Programming and Quantitative Skills for IBA - R

Christoph Walsh

2023-10-01

ii

Table of contents

Chapter 1

About

Welcome to the online "book" for the R part of the first-year IBA course *Programming and Quantitative Skills*. This book accompanies the content covered in the lectures. On Canvas, I will post which chapters/slides we will cover in each lecture, as well as other material.

Chapter 2

Getting Started

In this chapter we will briefly talk about what R and RStudio are, and how to install them on your computer.

2.1   What is R and what is RStudio?

R is a programming language which specializes in statistical computing and graphics.

RStudio, on the other hand, is a desktop application where you can write R code, execute R programs, and view plots created by R.

Thus R is the programming language itself, and RStudio is the desktop application you will use to write and execute R code.

2.1.1   What is a programming language?

Without getting into a complicated details, a programming language is a way to communicate to a computer via written text in a way that the computer can understand so that you can instruct it to do various operations for you. This is very different to how we might be used to interacting with a computer, which often involves pointing and clicking on different buttons and menus with your mouse.

Knowing how to program is a very useful skill because you can automate repetitive tasks that would take you a very long time if you had to them "by hand" (i.e. by clicking things with your mouse). For example, suppose you work in a hotel in a city and you need to check how much your competitors are charging for rooms on different days so that you can adjust prices to stay competitive. Every day you have to go to all the different websites of the competing hotels and take note of the prices in an Excel sheet. With programming, what you

could do instead is write code that tells the computer to automatically visit those websites every day, record the hotel room prices, and put them in an Excel sheet for you. This is a process called *web scraping*. This is just one example of the many ways programming languages can automate repetitive tasks.

When humans speak to each other and someone makes a grammar mistake, it usually isn't a big deal. We usually know what they mean. But if you make a "syntax" mistake in a programming language, it won't understand what you mean. The computer will either throw an error or, worse still, do something you didn't want it to do. Therefore we need to be very careful when writing in a programming language.

2.1.2 Why learn R?

There are many different programming languages out there, and each have their different strengths and weaknesses. R is specialized in working with datasets, performing statistical analyses and visualizing data. These will be very useful for later courses in the IBA program, and for many different jobs you might have in the future after your studies.

R has many advantages over alternatives:

- R is free, open source and runs on all common operating systems. This means you can share your code with anyone and they will be able to run it, no matter what computer they are on or where they are in the world.
- There is a very large active community that creates packages to do a wide-range of operations, keeping R up to date with the latest developments in Statistics and other fields. Excellent community help is also available at Stackoverflow.
- The R language is arguably easier to learn than some similar alternatives. Programming languages also have a lot in common, so if you learn one it's much easier to learn another one. With some R knowledge, it makes learning other languages, such as Python or Julia, much easier.
- You can code in R using RStudio which is a great free integrated desktop environment to write and run R code.
- R is also extremely versatile in what it can do. For example, this online "book" and the accompanying slides were made entirely in RStudio! R can also make very nice data visualizations, such as this or this.

In the second year of the IBA program you will take Statistics 2. There we will learn how to estimate statistical models using R. You can therefore also think of this course as a foundation providing the background programming knowledge for that course. Thus your journey with R won't end after you take the exam for this course. You will use it again and again throughout your studies!

More recently, employers are increasingly looking for people with programming skills. Knowledge of R is therefore a great addition to your CV when you look for a job after your studies!

2.2 Installing R and RStudio

To get started, we need to install both R and RStudio. Install R first, then RStudio:

- To download and install R, go here.
- To download and install RStudio, go here.

If you are on a university computer, R and RStudio will already be installed.

Below are OS-specific instructions.

2.2.1 Installation on Windows

- Go here, click on "Download R for Windows" and then click on the link for "base". Then click on "Download R-X.Y.Z for Windows", where X.Y.Z is the latest version number. Click on the downloaded `.exe` file and follow the installation steps.
- Go here and download the `.exe` file listed next to the Windows OS. Click on the downloaded `.exe` file and follow the installation steps.

On Canvas you will find a video demonstration of me installing both R and RStudio on Windows.

2.2.2 Installation on MacOS

- Go here, click on "Download R for macOS". If you have a newer Mac with an M1 or M2 processor, download the `.pkg` file with "arm64" in the name. For older Macs, download the "x86_64" one. Click on the file and follow the installation steps.
- Go here and download the `.dmg` file listed next to macOS. Click on the downloaded `.dmg` file and copy it to your applications.

If you have difficulties, there are many videos on YouTube demonstrating the installation of R and RStudio on a Mac.

2.2.3 Installation on Ubuntu/Debian

- To install R, run the following command in the command line: `sudo apt install r-base`.
- To install RStudio, first download the `.deb` file from the RStudio website. Then in terminal, change to the directory containing the file and install it with `sudo dpkg -i rstudio-*.deb`.

2.3 Opening RStudio

After the installation, try opening RStudio on your computer. If prompted to choose a version of R, just choose the default option. After opening RStudio

you should see something like:
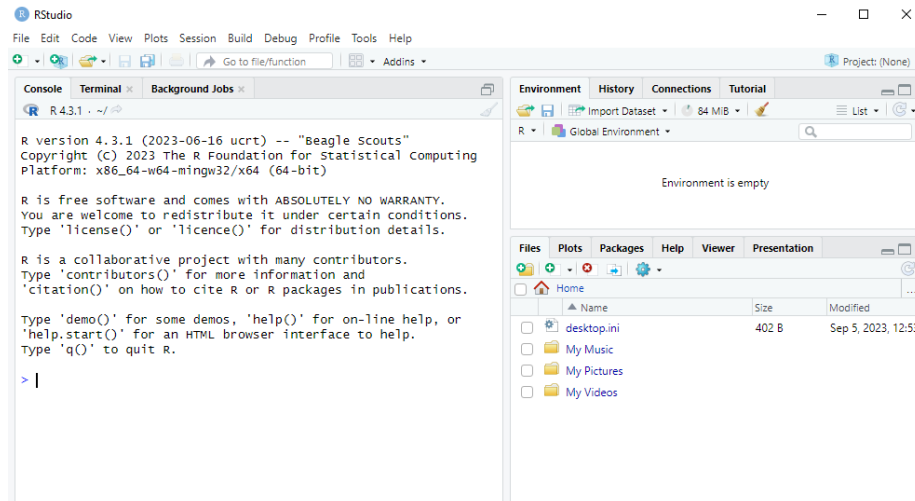
What we will do in the next chapter is learn some basic commands in R.

