression tells R to bring the variable `names_pm` within the data frame `my_data`. The symbol `$` refers to the 'within' part of this sentence.

Just like you can convey the same meaning using different sentence structures, there are different ways to do the same thing in R. This is because R is working exactly like a language: it is a language to communicate with the computer.

Another way is using the square brackets notation `[]`. `names_pm` is the first column in the data frame. To get the variable, you could type the following:

```
my_data[,1]
#> [1] "Keir Starmer"  "Rishi Sunak"   "Liz Truss"     "Boris Johnson"
#> [5] "Theresa May"   "David Cameron" "Gordon Brown"  "Tony Blair"
```

Note that we did not simply write `my_data[1]`. There is a comma: `my_data[,1]`

In a spreadsheet, we have two dimensions: rows and columns. By convention, rows are considered as the first dimension, and columns are considered as the second. This is why we had to use a comma to designate that we are interested in columns. If left the first dimension unspecified, which tells R to bring everything.

If you want to get the first row, you would type the following:

```
my_data[1, ]
#>       names_pm birth_years
#> 1 Keir Starmer        1962
```

Try it yourself; get the fourth row.

```
my_data[4,]
#>        names_pm birth_years
#> 4 Boris Johnson        1964
```

Let's put these together: you can tell R to bring a specific observation. For example, third row of second column.

```
my_data[3,2]
#> [1] "1975"
```

You can also ask for multiple items by plugging in the combine function.

```
my_data[c(3,4), 2]
#> [1] "1975" "1964"
```

Consider the command above. Try to formulate it in plain English. What does it tell to do R?

2 Data in R

Last week we started with a gentle introduction to R. We created a basic dataset, with the names of UK Prime Ministers and their respective birthdays.

Let's remember the steps. First, we created an object called `pm_names`. We did this by using the function `c()` which refers to combine.

```
names_pm <- c("Keir Starmer",
              "Rishi Sunak",
              "Liz Truss",
              "Boris Johnson",
              "Theresa May",
              "David Cameron",
              "Gordon Brown",
              "Tony Blair")
```

The object `pm_names` holds the information we feed into: the names of the Prime Ministers. Let's check the object.

```
names_pm
#> [1] "Keir Starmer"  "Rishi Sunak"   "Liz Truss"     "Boris Johnson"
#> [5] "Theresa May"   "David Cameron" "Gordon Brown"  "Tony Blair"
```

This is basically the names of the last eight UK Prime Ministers. The number in square brackets refers to an item's position. For example [1] in front of Keir Starmer tells me that the first item in the object names_pm is Keir Starmer. Similarly the fifth item in names_pm is Theresa May.

Let's recall the square brackets notation [] which is used to get a specific item from an object.

```
# First item in names_pm is "Keir Starmer"
names_pm[1]
#> [1] "Keir Starmer"

# Fifth item in names_pm is "Theresa May"
names_pm[5]
```

```
#> [1] "Theresa May"

# First and fifth item in the names_pm
names_pm[c(1,5)]
#> [1] "Keir Starmer" "Theresa May"

# Fourth item in the names_pm
names_pm[4]
#> [1] "Boris Johnson"
```

> **Vectors**
>
> A series of information following each other is may called a *vector*. The object `names_pm` is a vector because it contains a series of information in a sequence. More specifically, `names_pm` is a vector of names. We will revisit the term *vector*.

Last week, we also created another object called birth_years, storing the information of birth years of each Prime Minister in names_pm. Also recall that the order of year of birth is important. For example, first item in birth_years should be Keir Starmers's year of birth, second item should be Rishi Sunak's, and so on.

```
birth_years <- c(1962, # Keir Starmer
                 1980, # Rishi Sunak
                 1975, # Liz Truss
                 1964, # Boris Johnson
                 1956, # Theresa May
                 1966, # David Cameron
                 1951, # Gordon Brown
                 1953  # Tony Blair)
                 )
```

The object `birth_years` is a numerical vector. It contains numbers. Let's check the object.

```
birth_years
#> [1] 1962 1980 1975 1964 1956 1966 1951 1953
```

2.1 Class of an object

R understand the differences between textual and numerical information. We can check the class of an object using the `class()` function.

```
# birth_years contain numerical information
class(birth_years)
#> [1] "numeric"

# names_pm contain textual information, which is called character in R
class(names_pm)
#> [1] "character"
```

2.2 Length of an object

We typed names of last eight Prime Ministers and their respective birth years. The number of items in names_pm and birth_years should be both eight. We can see the number of items in a vector by the `length()` function.

```
# Number of items in a vector can be seen by length()

# Length of names_pm:
length(names_pm)
#> [1] 8

# Length of birth_years:
length(birth_years)
#> [1] 8
```

2.3 *is equal to* operator

You can ask R whether two things are equal to each other or not. To do so, we are going to use the `==` operator, which means is equal to.

```
# is equal to operator: ==

# is the length of names_pm equal to birth_years
length(names_pm) == length(birth_years)
#> [1] TRUE
```

The number of items in both objects (`names_pm` and `birth_years`) is the same because both vectors contain eight pieces of information.

How about the class of the objects?

```
# is the class of names_pm equal to birth_years
class(names_pm) == class(birth_years)
#> [1] FALSE
```

The class of the objects is not the same because `names_pm` contains textual information whereas `birth_years` contains numerical information.

2.4 Creating a simple dataset

Last week we created a simple spreadsheet that looked like the data shown in

We can achieve this by doing a column bind which refers to vertically binding two vectors and can be done using the `cbind()` function.

```
my_data <- cbind(names_pm, birth_years)

# let's check the object we created
my_data
#>      names_pm        birth_years
#> [1,] "Keir Starmer"  "1962"
#> [2,] "Rishi Sunak"   "1980"
#> [3,] "Liz Truss"     "1975"
#> [4,] "Boris Johnson" "1964"
#> [5,] "Theresa May"   "1956"
#> [6,] "David Cameron" "1966"
#> [7,] "Gordon Brown"  "1951"
#> [8,] "Tony Blair"    "1953"
```

The first column in `my_data` is `names_pm` and the second column is `birth_years`. We have now two dimensions: columns and rows.

Recall that to ask R to bring a specific item in a two-dimensional object, such as a spreadsheet, we can use the square-brackets `[]` notation but we need to specify both dimension.

> **Rows and Columns in a Spreadsheet**
>
> First dimension refers to rows and second dimension refers to columns.

Let's get the third row in second column.

```
# Third row in second column
my_data[3,2]
#> birth_years
#>      "1975"
```

To sum up, we bound two vectors by column. Each column is a vector. We can call these column vectors. To get the first column, names_pm, we can use the square brackets notation.

```
# Bring the first column
my_data[,1]
#> [1] "Keir Starmer"  "Rishi Sunak"   "Liz Truss"     "Boris Johnson"
#> [5] "Theresa May"   "David Cameron" "Gordon Brown"  "Tony Blair"

# Bring the second column
my_data[,2]
#> [1] "1962" "1980" "1975" "1964" "1956" "1966" "1951" "1953"
```

To get a specific column vector, we left the first dimension unspecificed. Recall that the first dimension designates the row, so leaving it unspecified means *everything*.

We could also use column names instead of column numbers.

```
# Bring the column birth_years
my_data[,"birth_years"]
#> [1] "1962" "1980" "1975" "1964" "1956" "1966" "1951" "1953"

# Bring the column names_pm
my_data[,"names_pm"]
#> [1] "Keir Starmer"  "Rishi Sunak"   "Liz Truss"     "Boris Johnson"
#> [5] "Theresa May"   "David Cameron" "Gordon Brown"  "Tony Blair"
```

We can do the same for rows. To get a row vector, use the squared bracket notation.

```
# Bring the first row
my_data[1, ]
#>       names_pm    birth_years
#> "Keir Starmer"         "1962"

# Bring the third row
my_data[3,]
#>    names_pm birth_years
#> "Liz Truss"      "1975"
```

```
# Bring the fourth row
my_data[4,]
#>        names_pm     birth_years
#> "Boris Johnson"          "1964"
```

R will give you an error message if you go out of bounds.

```
# Bring the third column
my_data[,3]
#> Error in my_data[, 3]: subscript out of bounds

# Bring the 10th row

# Bring the second column, ninth row
my_data[9,2]
#> Error in my_data[9, 2]: subscript out of bounds
```

2.5 Data frame

It is customary to keep a spreadsheet-like looking data (i.e., two-dimensional) as something called a *data frame* in R. Let's check the class of my_data.

```
# Class of my_data
class(my_data)
#> [1] "matrix" "array"
```

It looks like the class of `my_data` is *matrix* and *array*. Matrix is a two-dimensional array.

We can turn my_data into a data frame.

```
# Turn my_data into data frame
my_data <- as.data.frame(my_data)
# this just overwrote my_data as a data frame

# Check its class
class(my_data)
#> [1] "data.frame"
```

In this module, we will primarily work with data frames.

