Social Science Research Methods

Barış Arı

2024-09-24

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.

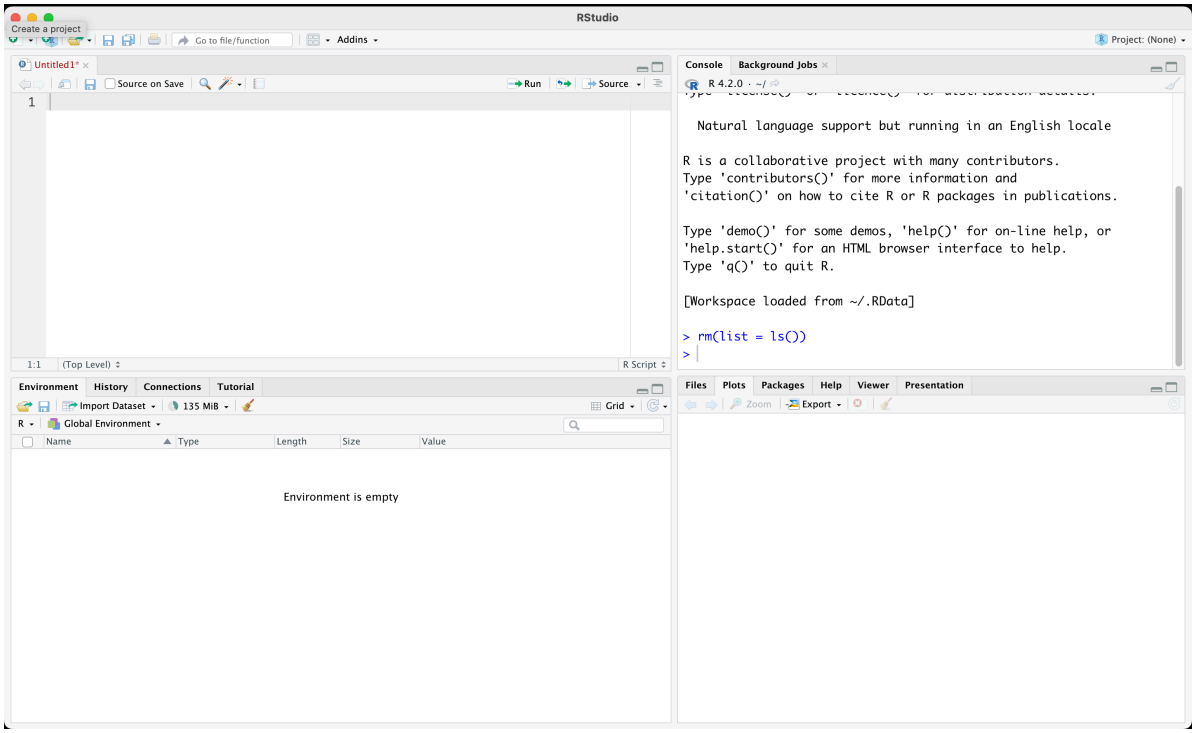


Figure 1.1: R Studio screen with four panes

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 `Control-S` (windows), `Command-S` (mac), `Ctrl-S` (linux). 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*.