Chapter 3

R as a Calculator

In this chapter we will learn how to use R as a calculator.
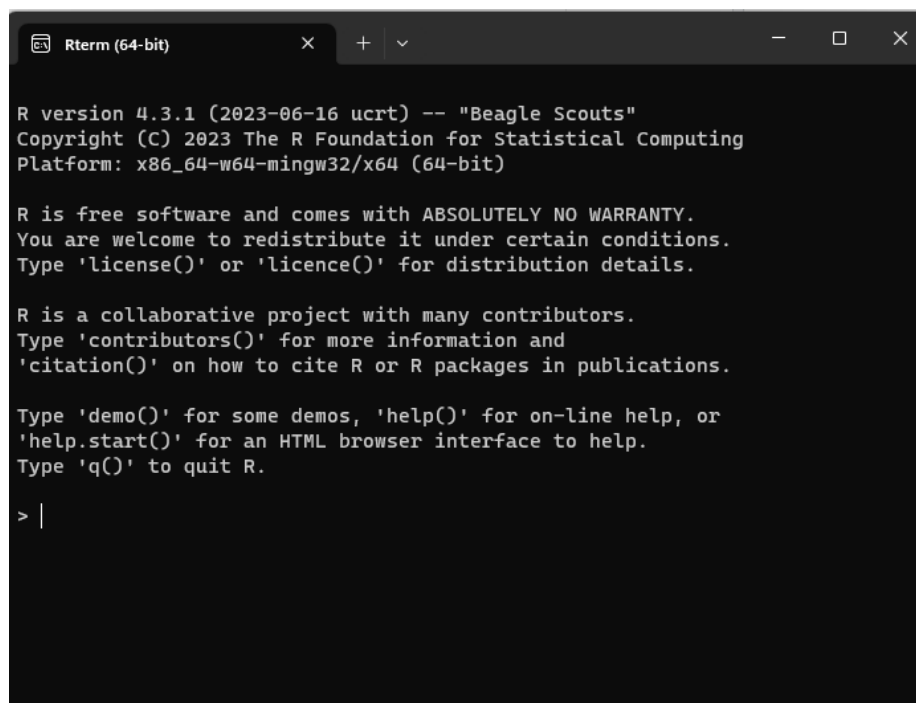
3.1   The R Console

The R console is where you provide commands in the R programming language to be executed by the computer. It is possible to access the R console without RStudio using a very basic command-line interface that looks like this:[1]

However, RStudio also has this R console, as well as many other useful features. Therefore we will stick to using R within RStudio for the rest of this course.

Let's learn how to do a very basic calculation in R. If we want to calculate $2 + 3$, we simply go to the *Console* tab in RStudio and type `2+3`, just like in the screenshot below:

---

[1] If the date is `01/01/69`, the format `%d/%m/%y` will interpret it as January 1 **1969**. But if the date is `01/01/68`, it will interpret it as January 1 **2068**. All short-format years after 69 are put in the 1900s and all short-format years before 69 are put in the 2000s. You don't need to remember these details for the exam though because we won't ever use dates outside of 1969-2068.

Figure 3.1: R Terminal

We then press `Enter` to see the answer. What we will see then is R giving the following output:
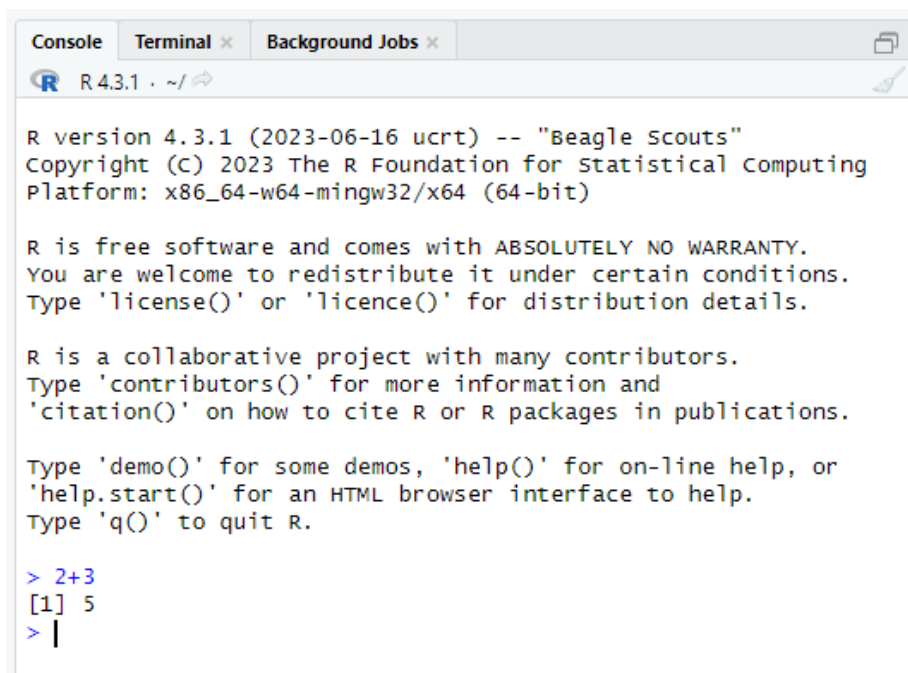
The `[1]` before the `5` here essentially means the `5` is the first number in the output. This is obvious here, but this feature will be more useful later when we do operations on more numbers. But for now, we can just ignore the `[1]`.

In this book, I won't always show screenshots like this. Instead I will show code

snippets in boxes like this:

```
2 + 3
```

`[1] 5`

The part that is code will be in color and the output will be in a separate
gray box below it. In these code boxes there is a small clipboard icon on the
right which you can use to copy the code to be able to experiment with it in
RStudio yourself. This will be useful when we start writing longer commands
and programs.

We will now go through some different operations. We will also learn about
functions and their *arguments* along the way, which we will be be using again
and again throughout the rest of this course.

3.2   Addition, Subtraction, Multiplication and Division

Addition, subtraction, multiplication and division are given by the standard `+`,
`-`, `*` and `/` operators that you would use in other programs like Excel, or even
in an internet search engine. For example:

```
2 + 3
```

`[1] 5`

```
5 - 3
```

`[1] 2`

```
2 * 3
```

`[1] 6`

```
3 / 2
```

`[1] 1.5`

It is also possible to do multiple operations at the same time using parentheses.
For example, suppose we wanted to calculate:

$$\frac{2+4}{4 \times 2} = \frac{6}{8} = 0.75$$

We can calculate this in R as follows:

```
(2 + 4) / (4 * 2)
```

```
[1] 0.75
```

3.3 Troubleshooting: "Escaping" in R

Suppose by accident you left out the closing parentheses and you see the following:

```
> (2 + 4) / (4 * 2
+
```

R didn't run the command, but it also didn't give an Error. What happened is that **Enter** moved to a new line instead of executing the command. Essentially R is waiting for you to finish the command by closing the parentheses. If you want to "Escape" such a situation, you just need to press the **Esc** button. In general, if anything strange happens in R and you get stuck, you can always press the **Esc** button to return to "normal".