Recall that we can use the `$` notation when working with data frames.

```
# bring names_pm
my_data$names_pm
#> [1] "Keir Starmer"  "Rishi Sunak"   "Liz Truss"     "Boris Johnson"
#> [5] "Theresa May"   "David Cameron" "Gordon Brown"  "Tony Blair"

# bring birth_years
my_data$birth_years
#> [1] "1962" "1980" "1975" "1964" "1956" "1966" "1951" "1953"

# bring the third item in birth_years
my_data$birth_years[3]
#> [1] "1975"
```

We can check the number of columns and the number of rows of our data frame by using `ncol()` and `nrow()` functions.

```
# number of columns
ncol(my_data)
#> [1] 2

# number of rows
nrow(my_data)
#> [1] 8
```

A data frame, such as `my_data`, has two dimensions: rows and columns. Note that `my_data` has eight rows and two columns. We can use the `dim()` function to get the length of each dimension.

```
dim(my_data)
#> [1] 8 2
```

2.6 Variable, Row, Observation

Let's check some more terminology that is frequently used in data analysis.

A *column vector* typically shows a variable. A *row vector* typically shows an observation. A particular item, which is a *cell* in a spreadsheet, is a value. This is visualised in Figure 2.1.

When data do not come in this format, we will carry out something called *data wrangling* and reorganize the data so that each column is a variable, each row is an observation and each cell is a value.

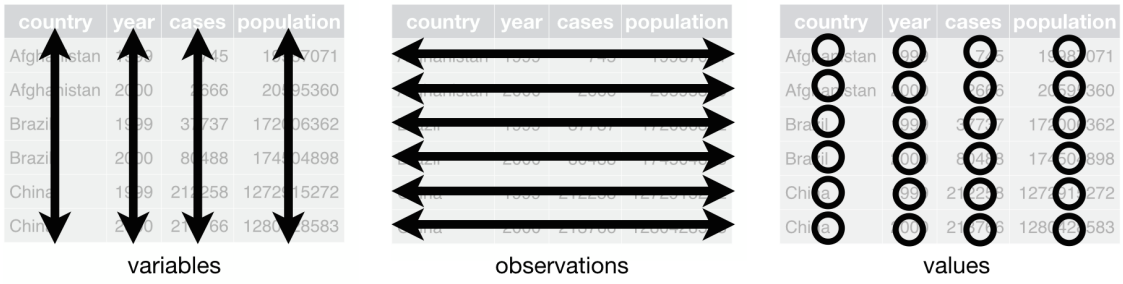


Figure 2.1: Variables, observations, rows. Well-organised data come in this form.

For simplicity, however, the datasets we are working on already come in this shape.

```
# A variable: a column (e.g., birth_years)
my_data$birth_years
#> [1] "1962" "1980" "1975" "1964" "1956" "1966" "1951" "1953"

# An observation: a row (e.g., second row)
my_data[2,]
#>      names_pm birth_years
#> 2 Rishi Sunak        1980

# A particular value (e.g., third row of second column)
my_data[3,2]
#> [1] "1975"
```

2.7 Numerical value stored as character

Let's say we would like to calculate each person's current age. We could simply tell R to subtract each birth year from current year (2024).

```
2024 - my_data$birth_years
#> Error in 2024 - my_data$birth_years: non-numeric argument to binary operator
```

Instead of the calculation, I get an error message: *non-numeric argument to…*! Let's see what is going on.

```
# Check the variable of interest
my_data$birth_years
#> [1] "1962" "1980" "1975" "1964" "1956" "1966" "1951" "1953"
```

`birth_years` is a vector of numbers but if you look closely, you will see that each number is shown within a pair of quotation mark. This is because R is keeping each number as text at the moment. Let's look at the class of the object.

```
# Class of birth_years
class(my_data$birth_years)
#> [1] "character"
```

Character means text. We are going to use `as.numeric()` function to tell R that information stored in `birth_years` is numerical, not text.

```
# Convert the variable to numerical
as.numeric(my_data$birth_years)
#> [1] 1962 1980 1975 1964 1956 1966 1951 1953
```

Now quotation marks disappeared. Beware: I have not overwritten the variable yet. It is just displayed on my screen for one time only. I need to overwrite the existing version to make it a permanent change.

```
my_data$birth_years <- as.numeric(my_data$birth_years)
```

This command tells R to:

1. go and get the variable `birth_years` inside the data frame `my_data`
2. convert it numeric
3. take the numerical output and assign it over the variable `birth_years` in the data frame `my_data`

Now `birth_years` should be numerical.

```
class(my_data$birth_years)
#> [1] "numeric"
```

Now, we can create the current age variable.

```r
# Calculate the current age
2023 - my_data$birth_years
#> [1] 61 43 48 59 67 57 72 70


# It is working. Let's assign this output to a new variable
my_data$age_current <- 2023 - my_data$birth_years

# Check my_data
my_data
#>        names_pm birth_years age_current
#> 1  Keir Starmer        1962          61
#> 2   Rishi Sunak        1980          43
#> 3     Liz Truss        1975          48
#> 4 Boris Johnson        1964          59
#> 5   Theresa May        1956          67
#> 6 David Cameron        1966          57
#> 7  Gordon Brown        1951          72
#> 8    Tony Blair        1953          70
```

2.8 Categorical data

I would like to add a variable showing the party of each Prime Minister. I can create a vector and add this as a column in `my_data`.

For example, Rishi Sunak is from Conservative Party, Liz Truss is also Conservative. Kier Starmer, Gordon Brown and Tony Blair are Labour.

We need a vector where the first item is Labour, followed by five Conservative, and two Labour at the end.

We could write it one by one in order. It would be long and cumbersome, but it would do the job.

```r
parties_long_version <-  c("Labour", # first item Labour
                           "Conservative", # followed by five Conservative (1)
                           "Conservative", # (2)
                           "Conservative", # (3)
                           "Conservative", # (4)
                           "Conservative", # (5)
                           "Labour", # This corresponds to Gordon Brown
```

```
                                  "Labour" # Finally, Tony Blair
                                  )
```

Let's check the object we created.

```
parties_long_version
#> [1] "Labour"       "Conservative" "Conservative" "Conservative" "Conservative"
#> [6] "Conservative" "Labour"       "Labour"
```

This looks good. I could assign it into a new column in `my_data`. But we are learning, so let's try another and faster-to-write way.

Instead of repeating Conservative five times, I could use the repeat function: `rep()`.

```
# rep() repeats an input n times
rep("Conservative", 5)
#> [1] "Conservative" "Conservative" "Conservative" "Conservative" "Conservative"
```

Using this approach, I can build the vector again. I need one "Labour", five "Conservative", and two "Labour", in this order. I need to combine them using `(c)`.

```
parties_short_version <-  c("Labour",
                            rep("Conservative", 5),
                            rep("Labour", 2))
```

Let's check the object we created

```
parties_short_version
#> [1] "Labour"       "Conservative" "Conservative" "Conservative" "Conservative"
#> [6] "Conservative" "Labour"       "Labour"
```

Both versions should be the same, meaning `parties_long_version` should be equal to `parties_short_version`.

```
parties_long_version == parties_short_version
#> [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Now, let's put it into my data frame.

```
my_data$party <- parties_short_version
```

Let's check the data frame.

```
my_data
#>        names_pm birth_years age_current        party
#> 1  Keir Starmer        1962          61       Labour
#> 2   Rishi Sunak        1980          43 Conservative
#> 3     Liz Truss        1975          48 Conservative
#> 4 Boris Johnson        1964          59 Conservative
#> 5   Theresa May        1956          67 Conservative
#> 6 David Cameron        1966          57 Conservative
#> 7  Gordon Brown        1951          72       Labour
#> 8    Tony Blair        1953          70       Labour
```

2.9 Counting frequencies using table ()

Let's see how many individuals from each party is in my data frame. You could count it one by one in a small data set such as this one. I can see that there are 5 Conservatives and three Labour, but imagine that it was a large dataset where counting manually was not an option.

We can use `table()` function to achieve this.

```
table(my_data$party)
#>
#> Conservative       Labour
#>            5            3
```

2.10 Saving data

In the final step, we will learn how to save a data frame such as `my_data` for future use. We have a few options:

1. Write my_data into a spreadsheet-like file.
2. Save the whole R environment with all the objects inside.

We will cover option #1 here.

You have probably used Microsoft Excel (or Google Sheets) to work on spreadsheets before. There are different spreadsheet file types (such as Excels `.xlsx`), but the most common and

compatible one is `.csv`, which stands for comma separated values. This is basically plain text that any computer and most electronic devices can open.

```
write.csv(x = my_data, file = "my_first_file.csv", row.names = F)
```

This should create a file somewhere in your computer, more precisely, in your **working directory**. Let's see where it has saved the file by looking at the working directory.