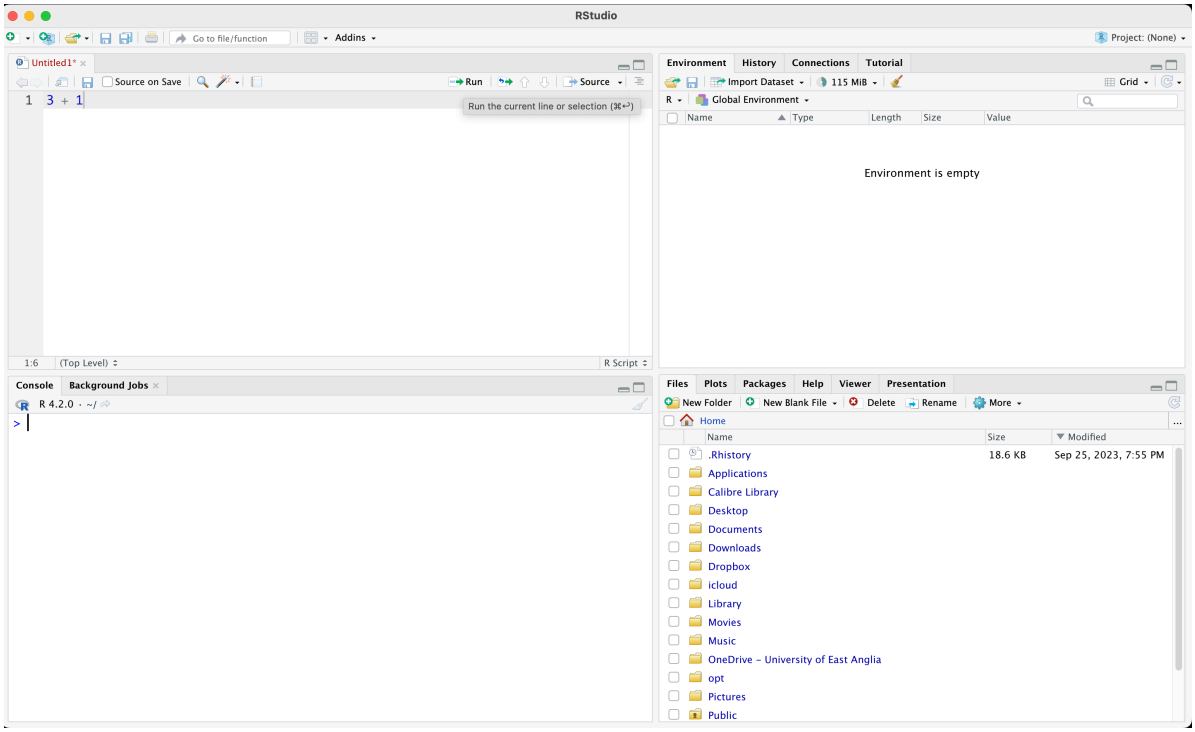


Figure 1.2: Our first calculation

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 in eval(expr, envir, enclos): 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 in eval(expr, envir, enclos): 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)
                 )
```

Class of an object

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
```

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"
```

2.1 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
```

2.2 *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.

2.3 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
```

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
```

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.

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

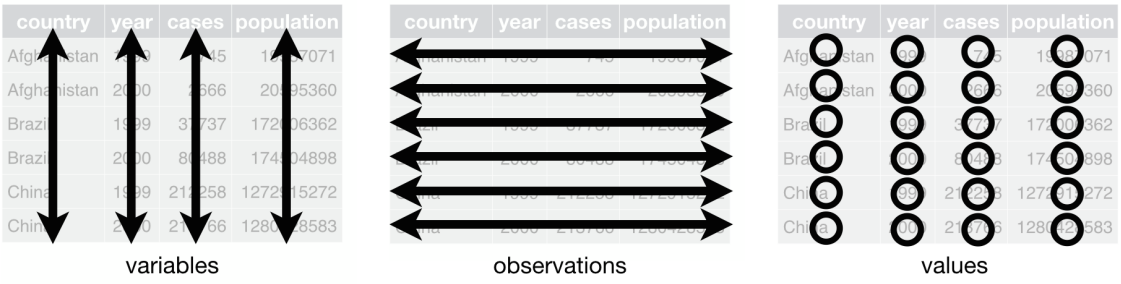


Figure 2.1: Variables, observations, rows. Well-organised data come in this form.

```
# 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"
```

2.4 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 FUN(left, right): 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.

```
# 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
```

2.5 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.

```
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
```

2.6 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
```

2.7 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()
```

2.8 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 `Control-S` (windows), `Command-S` (mac), `Ctrl-S` (linux).