3.4 Exponentiation (Taking Powers of Numbers)

In mathematics notation x^n means we multiply x by itself n times. For example, $2^3 = 2 \times 2 \times 2 = 8$. In R we use the ^ operator to do this:

```
2^3
```

```
[1] 8
```

3.5 Absolute value

Taking the absolute value turns a negative number into the same number without a minus sign. It has no effect on positive numbers.

In mathematics notation we write $|x|$ for the absolute value of x. The formal definition is:

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

Here are some examples:

- $|-2| = 2$
- $|3| = 3$.

This is what the function looks like when we plot it for different x:

```r
if (!require(ggplot2)) install.packages("ggplot2")
library(ggplot2)
x <- seq(-2, 2, length.out = 1000)
ggplot(data.frame(x, y = abs(x)), aes(x, y)) +
  geom_line() +
  ylab("abs(x)")
```
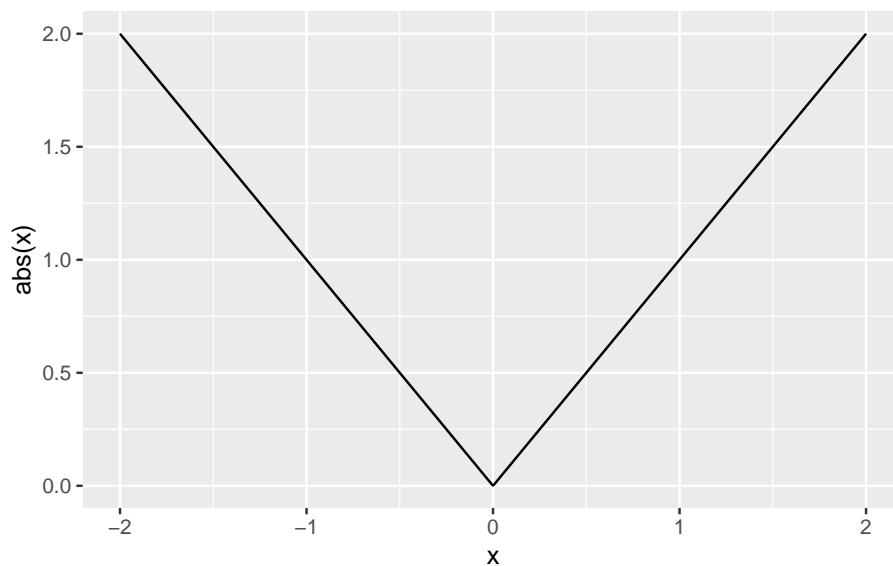

We'll learn how to make plots like this later in Chapter 16.

In R we can calculate these with:

```r
abs(-2)
```

```
[1] 2
```

```r
abs(3)
```

```
[1] 3
```

Taking the absolute value in R involves using what is called a *function*. Functions are used by calling their names and giving the *arguments* to the function in parentheses. When we do `abs(-2)`, `abs` is the name of the function and `-2` is the argument.

In many ways the functions in R work a lot like the functions in Excel, just they might have different names or be used a bit differently. For example, in Excel you write `=ABS(-2)` to take the absolute value of -2. The argument is the same, and the function name only differs in that in Excel you need to use capital letters whereas in R you use lowercase letters (in addition, Excel requires you to put an `=` before the function name).

When using functions it is helpful to read their help pages. You can look at this by typing `help(abs)` or `?abs` in the Console and pressing `Enter`. The help page then pops up in the *Help* tab, like in the screenshot below:
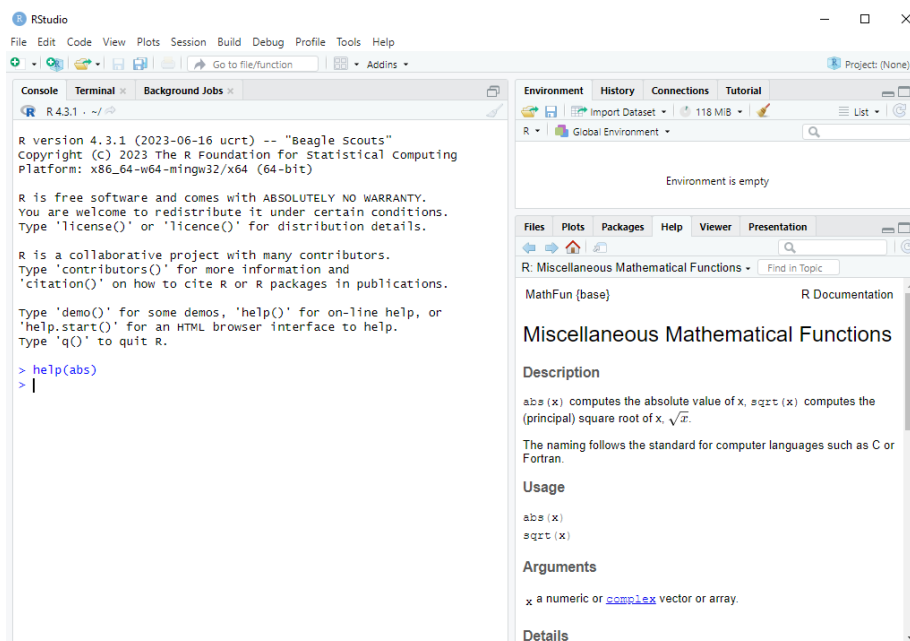
Figure 3.2: Help Page for `abs`

We can see that it says `abs(x) computes the absolute value of x`. So we are told that `x` is the argument.

We will be using many different functions and it's a good habit of to look at their help pages. The help pages will also be available to you in the exam.

3.6 Square and Cubed Roots

The square root of a number x is the y that solves $y^2 = x$. For example, if $x = 4$, both $y = -2$ and $y = 2$ solve this. The principal square root is the positive y from this.