```
getwd()
```

2.11 Working directory

`getwd()` means get working directory. Working directory is your default file path. This is where R looks for files and saves any output.

Working directory can be different in each computer. R Studio has nice tools for navigation.

You can directly go to your working directory through Files tab (usually in right bottom corner) and using the More drop-down menu. Under there there are a few options:

- Set as working directory: sets your working directory as the current directory shown in Files
- Go to working directory: takes you to current working directory

If you click on go to working directory, you should see **my__first__file.csv** here.

It is a good idea to create a new directory (folder for Windows) for this module. Your folder names should be simple and easy to write. For example: *research_methods* is a good name.

Try not to use space in file names. Underscore or dash are better alternatives. Also, I encourage always using lowercase for file names, which also goes for object names in R.

You can use your operating system to create this directory. You could also use R Studio's Files tab. Put this folder somewhere easy to access.

Let's save your R script. You can use drop-down menu: `File >> Save` OR simply by using the keyboard shortcut `Ctrl-S` (linux), `Command-S` (mac), `Control-S` (windows).

Give an intuitive name to your script. For example, *learn_02.R* is a good name.

It is generally good idea to keep your data in a sub-directory named data. Create such a directory and move `my_first_file.csv` there.

Next week, we will continue with a simple dataset, which will be available on Blackboard.

3 Data Basics

If you have followed the instructions in Chapter 2.10, you should have a folder for this module in which one R file named `01_R_basics.R` and one folder `named` data are present.

You can open R Studio by launching `01_R_basics.R`. You should see a screen similar to Figure 3.1 R Studio will automatically set working directory to your module folder. This approach allows us to have a tidy filing system. Our data files are in the `data` folder.

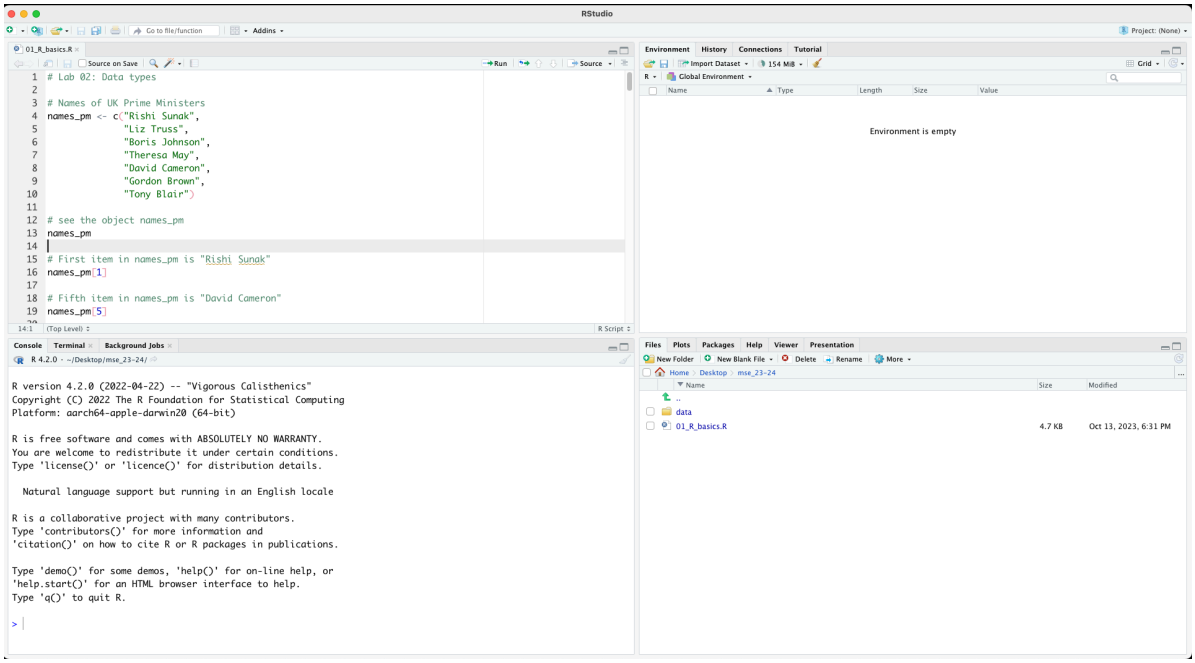


Figure 3.1: R Studio screen

Let's create a new R Script for this week. It is either File > New File > R Script or `Shift-Ctrl-N` (linux), `Shift-Command-N` (mac), `Shift-Ctrl-N` (windows). You can save your script by `Ctrl-S` (linux), `Command-S` (mac), `Ctrl-S` (windows). Give it an intuitive name such as `02_analysis_basics.R`.

3.1 Load a .csv file

For practice purposes, let's load the simple dataframe we created last week. We will use `read.csv()` function. Recall that the file is under the folder `data`.

Let's download the dataset names `World in 2010`. This dataset is available on Blackboard. It can also be downloaded from here. A codebook is also available on Blackboard. Make sure to take a look at it

Let's be tidy and move this file into the data folder under our module folder. You can navigate in your module directory within R Studio. For example, the files tab in Figure 3.2 show the folder data.

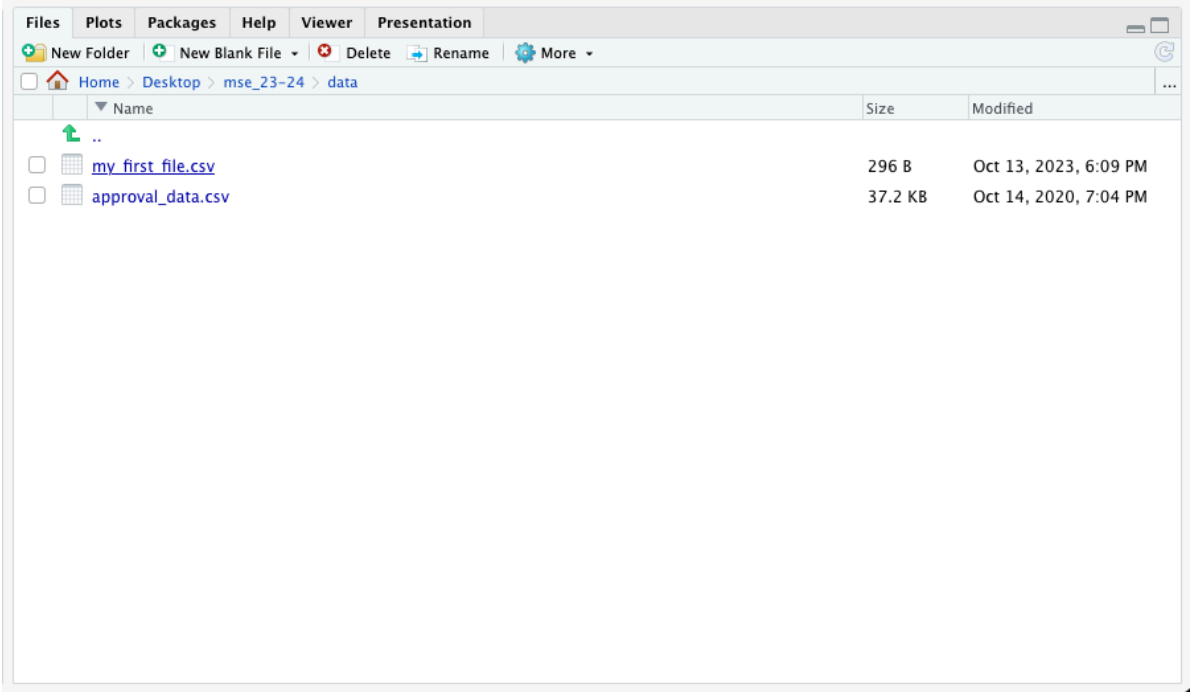


Figure 3.2: R Studio screen

You can actually list files in your computer with R commands too!

```
# list files under working directory
list.files()
```

```
# list files under data folder
list.files("data/")
```

Now we are ready to load `world_in_2010.csv` into R. We should try to understand the structure of the dataset. Let's have a good sense of this dataset.

```
# load data:
df <- read.csv("data/world_in_2010.csv")

# have a look using View(df)
# View(df)

# type of the object
class(df)
#> [1] "data.frame"
#> [1] "data.frame"

# how many variables?
ncol(df)
#> [1] 42
#> [1] 42

# how many rows?
nrow(df)
#> [1] 166
#> [1] 166

# names of the variables
names(df)
#>  [1] "COWcode"                                "Country_Code"
#>  [3] "Country_Name"                           "WB_Region"
#>  [5] "WB_IncomeGroup"                         "Population_total"
#>  [7] "Urban_pop"                              "GDP_pc_PPP"
#>  [9] "Infant_Mortality_Rate"                  "Life_exp_female"
#> [11] "Life_exp_male"                          "HIV"
#> [13] "Literacy_rate_female"                   "Literacy_rate_all"
#> [15] "Current_acc_bal_USD"                    "Current_acc_bal_perc_of_GDP"
#> [17] "ODA_USD"                                "ODA_perc_of_GNI"
#> [19] "Natural_resources_rents_perc_of_GDP"    "FDI_net_inflows_perc_of_GDP"
#> [21] "Net_migration_2008_2012"                "GINI_index_WB_estimate"
#> [23] "Inc_share_by_highest_10per"             "Unemployment_rate"
#> [25] "Surface_area_sq_km"                     "v2x_polyarchy"
#> [27] "democracy"                              "v2x_libdem"
#> [29] "v2x_egaldem"                            "Geographical_Region"
#> [31] "UN_vote_PctAgreeUS"                     "UN_vote_PctAgreeRUSSIA"
#> [33] "UN_vote_PctAgreeBrazil"                 "UN_vote_PctAgreeChina"
```

```
#> [35] "UN_vote_PctAgreeIndia"              "UN_vote_PctAgreeIsrael"
#> [37] "milex"                              "milper"
#> [39] "cinc"                               "CivilConflict"
#> [41] "Corruptions_Perspectives_Index"     "Turnout"
```

Each row represents a state in the international system. So we can say that the unit of observation is the state. Variables show several attributes of each state (note that I will use state and country interchangeable).

We call this a cross-sectional data because we have units but no time dimension. The whole dataset is for the year 2010. Usually, such country-level data would also contain multiple years so that comparison can be over-time, but to keep things simple for now, we are only working with a single year. When we add the time dimension to cross-sectional data, we will call it time-series cross-sectional.

How many countries do we have in this dataset? We already know the answer because we checked the number of rows. Recall that each row represents a state in the international system so that the total number of rows will give me the number of countries in the data.