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

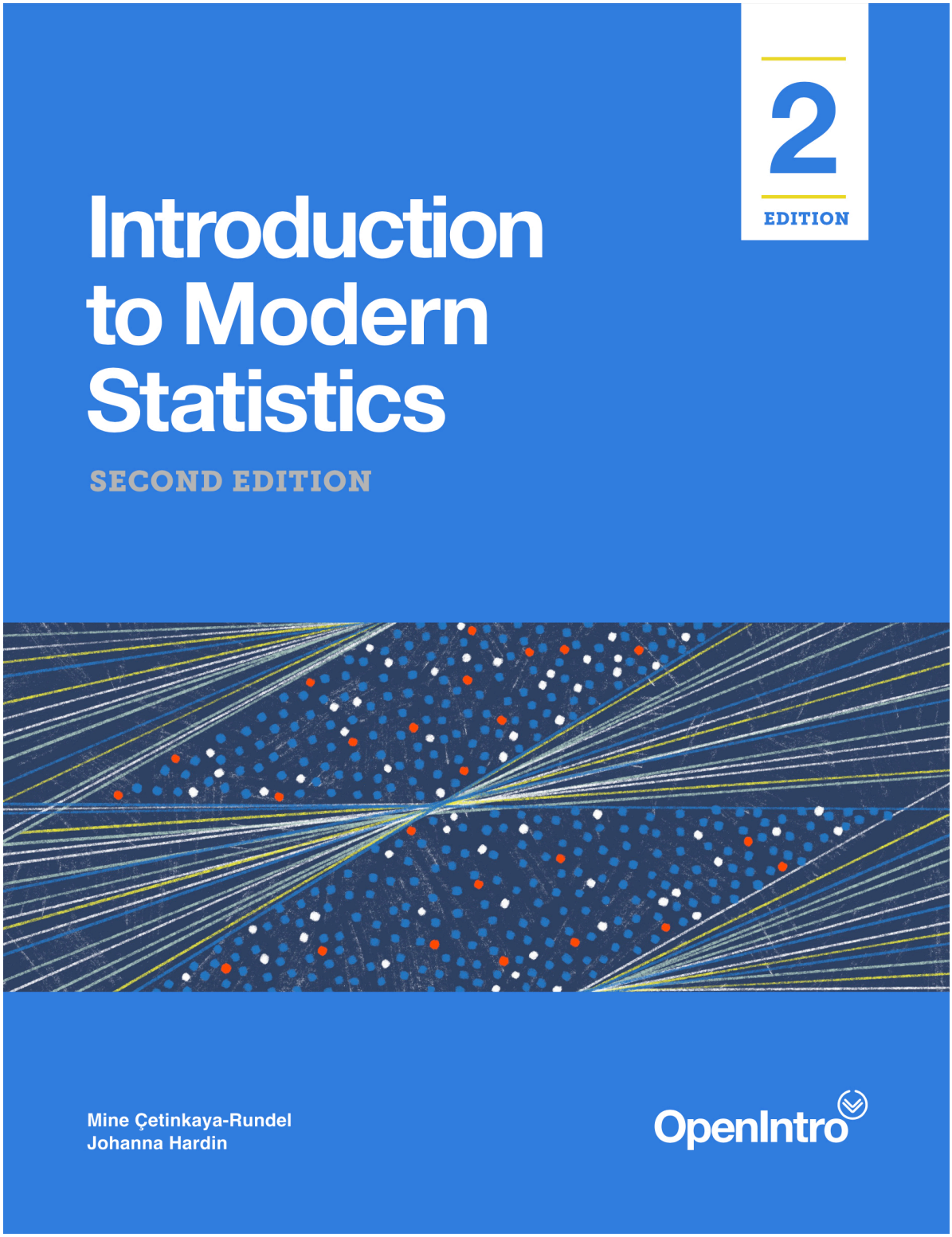
It is generally good idea to keep your data in a sub-directory named data. Create such a directory and move `my_first_file.csv` there.

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

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

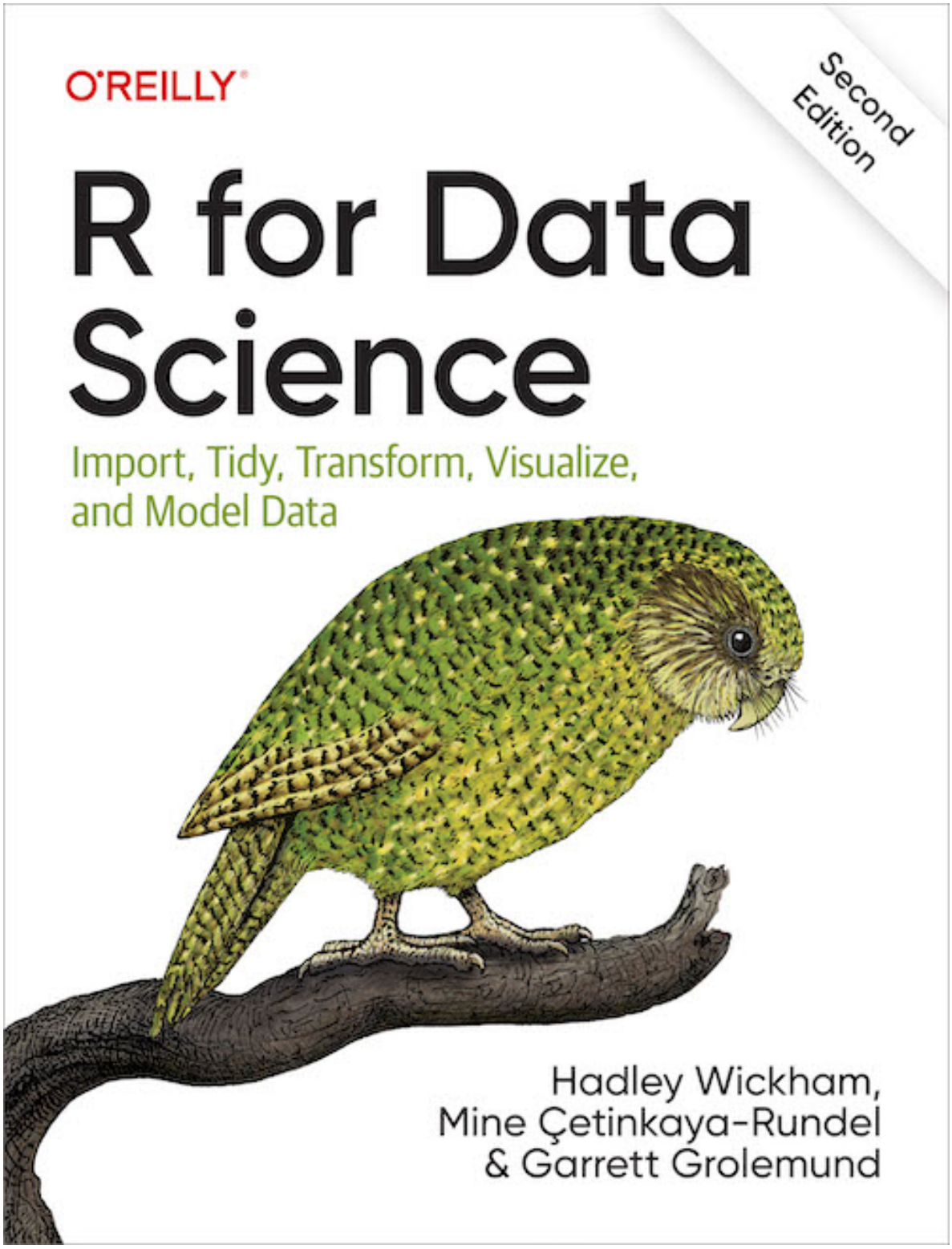
Hands-on Programming with R by Garrett Grolemund is a straightforward introduction to R. It is useful to learn the basics of R notation.