Here is what the square root function looks like for different x:

```r
if (!require(ggplot2)) install.packages("ggplot2")
library(ggplot2)
x <- seq(0, 10, length.out = 1000)
ggplot(data.frame(x, y = sqrt(x)), aes(x, y)) +
  geom_line() +
  ylab("sqrt(x)")
```
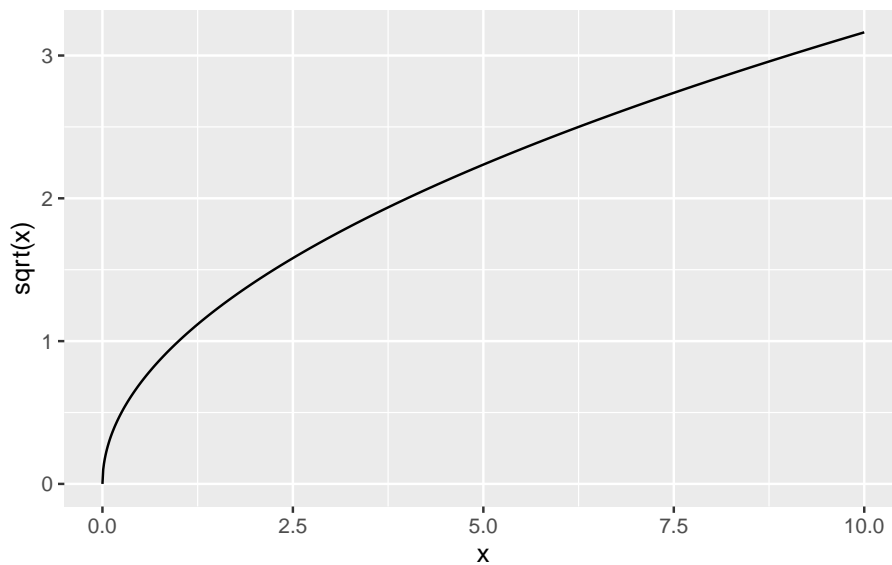

We take the principal square root in R using the `sqrt()` function:

```r
sqrt(9)
```

```
[1] 3
```

It is also possible to take a square root by exponentiating a number by $\frac{1}{2}$:

```r
9^(1/2)
```

```
[1] 3
```

With this approach we can also take the cubed root of a number: $\sqrt[3]{8} = 8^{\frac{1}{3}} = 2$:

```r
8^(1/3)
```

```
[1] 2
```

3.7 Exponentials

A very important function in mathematics and statistics is the exponential function. The definition of $\exp(x)$, or e^x, is given by:

$$e^x = \lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n$$

Note: you don't need to know or remember this definition for the exam. You only need to know how to use the exponential function in R.

This is what the function looks like:

```
if (!require(ggplot2)) install.packages("ggplot2")
library(ggplot2)
x <- seq(-2, 2, length.out = 1000)
ggplot(data.frame(x, y = exp(x)), aes(x, y)) +
  geom_line() +
  ylab("exp(x)")
```
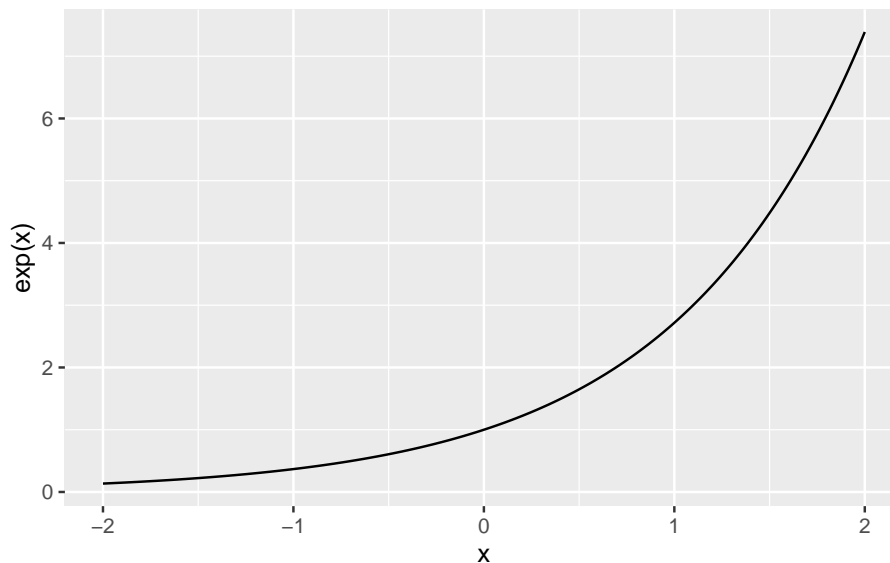

In R we can use the `exp()` function to calculate the exponential of any number:

```
exp(1)
```

```
[1] 2.718282
```

3.8   Logarithms

Another common mathematical function is the logarithm, which is like the reverse of exponentiation.

The log of a number x to a base b, denoted $\log_b(x)$, is the number of times we need to multiply b by itself to get x. For example, $\log_{10}(100) = 2$, because $10 \times 10 = 100$. We need to multiply the base $b = 10$ by itself twice to get to $x = 100$.

A special logarithm is the natural logarithm, $\log_e(x)$, which is the logarithm to the base $\exp(1) = e^1 \approx 2.7183$. This is also written as $\ln(x)$.

This is what the function looks like:

```r
if (!require(ggplot2)) install.packages("ggplot2")
library(ggplot2)
x <- seq(0.01, 10, length.out = 1000)
ggplot(data.frame(x, y = log(x)), aes(x, y)) +
  geom_line() +
  ylab("ln(x)")
```
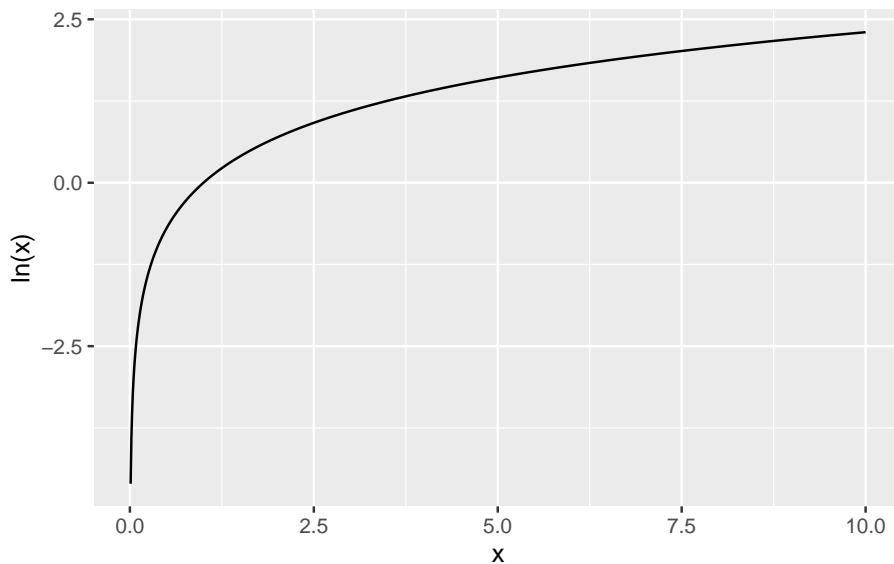

In R we use the `log()` function to calculate the natural logarithm:

```r
log(1)
```

```
[1] 0
```

What if we want to calculate the logarithm to a base other than e? If we look at the help page for `log()` using `help(log)` or `?log` we can see that the `log` function has 2 arguments:

- `x`: the number we want to take the log of.
- `base`: "the base with respect to which the logarithms are computed. Defaults to e=`exp(1)`".