```
# how many rows?
nrow(df)
#> [1] 166
```

3.2 Summary of a categorical variable

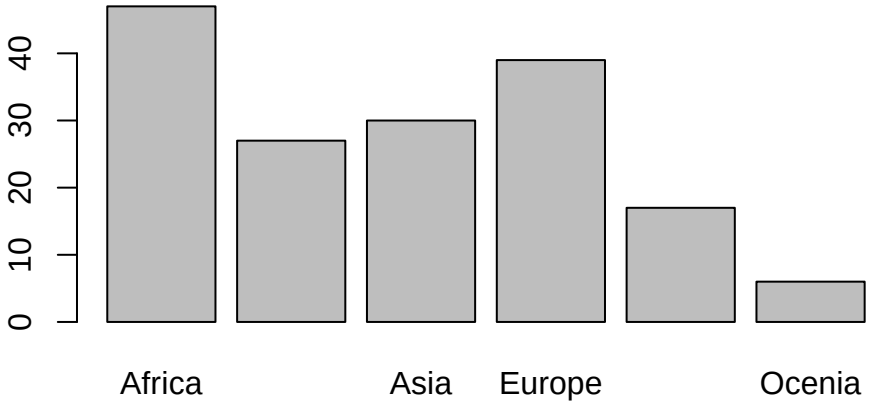
Let's describe a categorical variable. We can work with `Geographical_Region`, which records where a country is located geographically. To see how many countries are in each region, we can use `table()` and provide a descriptive summary of this variable.

```
table(df$Geographical_Region)
#>
#>                   Africa                 Americas                     Asia
#>                       47                       27                       30
#>                   Europe Mid. East & North Africa                   Ocenia
#>                       39                       17                        6
```

Now, we have a frequency table of `Geographical_Region`. I can see that 47 countriese are in Africa, 27 in Americas and so on. For categorical variables, a frequency table is appropriate for a descriptive summary.

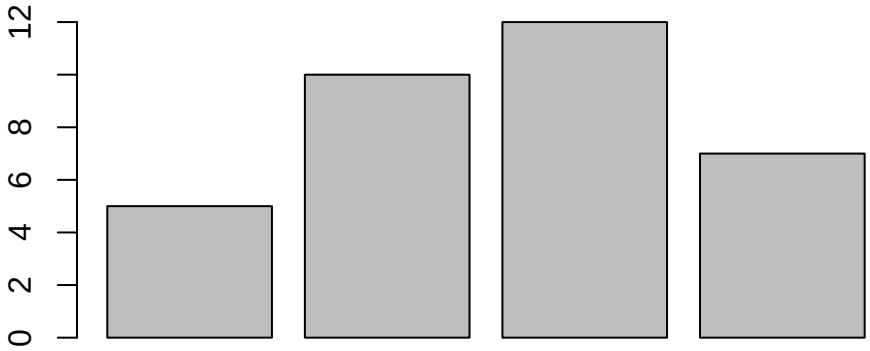
We can also create a bar plot and visually summarize the data. Each bar will represent the number of countries in each geographical region. In short, we will visually display the information in `table(df$Geographical_Region)` using a barplot.

```
barplot(height = table(df$Geographical_Region))
```



This is a good start, but we will do better. Before going any further, let's unpack the code. The function `barplot()` takes a vector of numbers, which it uses to display heights. For instance, if we want to display four bars with heights 5, 10, 12 and 7, we can plug such a vector into barplot().

```
barplot(height = c(5,10,12,7))
```



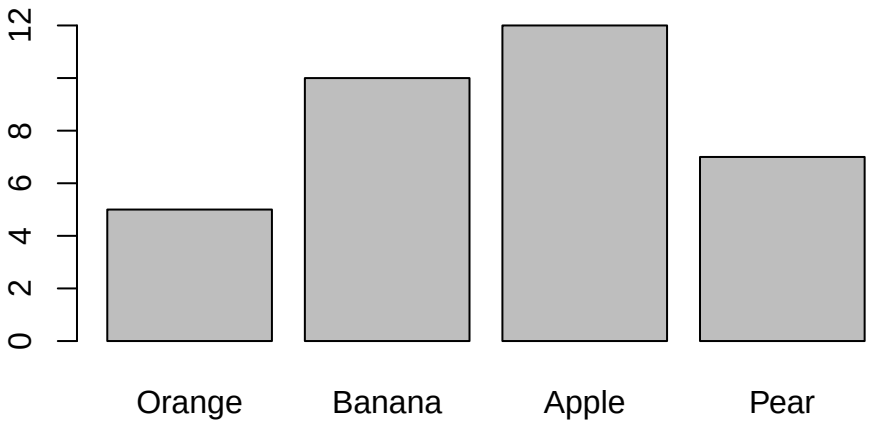
Note that bars above don't have labels at the moment, because we did not provide any information. For Geographical_Region, however, `table()` creates a named vector.

```
# df$Geographical_Region is a named vector:
table(df$Geographical_Region)
#>
#>                  Africa                Americas                    Asia
```

```
#>                       47                       27                       30
#>                   Europe Mid. East & North Africa                   Ocenia
#>                       39                       17                        6

# Names:
names(table(df$Geographical_Region))
#> [1] "Africa"                   "Americas"
#> [3] "Asia"                     "Europe"
#> [5] "Mid. East & North Africa" "Ocenia"
```

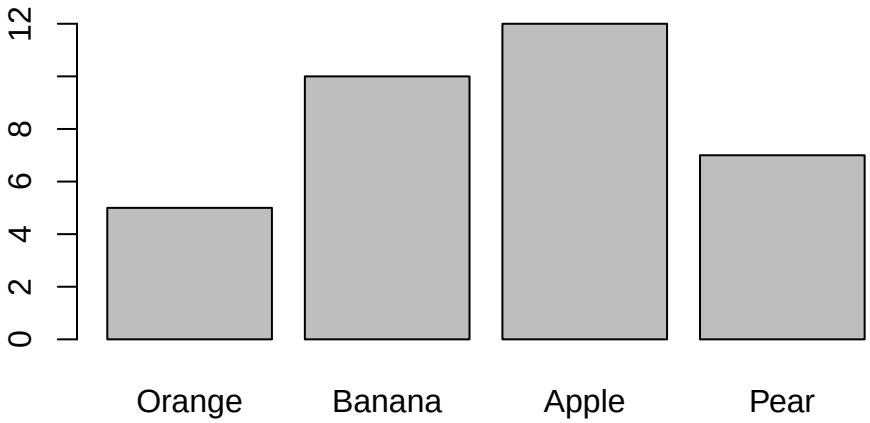
Going back to our hypothetical barplot, we can provide it a names argument. Let's say the numbers 5, 10, 12 and 7 correspond to Orange, Banana, Apple, Pear.

```
barplot(height = c(5,10,12,7), names.arg = c("Orange", "Banana", "Apple", "Pear"))
```



For easier read and navigation, it is a good idea to use a systematic way to write the code. You can press (linux), (mac), (windows) after the end of the first argument.

```
# easier to read code
barplot(height = c(5,10,12,7),
        names.arg = c("Orange", "Banana", "Apple", "Pear")
        )
```



Now we have a very good idea how `barplot()` works.

Going back to the bar plot for `Geographical_Region`, the category "Mid. East & North Africa" is too long. I want to change it with just "MENA".

You can take assign table input to an object and work on it.