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.

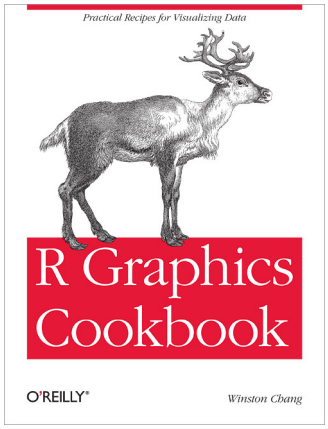
It can be used as a reference textbook, especially when you are struggling to recall the syntax. It has many examples to get a grasp (or remember) how to use many base R and tidyverse functions.

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

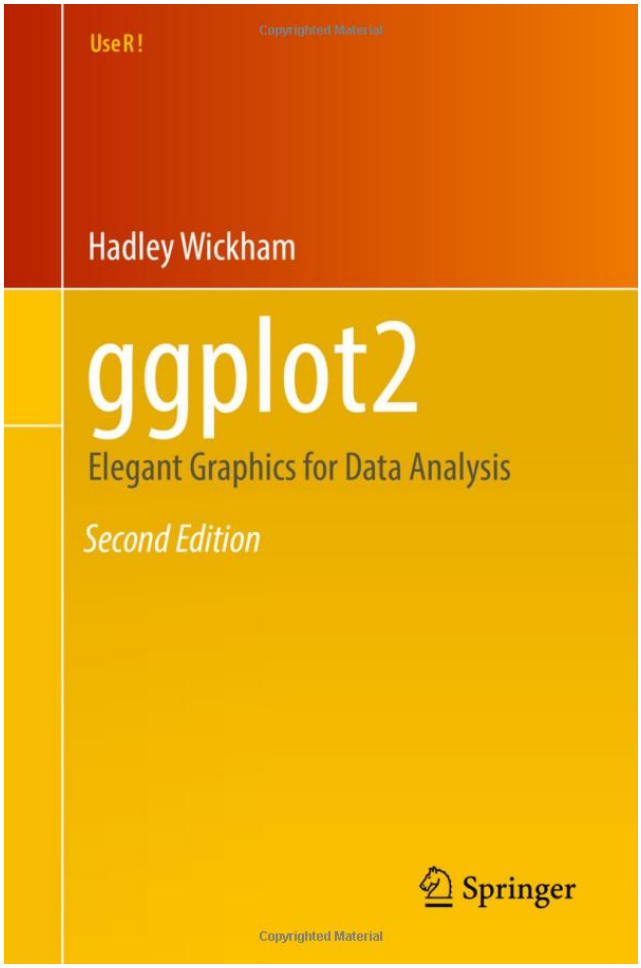
R Graphics Cookbook

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/