This is the first time that we have seen a function with more than one argument. Earlier when we used the `log()` function to calculate the natural logarithm we only used one argument because we used the *default setting* for the base. But when we want to use a base other than e, we need to specify it.

How we calculate $\log_{10}(100)$ in R is as follows:

```
log(100, base = 10)
```

```
[1] 2
```

We write both arguments into the `log()` function, separated by commas.

This is just like how we used functions with more than one argument in Excel, for example the `VLOOKUP` function. We separated the arguments there by commas as well.

Chapter 4

Objects and Object Types

In this chapter we will learn how to store objects in the R environment and about different *object types* in R.

4.1 The Assignment Operator

4.1.1 Assigning Objects

In the previous chapter when we were using R as a calculator, we simply typed the numbers we wanted to add if we wanted to add them, like `2 + 3`. We can also store numbers in R as *objects*. We do this using the assignment operator `<-`, which is a "less than symbol" and a "minus" symbol next to each other.

For example, let's assign the value 2 to an object called *a* and the value 3 to an object called *b*:

```
a <- 2
b <- 3
```

The `<-` operator assigns the value `2` to `a`, and similarly for `b`.

Variable names can be multiple characters long and can contain dots (`.`), underscores (`_`) and numbers - as long as they don't start with a number. For example, `a1`, `a.1`, and `a_1` are possible variable names, but `1a` is not.

Because we use the assignment operator `<-` so often, RStudio has a shortcut for it. If you hold `Alt` and press `-`, RStudio will write `<-`, including spaces around it. It makes the spaces because `a <- 2` is easier to read than `a<-2`.

4.1.2   The Environment Tab in RStudio

When we do this, we see values $a = 2$ and $b = 3$ in the *Environment* tab in RStudio, just like in the screenshot below:
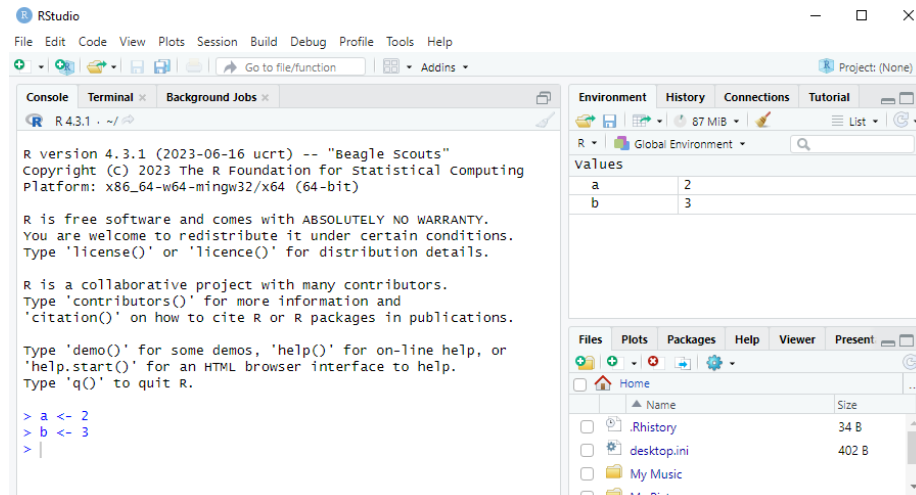
We can now perform all the operations we learned about using a and b instead of the numbers. For example:

```
a + b
```

[1] 5

```
a / b
```

[1] 0.6666667

4.1.3   Troubleshooting with the Assignment Operator

Although you can assign 2 to a with either `a <- 2` or `a<-2`, it is very important that you don't have a space in between the `<` and the `-` in the assignment operator. If you try to instead do `a < - 2`, R will check if `a` is less than -2, instead of assigning 2 to `a`. If `a` is not stored in the Environment you will get an error that says `Error: object 'a' not found`. This is another reason why need to be very careful when typing our code! It's a good idea therefore to use the `Alt+-` shortcut to make `<-`.

It is also possible to use the `=` sign for assignment instead of `<-`. For example, it's possible to do `a = 2` instead. I will use `<-` in this course as it is the recommended approach in R style guides, but you are free to use `=` instead in the exam and assignments if you prefer.

4.2 Common Object Types

We now go through some different object types.

4.2.1 Numeric Vectors

In R we often work with *vectors*, which are collections of values of the same type. You can think of these as a column of data in an Excel file. If we want to store the vector of numbers

$$a = \begin{pmatrix} 1 \\ 3 \\ 7 \\ 2 \end{pmatrix}$$

in R we can use the `c()` function, where "c" stands for *combine*. We put each element of the vector in `c()` separated by commas:

```
a <- c(1, 3, 7, 2)
```

Notice in the Environment tab that now we have overwritten the `a <- 2` that we had before. We can see that `a` is now a `num [1:4]`: it's a numeric vector with 4 elements.[1]

One thing worth mentioning is that when we store single numbers, such as with `b <- 3`, we are actually creating a numeric vector with only 1 element (instead of 4 like in the example above). We could create an identical object with `b <- c(3)` instead.