```
# Get the table() into an object
georeg_tab <- table(df$Geographical_Region)

# Check the object
georeg_tab
#>
#>                   Africa                 Americas                     Asia
#>                       47                       27                       30
#>                   Europe Mid. East & North Africa                   Ocenia
#>                       39                       17                        6

# check the names of each value
names(georeg_tab)
#> [1] "Africa"                   "Americas"
#> [3] "Asia"                     "Europe"
#> [5] "Mid. East & North Africa" "Ocenia"
```

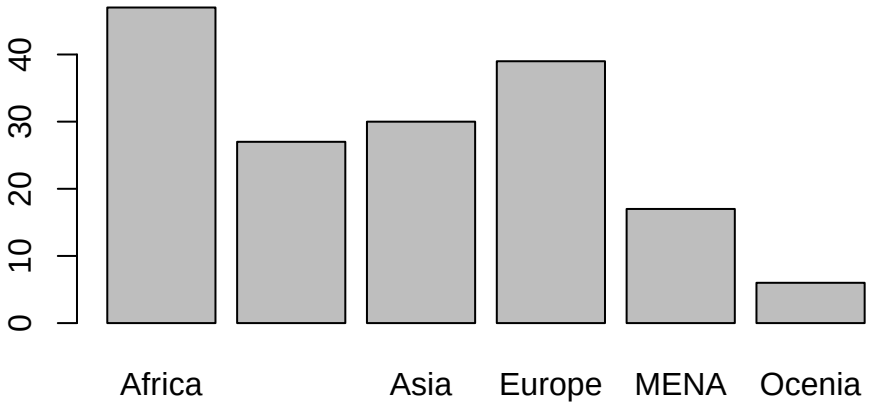
Fifth item in `names(georeg_tab)` is "Mid. East & North Africa". I could go and change it by using the squared bracket `[]` notation.

```
# Fifth Item in Barplot
names(georeg_tab)[5]
#> [1] "Mid. East & North Africa"
```

```r
# To change it you can assign a new name
names(georeg_tab)[5] <- "MENA"

# Check the object again
georeg_tab
#>   Africa Americas     Asia   Europe     MENA   Ocenia
#>       47       27       30       39       17        6
```

Now we are ready to do the barplot.

```r
barplot(georeg_tab)
```



3.3 Understanding functions

We used the command `barplot()` to create a bar graph. `barplot()` is called a function. A function usually gets an object (or objects), and returns something, such as a graph.

You can use the `help()` function to read more about a specific function. For example, let's read the help file for `barplot()`.

```r
help(barplot)
```

An input of a function is called an argument. We don't need to specify all arguments, but some of them are always required. For example, for `barplot()`, we need to define the height.

```r
barplot(height = c(5, 10, 15))
```

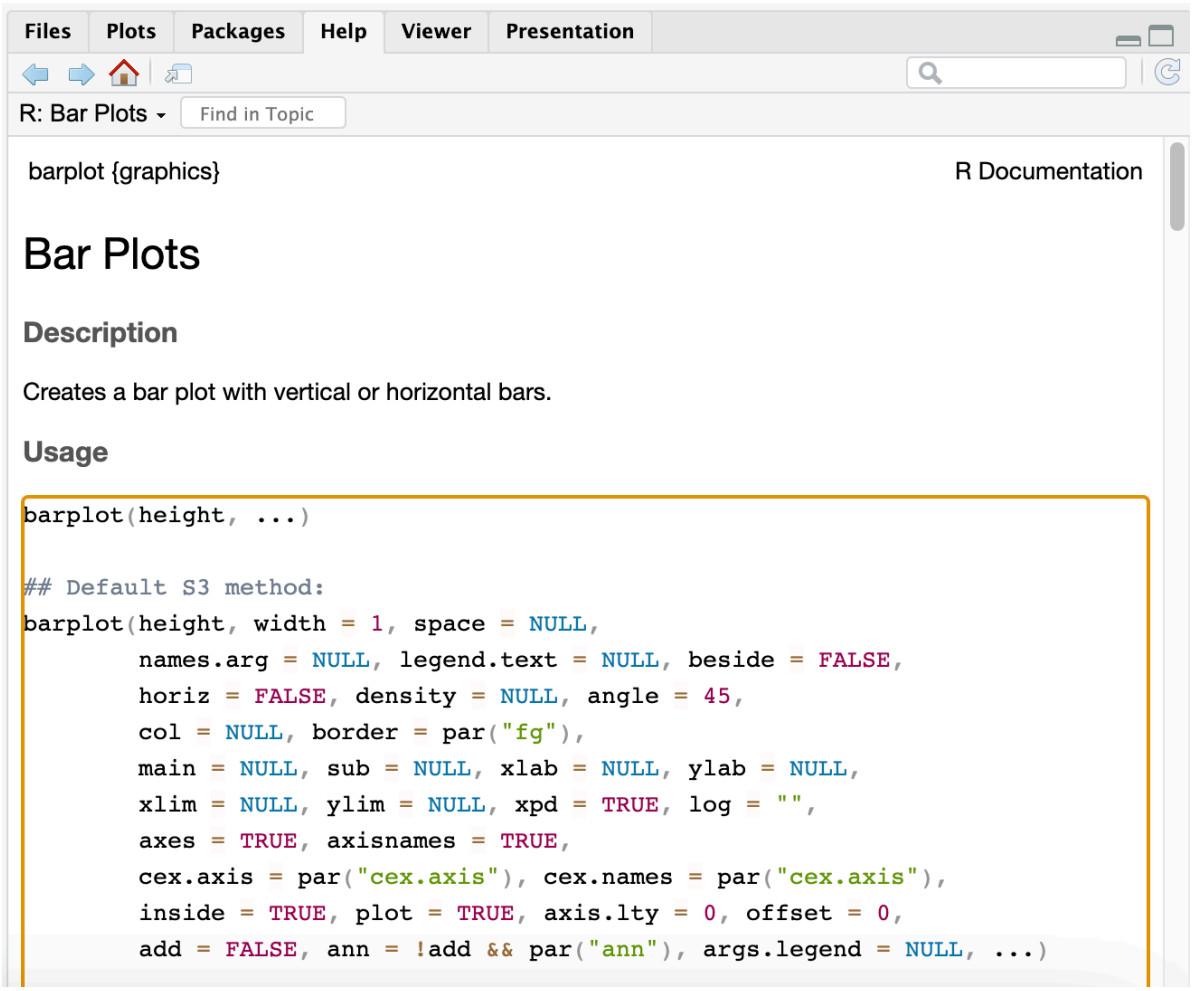
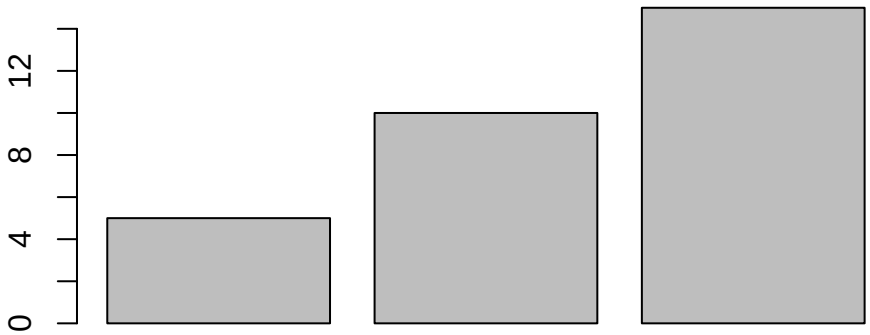
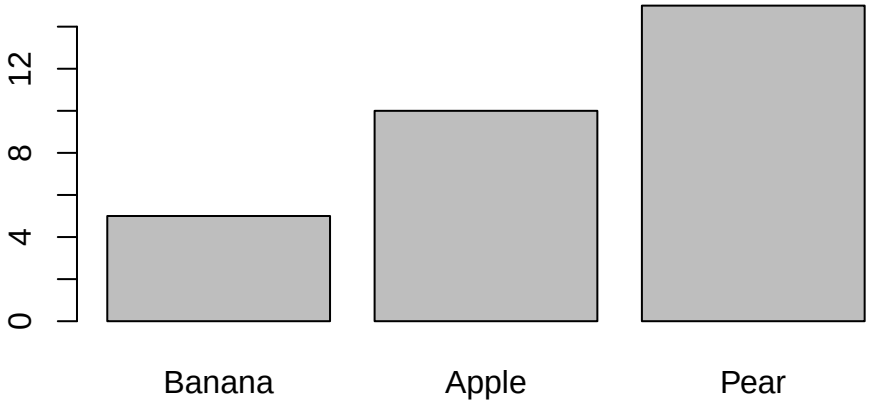


Figure 3.3: Help tab explaining barplot()



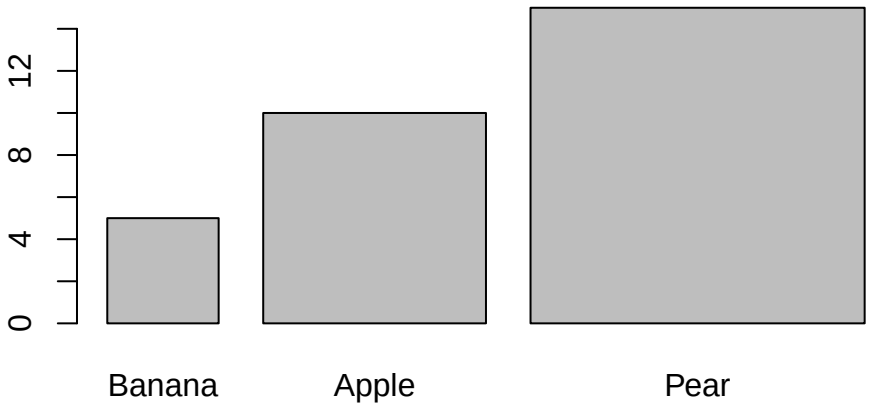
We can add names by adding `names.arg`.

```
barplot(height = c(5, 10, 15),
        names.arg = c("Banana", "Apple", "Pear"))
```



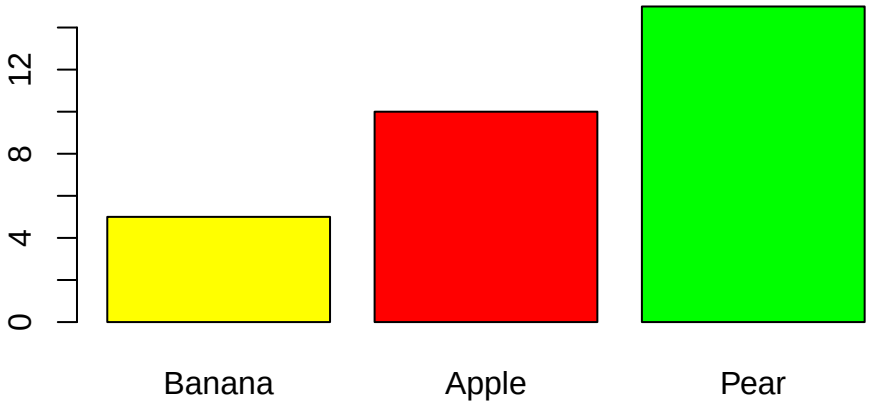
You can change the width of each bar.

```
barplot(height = c(5, 10, 15),
        names.arg = c("Banana", "Apple", "Pear"),
        width = c(1,2,3))
```

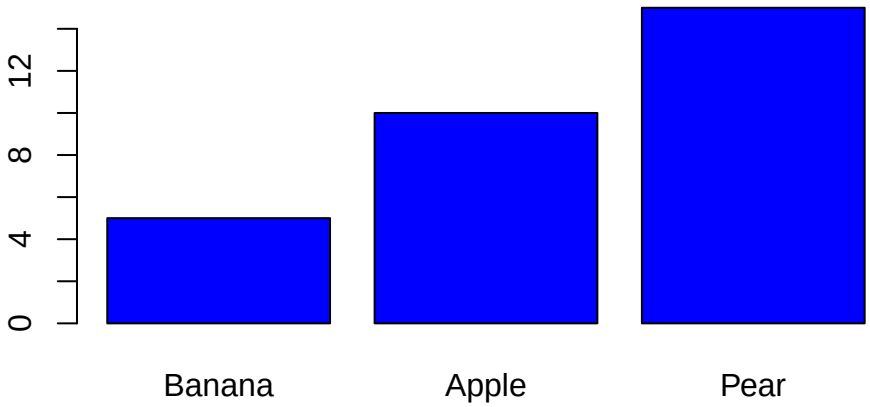


It is also possible to change colors (for all three, or one by one).

```
# yellow, red and green colors
barplot(height = c(5, 10, 15),
        names.arg = c("Banana", "Apple", "Pear"),
        col = c("yellow", "red", "green"))
```




```
# all blue
barplot(height = c(5, 10, 15),
        names.arg = c("Banana", "Apple", "Pear"),
        col = "blue")
```



3.4 Summary of a numerical variable

Next, let's work with a numerical variable. `Life_exp_female` is a measurement of female life expectancy across the world in 2010. The function `summary()` will give the min, max, mean, median, and first and third quartiles.

```
# Summary of the variable
summary(df$Life_exp_female)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   48.88   64.41   74.91   71.94   78.73   86.30
```

We can also check such statistics one by one.

```
# mean of a variable
mean(df$Life_exp_female)
#> [1] 71.94012

# minimum
min(df$Life_exp_female)
#> [1] 48.88

# maximum
max(df$Life_exp_female)
#> [1] 86.3
```

Recall that quartiles divides the data into four parts. Median is the mid-point, which is also called the second quartile.

Table 3.1: Five point data summary

Symbol	Name(s)	Definition	Use
min	Minimum	Minimum of data	The lowest data point
Q_1	25^{th} Percentile First Quartile	Splits off the lowest 25% of data from the highest 75%	A common low value
Q_2	50^{th} Percentile Second Quartile Median	Middle of data	A common value
Q_3	75^{th} Percentile Third Quartile	splits off the highest 75% of data from the lowest quarter	A common high value
max	Maximum	Maximum of data	The highest data point

Besides `summary()` we can also use `median()` and `quantile()` functions to get the quartiles.

```
# median:
median(df$Life_exp_female)
#> [1] 74.905

# also median:
quantile(df$Life_exp_female, probs = 0.50) # 0.50 indicates half-way (50%)
#>    50%
#> 74.905

# first quartile
quantile(df$Life_exp_female, probs = 0.25) # 0.25 indicates 25%
#>   25%
#> 64.41

# third quartile
quantile(df$Life_exp_female, probs = 0.75) # 0.75 indicates 75%
#>   75%
#> 78.73

# five point summary
quantile(df$Life_exp_female, probs = c(0, 0.25, 0.50, 0.75, 1))
#>     0%    25%    50%    75%   100%
#> 48.880 64.410 74.905 78.730 86.300
```

Using `summary()` to get a five-point numerical summary is perfectly fine. We briefly visited

`quantile()` for demonstration purposes. There are many ways to achieve the same thing in R.

For dispersion, variance and standard deviation can be calculated.

```
# standard deviation
sd(df$Life_exp_female)
#> [1] 9.353926

# variance
var(df$Life_exp_female)
#> [1] 87.49594

# recall that square root of variance is standard deviation
sqrt( var(df$Life_exp_female) )
#> [1] 9.353926

# ask R if you don't believe me
sd(df$Life_exp_female) == sqrt( var(df$Life_exp_female) )
#> [1] TRUE
```

Note that `==` here means *is equal to?*

```
# 2 plus 2 is 4, or is it?
2 + 2 == 4
#> [1] TRUE

# is 10 / 2 equal to 4?
10 / 2 == 4
#> [1] FALSE
# no
```

3.5 Visual summary

When working on a new dataset, it is always a good idea to start by visually summarizing your variables of interest one-by-one. This will help you to get a better understanding of the data.

For a numerical variable, two types of graphs are appropriate for a visual summary:

- Histogram
- Box plot

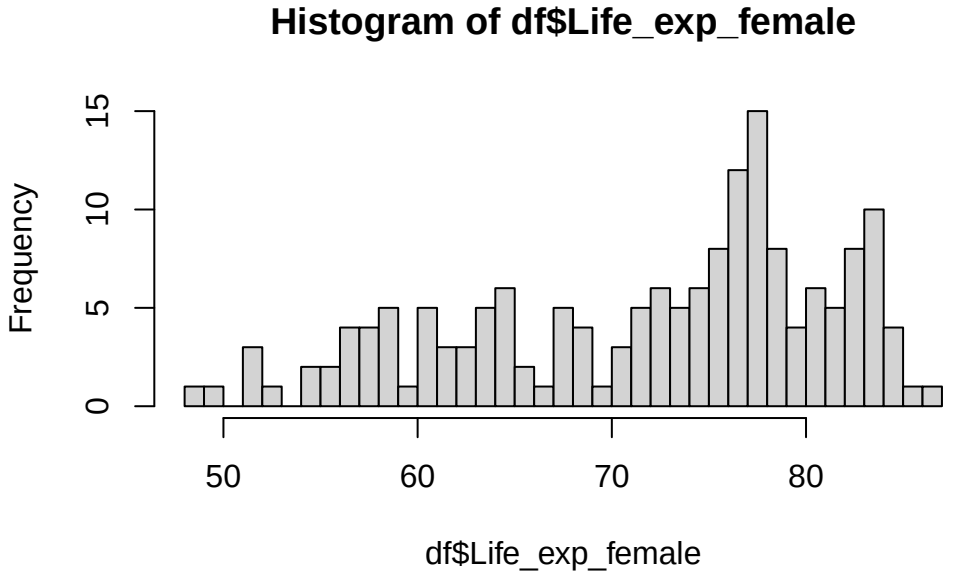
3.5.1 Histogram

```
# histogram:
hist(df$Life_exp_female)
```

You can tell R how many bins you would like to have in your histogram by using the `breaks` argument, but as the R help file clarifies, this number is a suggestion only, and R can use a different (but similar value) to draw a pretty histogram.

```
# histogram with fewer bins
hist(df$Life_exp_female, breaks = 5)
```


```
# histogram with higher number of bins
hist(df$Life_exp_female, breaks = 30)
```



You might want to specify cut points. For example, you might want to have a sequence from 40 to 100 with increment of 5.