4.2.2 Logical Vectors

Often we have data on a variable where the answers are "Yes" or "No". We often code these as a logical vector which is binary: the elements are either `TRUE` (corresponding to "Yes") or `FALSE` (corresponding to "No"). For example:

```
a <- c(TRUE, FALSE, TRUE, TRUE)
```

We can see in the Environment tab that `a` is now a `logi [1:4]`: it's a logical vector with 4 elements. It's possible to convert logical vectors into numeric ones with 1s replacing the `TRUE`s and 0s replacing the `FALSE`s. We can do this with the `as.numeric()` function:

```
as.numeric(a)
```

---

[1]If the date is `01/01/69`, the format `%d/%m/%y` will interpret it as January 1 **1969**. But if the date is `01/01/68`, it will interpret it as January 1 **2068**. All short-format years after 69 are put in the 1900s and all short-format years before 69 are put in the 2000s. You don't need to remember these details for the exam though because we won't ever use dates outside of 1969-2068.

```
[1] 1 0 1 1
```

4.2.3 Character Vectors

R can also work with character vectors which are vectors composed of letters or words instead of numbers or logical constants (`TRUE` or `FALSE`). We have to write the words in quotes, otherwise R will think we are providing it with variable names instead:

```r
a <- c("programming", "and", "quantitative", "skills")
```

4.2.4 Factors (Categorical Variables)

Surveys often contain questions with multiple possible responses. For example, imagine a survey which asked people how long it took them to travel to campus and what mode of transportation they used, with the options being:

1. Train
2. Walk
3. Cycle

Suppose we have 6 responses for this survey and we coded the times (in minutes) as a numeric vector `time` and the travel modes as a character vector `travel_mode`:

```r
time <- c(25, 20, 15, 10, 17, 30)
travel_mode <- c("train", "train", "walk", "cycle", "walk", "train")
```

Because categorical variables like `travel_mode` are so common, R has a special object type for them called *factors*. We can turn any vector into a factor using the `factor()` function:

```r
travel_mode <- factor(travel_mode)
travel_mode
```

```
[1] train train walk  cycle walk  train
Levels: cycle train walk
```

We can see in the Environment that we have a factor with 3 levels, `"cycle"`, `"train"`, `"walk"`. The *levels* are all of the different categories.

Having a variable in this format will be very useful when we learn how to visualize data. They will also become very useful when we estimate statistical models with categorical data in Statistics 2 next year.

4.2.5 Data Frame

An object that we will use very frequently is the `data.frame`. This is a rectangular object with different columns representing different variables and rows representing different observations. For example, we could collect the 6 survey respondents about their commute into a `data.frame` as follows:

```
df <- data.frame(travel_mode, time)
df
```

```
  travel_mode time
1       train   25
2       train   20
3        walk   15
4       cycle   10
5        walk   17
6       train   30
```

The variable names are listed on top with the values underneath. On the side we can see the numbers 1 to 6, which index the rows of the `data.frame`.

We can also view the `data.frame` in RStudio by clicking on `df` in the Environment tab. You could also open this by typing `View(df)` in the console. The first row means that the first respondent took the train and it took 25 minutes. The second row means that the second respondent also took the train and it took 20 minutes.

When variables are organized in a `data.frame`, it becomes very easy to summarize the data and make visualizations with them. We will learn how to do this in the upcoming chapters.

4.2.6 Lists

A `data.frame` is actually a special type of `list`, which is another object type. While all elements of a vector (created with the `c()` function) must have the same type (numeric, logical or factor), a `list` can have elements of any type, and also any length.

Here is an example of a list:

```
my_list <- list(x = 1:3, y = TRUE, z = c("a", "b"))
```

It has elements that are numeric, logical and character vectors, and the elements all have different lengths (3, 1 and 2).

A `data.frame` can have elements/columns of different types (such as in the travel mode example above, which had numeric and factor variables), but all
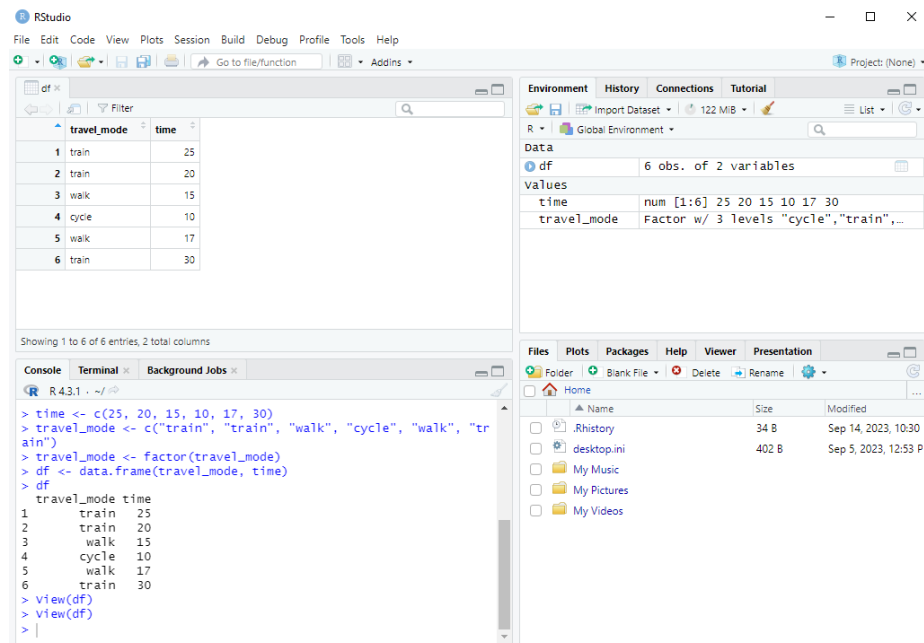
Figure 4.1: Viewing a dataframe in RStudio

elements/columns of a `data.frame` must have the same length (unlike a `list` where any length is possible).

Chapter 5

Operations on Vectors

In this chapter we will learn how to do some operations on vectors in R.

5.1 Indexing

Suppose we have a vector `a` with 5 elements and we wanted to isolate the 3rd element of it. We can do this with what is called *indexing*. To get the 3rd element of a vector `a`, we do `a[3]`. Let's see this with an example:

```
a <- c(1, 2, 4, 3, 2)
a[3]
```

```
[1] 4
```

We can also extract multiple elements of the vector using a vector of indices inside the `[]`. For example, suppose we wanted to get the 1st, 3rd and 4th element of `a`. We would put the vector `c(1, 3, 4)` inside the square brackets:

```
a[c(1, 3, 4)]
```

```
[1] 1 4 3
```

We can also extract elements of a vector using a logical vector. Doing this will extract the elements where the logical vector is `TRUE`. To do this, the logical vector needs to have the same length as the vector we are trying to index. Like above, if we want the 1st, 3rd and 4th element of `a`, we can use a vector with `TRUE` in the 1st, 3rd and 4th element and `FALSE` everywhere else:

```
a[c(TRUE, FALSE, TRUE, TRUE, FALSE)]
```

```
[1] 1 4 3
```

Suppose I want everything in a vector except one element: I want to exclude one element from the vector. For example, suppose I want to see the entire vector `a` except the 2nd element. We can do this using `-2` in the brackets:

```
a[-2]
```

```
[1] 1 4 3 2
```

5.2   Sequences

Often it is useful to create a sequence of numbers. For a simple sequence like 1, 2, 3, ..., 10, we can just do:

```
1:10
```

```
[1]  1  2  3  4  5  6  7  8  9 10
```