```
# a sequence from 40 to 100 by 5
seq(from = 40, to = 100, by = 5)
#>  [1]  40  45  50  55  60  65  70  75  80  85  90  95 100
```

```r
# put it into an object
my_breaks <- seq(from = 40, to = 100, by = 5)
# this will be our break points

#tell R to do the histogram using these points
hist(df$Life_exp_female, breaks = my_breaks)
```

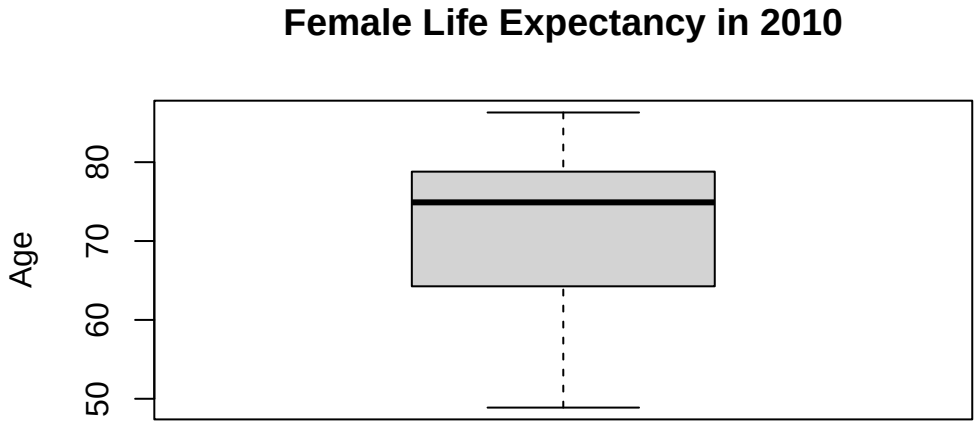
This looks quite nice and intuitive. However, titles are not defined. I don't want to see `df$Life_exp_female` as an axis or main graph title.

```r
#Histogram with titles and breaks
hist(df$Life_exp_female,
     breaks = my_breaks,
     main = "Female Life Expectancy in 2010",
     ylab = "Number of Countries",
     xlab = "Age"
     )
```

3.5.2 Box plots

A box plot is another option for graphically summarising a numerical variable. Box graphs are really nice to get a sense of the data, understand the distribution, and quickly see if there are any outliers. They are also very good at creating visual comparisons across groups, something which we will cover in the upcoming weeks.

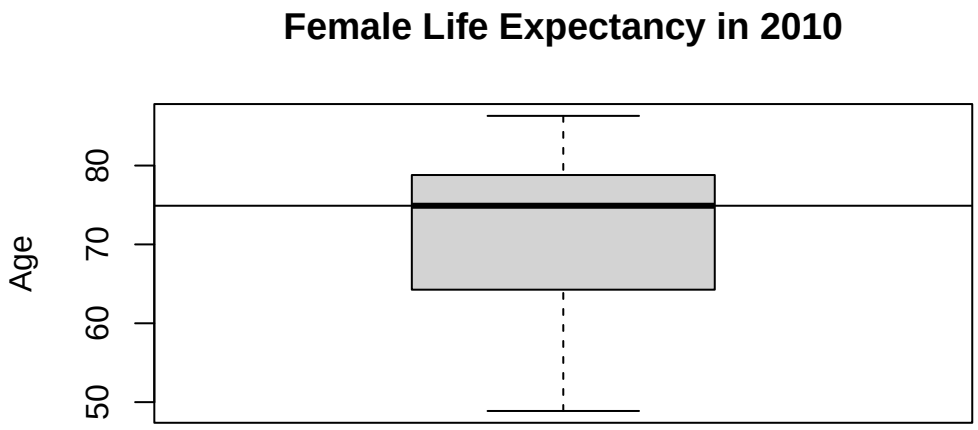
```
# Boxplot
boxplot(df$Life_exp_female,
        main = "Female Life Expectancy in 2010",
        ylab = "Age")
```



This box plot helps us to visualize the five-point summary. You can see minimum, max, median and first and third quartiles.

If you don't believe me, we can plot these over the boxplot.

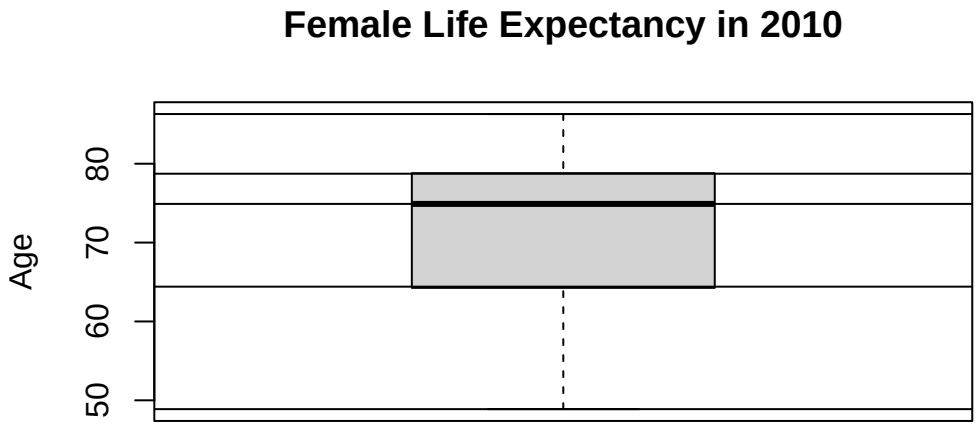
```
# box plot:
boxplot(df$Life_exp_female,
        main = "Female Life Expectancy in 2010",
        ylab = "Age")

# you can add a line to a plot by abline
# h stands for horizontal
# tell R to draw a horizontal line at the median of Life_exp_female
abline(h = median(df$Life_exp_female))
```



You can draw lines for each statistic.

```
# box plot:
boxplot(df$Life_exp_female,
        main = "Female Life Expectancy in 2010",
        ylab = "Age")

# median
abline(h = median(df$Life_exp_female))

# min
abline(h = min(df$Life_exp_female))

# max
abline(h = max(df$Life_exp_female))
```

```r
# first quartile
abline(h = quantile(df$Life_exp_female, probs = 0.25))

# third quartile
abline(h = quantile(df$Life_exp_female, probs = 0.75))
```

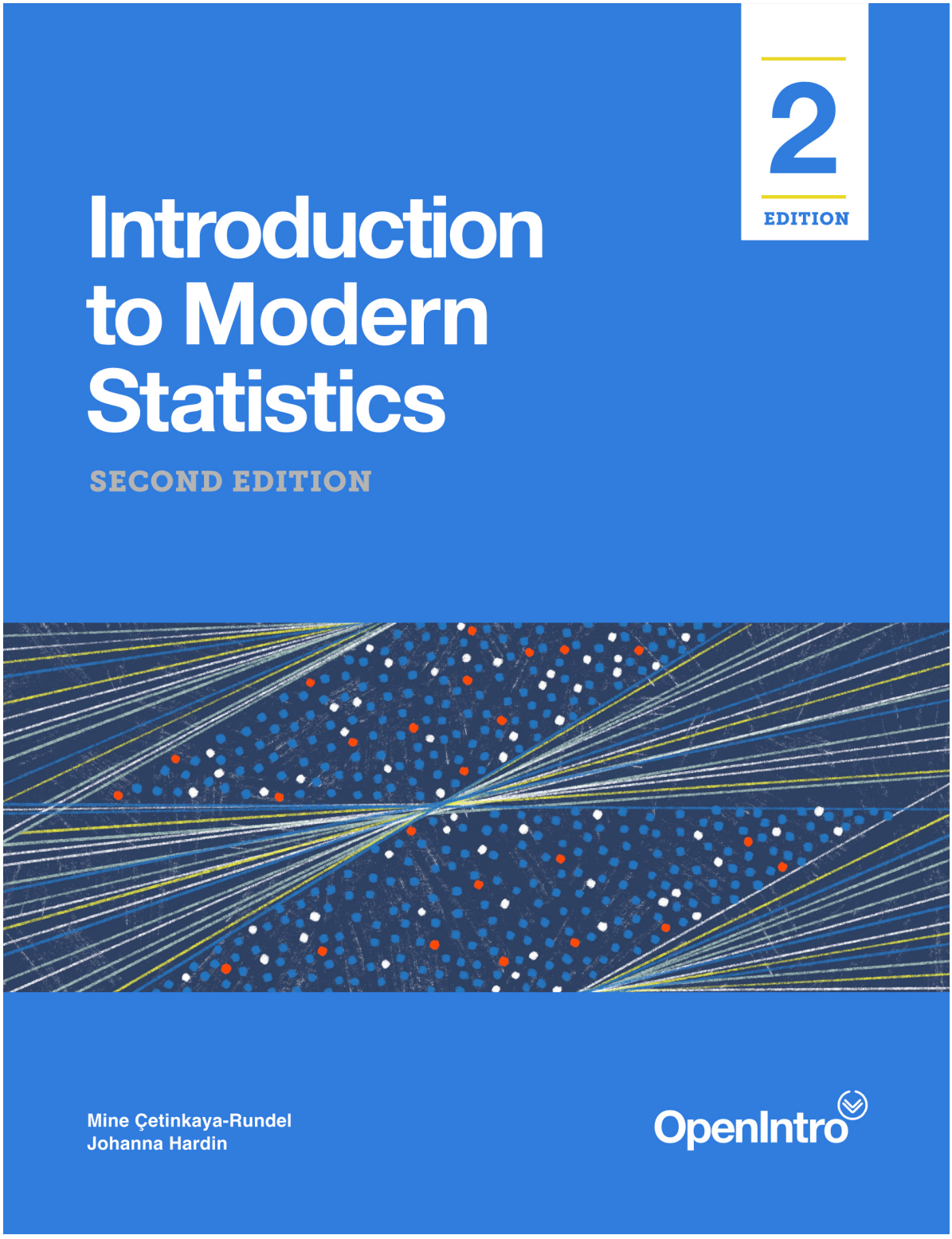


We drew these lines for demonstration purposes only. It is to learn that a box plot visualizes the information in a five-point summary. Normally, we wouldn't show them this way because we can assume the reader knows how to read a box plot.

Resources

Here are some resources that are very useful for learning R and quantitative research methods.

Introduction to Modern Statistics (2e)



Introduction to Modern Statistics by Mine Çetinkaya-Rundel and Johanna Hardin is an excellent textbook for learning foundational concepts in statistics and data analysis while learning R.

The online textbook is free and available at https://openintro-ims.netlify.app/.

Also see http://openintro.org/book/ims for supplementary materials and additional resources.

Hands-On Programming with R

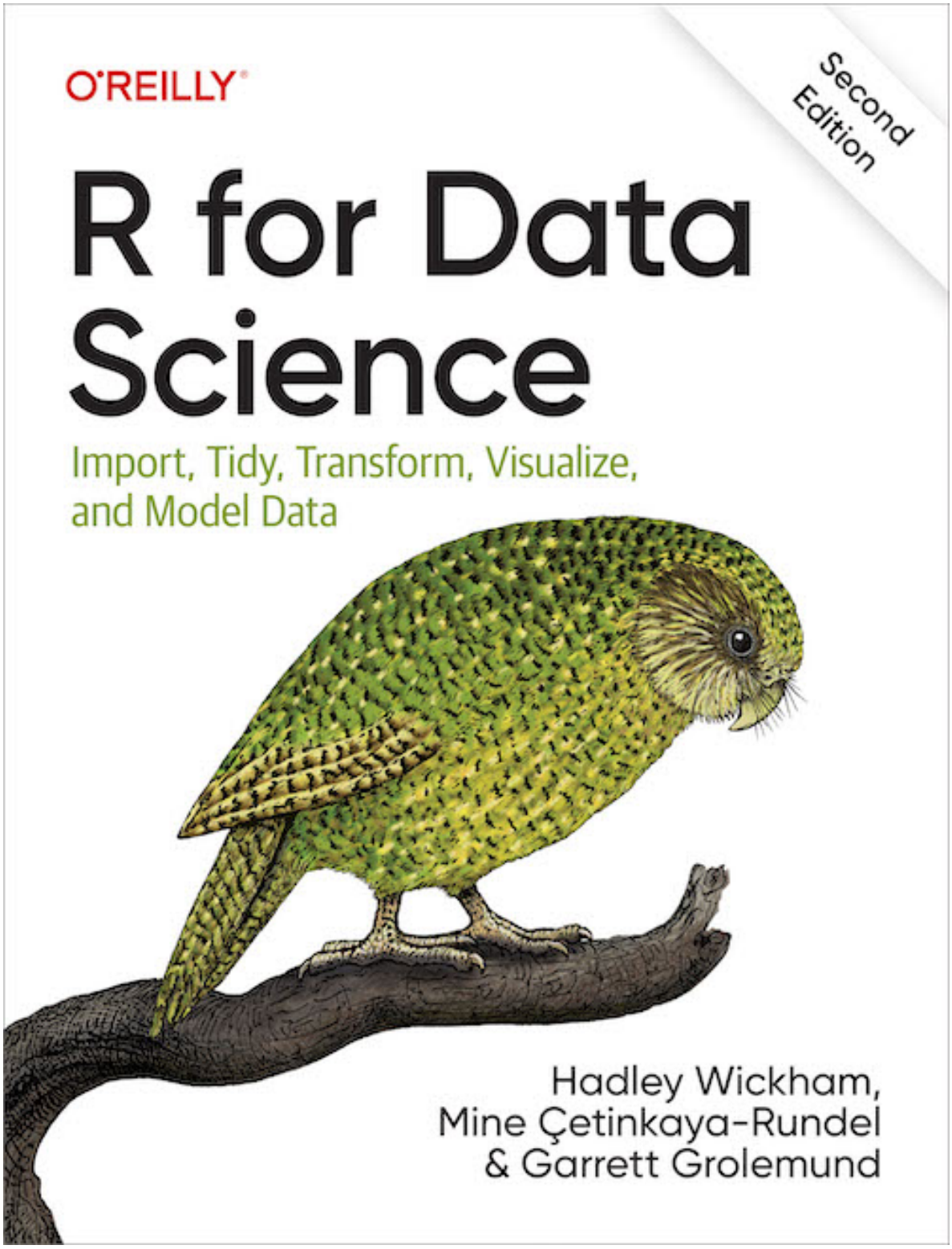
Hands-on Programming with R by Garrett Grolemund is a straightforward introduction to R. It is useful to learn the basics of R notation.

We cover most of the content in Part 1 & 2 in the first four weeks, but if you want to approach the same content from a different angle, you will find this textbook useful.

It is freely available here: https://rstudio-education.github.io/hopr/


O'REILLY
Hands-On Programming with R
WRITE YOUR OWN FUNCTIONS AND SIMULATIONS
Garrett Grolemund
Foreword by Hadley Wickham

R for Data Science (2e)



R for Data Science (2e) by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund is an introductory textbook on getting started with R and tidyverse for data management, analysis and visualisation. It is an excellent source to learn the basics of R, R Studio and tidyverse.

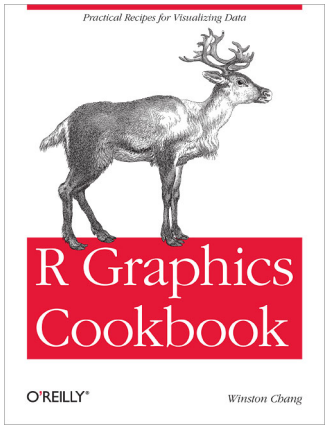
It can be used as a reference textbook, especially when you are struggling to recall the syntax. It has many examples to get a grasp (or remember) how to use many base R and tidyverse functions.

It is free and available here: https://r4ds.hadley.nz/

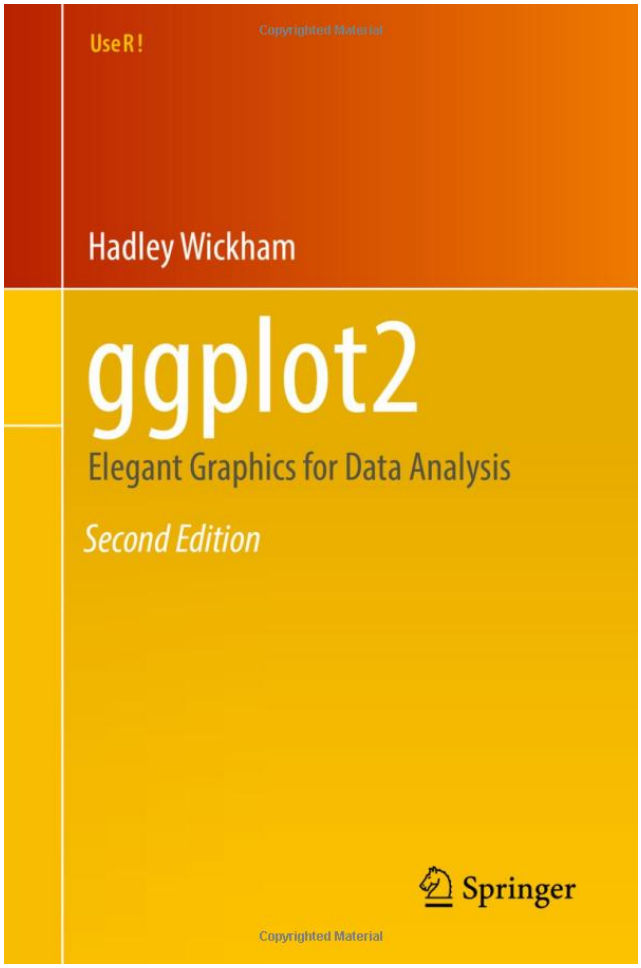
R Graphics Cookbook

R Graphics Cookbook by Winston Chang is a detailed textbook on creating visualisations in R via using ggplot2.

It is free and available here: https://r-graphics.org/



ggplot2: Elegant Graphics for Data Analysis (3e)



This book explains the underlying theory behind ggplot2. It is available here: https://ggplot2-book.org/