We can also make the sequence go backwards by reversing the numbers:

```
10:1
```

```
[1] 10  9  8  7  6  5  4  3  2  1
```

For sequences that don't jump in 1s we can use the `seq()` function. Suppose we wanted to have a sequence from 10 to 100 with steps of 10. We do that with:

```
seq(from = 10, to = 100, by = 10)
```

```
[1]   10   20   30   40   50   60   70   80   90 100
```

Instead of specifying the step length with `by`, we can alternatively specify the length of the sequence. Suppose I wanted to have a sequence going from 0 to 1 in equal steps with 5 numbers in total. I can do that using the `length.out` option:

```
seq(from = 0, to = 1, length.out = 5)
```

```
[1] 0.00 0.25 0.50 0.75 1.00
```

5.3   Repeating Numbers

If I wanted to create a vector which is 1 repeated 5 times, I could do:

```r
c(1, 1, 1, 1, 1)
```

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

But this would get very annoying to type and I could easily make a mistake if I wanted to make many more 1s. If we want to repeat a number many times, we can use the `rep()` function. For example, if we want to make 100 1s, we would do:

```r
rep(1, times = 100)
```

```
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[38] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[75] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

At the start of Chapter 3 we briefly mentioned that the `[1]` you see at the start of the output meant that the number we saw was the first one. Because with this example we have many numbers that run onto multiple lines, we can see a `[38]` at the start of line 2. This means that the first 1 on the 2nd line is the 38th element of the vector. The `[75]` on the 3rd line means the 1st one on that line is the 75th element.

The `rep()` function can also be combined with vectors. Suppose I wanted to repeat 1, 2, 3 four times:

```r
rep(1:3, times = 4)
```

```
 [1] 1 2 3 1 2 3 1 2 3 1 2 3
```

And if I instead wanted to repeat 1, 2, 3, **each** 4 times, I would use the `each` option:

```r
rep(1:3, each = 4)
```

```
 [1] 1 1 1 1 2 2 2 2 3 3 3 3
```

5.4 Summary Statistics for Vectors

We can get summary statistics for vectors using functions. Let's look at some common ones using a simple vector with the sequence 1 to 10:

```r
a <- 1:10
a
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

Get the number of elements of `a`:

```
length(a)
```

`[1] 10`

Get the minimum value in `a`:

```
min(a)
```

`[1] 1`

Get the maximum value in `a`:

```
max(a)
```

`[1] 10`

Get the average of all elements in `a`:

```
mean(a)
```

`[1] 5.5`

Get the median of all elements in `a`:

```
median(a)
```

`[1] 5.5`

Note on the median: Normally the median orders all elements of the vector and gives the element in the middle. Because we have an even number of elements in `a` (10 elements), the median is the average of the two values in the middle after sorting. Because it's already sorted, these middle values are 5 and 6, so the median is $(5 + 6)/2 = 5.5$.

Get the sum of all elements in `a`:

```
sum(a)
```

`[1] 55`

A useful way to quickly summarize a numeric vector is with the `summary()` function, which gives the minimum, maximum, mean, median and interquartile range:

```
summary(a)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    3.25    5.50    5.50    7.75   10.00
```

Another useful way of summarizing data is to tabulate it: to count the number of occurrences of each value. We can do that with the `table()` function:

```
a <- c(1, 3, 2, 4, 4, 2, 4)
table(a)
```

```
a
1 2 3 4
1 2 1 3
```

The output here means that 1 appeared once, 2 appeared twice, 3 appeared once and 4 appeared three times.

Chapter 6

Comparing Vectors

6.1   Comparing Numerical Vectors

Consider the following two vectors:

```
a <- 1:5
a
```

```
[1] 1 2 3 4 5
```

```
b <- 5:1
b
```

```
[1] 5 4 3 2 1
```

To find the indices of elements where $a > b$, we can use `a > b`. This returns a logical vector which is `TRUE` when a is greater than b and is `FALSE` otherwise:

```
a > b
```

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

Thus $a > b$ in the 4th and 5th element only, as $4 > 2$ in the 4th element and $5 > 1$ in the 5th element.

For $a \geq b$ (greater than or equal to) we use `a >= b`.

```
a >= b
```

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

The condition is satisfied in the 3rd element too, as $3 \geq 3$.

31

Similarly, for "less than" we use `<` and for less than or equal to we use `<=`:

```
a < b
```

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

```
a <= b
```

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

For $a = b$, we use `a == b`. We need to use two equal signs, because if we did `a = b`, it would replace the vector `a` with the vector `b`. Thus we use `==` to ask if the respective elements of the two vectors are equal:

```
a == b
```

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

Thus $a = b$ only in the 3rd element.

For $a \neq b$ (a not equal to b), we use `a != b`:

```
a != b
```

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

It's also possible to compare a vector to a single number. For example, like:

```
a > 3
```

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

But what is not possible is comparing a vector with 5 elements to a vector with only 4 elements. Either the two vectors should have the same length, or at least one of them has only 1 element.

6.2   Comparing Logical Vectors

Consider the following two logical vectors:

```
a <- c(TRUE, TRUE, FALSE, FALSE)
b <- c(TRUE, FALSE, TRUE, FALSE)
```

If we want to know where both a and b are `TRUE`, we use the *logical AND* operator `&`:

```
a & b
```

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

a and *b* are only both `TRUE` in first element.

To see when either *a* or *b* are `TRUE`, we use the *logical OR* operator `|`:

```
a | b
```

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

At least one of *a* or *b* are `TRUE` in all but the last element.

Suppose we wanted to return a logical vector which tells us when both *a* and *b* are `FALSE`. We can do that using the *logical NOT* operator `!`. First let's see what happens when we use the NOT operator on just `a`:

```
!a
```

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

Essentially it just flips the `TRUE`s to `FALSE`s and the `FALSE`s to `TRUE`s. To see when both *a* are *b* are `FALSE` we do:

```
!a & !b
```

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

Thus, this only happens in the 4th element.

Chapter 7

R Scripts

Up to now we have been writing commands directly into the R console. This is all fine if all you want to do is try out a few different simple commands. However, when working on a project with some data you will often be executing many commands and it's easy to lose track of what you are doing. It's also very easy to make mistakes. *R scripts* are a solution to this problem. An R script is a text file where you can write all of your commands in the order you want them run, and then you can tell RStudio to run the entire file of commands. You can also ask it to only run part of the file. This has many advantages:

- If you have run 10 commands to calculate something and then afterwards you decide to change what happened in one of the earlier commands, you would usually have to type all the commands again. In an R script you would just need to edit the line with that command. So R scripts can save you a lot of time.
- You or anyone else can easily reproduce your work by re-running the R script.
- By having all the commands in a script you can more easily spot any mistakes you might have.
- It is a way of saving your work.

Therefore it's best practice to write your commands in an R script.

7.1   Creating a New R Script

To get started, go to `File` → `New File` → `R Script` in RStudio. You can also use the `Ctrl+Shift+N` keyboard shortcut, or use the first toolbar button directly under `File`.

Note to Mac Users: The keyboard shortcut will be `Cmd+Shift+N` on a Mac. In general you will replace `Ctrl` with `Cmd` (and `Alt` with `Option`) in all keyboard

shortcuts that follow.

Test it out by typing a few commands into the script:

```
a <- 2
b <- 3
a + b
```

Note: The `Alt + -` shortcut to type `<-` also works in R scripts.

Your code in the R script should look like this:
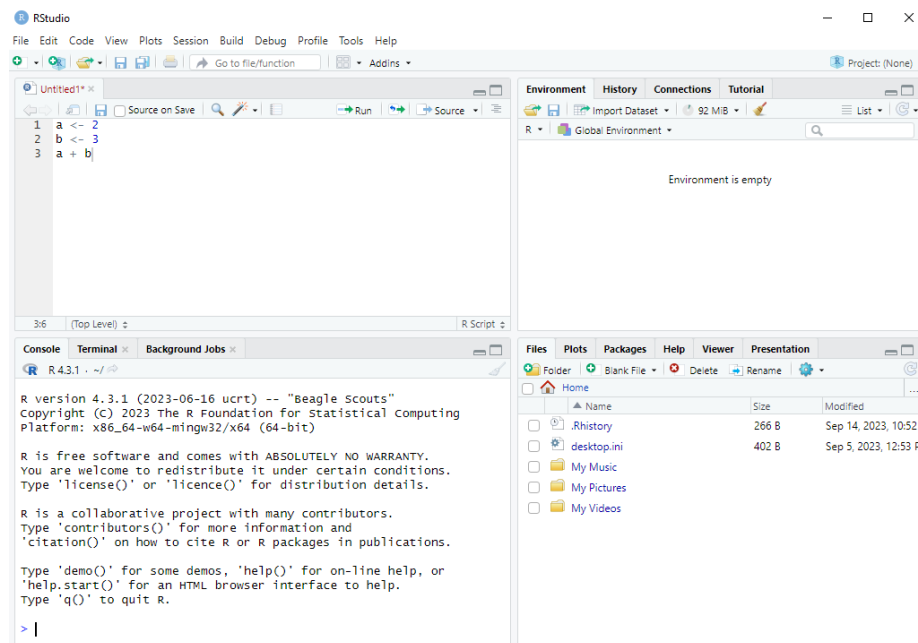
Figure 7.1: R scripts in RStudio

7.2   Running the Commands in an R Script

There are several different ways to run the commands in an R script.

7.2.1   Selecting Lines and Running

One way to run these lines is to do the following:

1. Select the lines, either with your mouse or with the keyboard shortcut `Ctrl+A`.

2. Running the selected lines, either with the `Run` toolbar button on top of the R script, or with the keyboard shortcut `Ctrl+Enter`.

What is nice about this method is you can run a subset of the commands in your R script. Instead of selecting all the lines, you just select the lines you want to run. This can be useful if some of the lines in your code are slow to run and you don't need to run those lines again.

7.2.2 Sourcing

7.2.2.1 Sourcing with Echo

Another way to run the entire script is to *source* the script. You can do this with the keyboard shortcut `Ctrl+Shift+Enter` or by clicking the down-arrow next to the `Source` button at the top of the script, and clicking "Source with Echo".

With this approach you don't need to select the lines first. It always runs the entire file.

7.2.2.2 Sourcing without Echo

You will notice that when you run source, the `source()` command appears in the console with the option `echo = TRUE`. The echo option prints all of the commands and the output on the screen. It is also possible to source without echo which then only prints what you want it to. This can be useful if you have a very long script and only want to see the output of a few different things in it when running it. You can do this by clicking the drop-down option next to the `Source` button and clicking "Source". You can also use the keyboard shortcut `Ctrl+Shift+S`.

If we try this using the example script above we will see that it does not print anything at all. We don't see the output of `a + b`. To be able to print the output of a line on the screen when using `source()` without echo, we need to use the `print()` function. We need to change our script to be:

```
a <- 2
b <- 3
print(a + b)
```

When we source the script without echo we then see the output:

```
[1] 5
```

7.3   Commenting in R

7.3.1   Commenting as Annotation

When writing an R script it is good practice to add comments throughout to explain what you are doing. This helps other people who are reading your code to understand what you are doing and your intentions. Most of all, though, it helps you to understand your own code when you look back at it after a few months. To add a comment in R you simply need to type a `#` and anything you write after the `#` is not run by R. For example:

```r
# Set values of a and b:
a <- 2
b <- 3
# Compute the sum of a and b and print:
print(a + b)
```

```
[1] 5
```

You can also add comments after a command on the same line. Everything before the `#` is run by R, and everything after and including the `#` is not run:

```r
a <- 2          # set a equal to 2
b <- 3          # set b equal to 3
print(a + b)    # Compute the sum of a and b and print:
```

```
[1] 5
```

7.3.2   Commenting to Not Run Certain Commands

If you have written some commands but you don't want to run them when you run/source your script, you can "comment them out". You just put a `#` before each line you don't want to run to turn them into "comments". You can "comment out" many lines at the same time by selecting the lines you want to comment out and using the `Ctrl+Shift+C` shortcut (or by going to `Code →` `Comment/Uncomment Lines`). If you want to "uncomment" these lines, you just need to select them and use the `Ctrl+Shift+C` shortcut again.

Chapter 8

Loading a CSV Dataset

Up to now we have been creating vectors and dataframes in R by hand. But usually we will be working with bigger datasets that we want to read into R from a file. R is able to read files from many different formats, such as text files and Excel files. It can also read in datasets created by other software, such as Stata, SPSS, or SAS. It can also read datasets straight from websites by providing the URL instead of the file name.

We will start by learning how to read in a comma-separated values (CSV) dataset from a CSV file. This is the most common format for datasets.

8.1   Structure of a CSV file

Before learning how to read a CSV file into R, let's first understand the structure of a CSV file.

We will do this with the example `data.frame` we saw before about how long it took people to travel to class. The dataset can be represented by the following table:

travel_mode	time
train	25
train	20
walk	15
cycle	10
walk	17
train	30

A CSV file containing this dataset would look like this:

```
travel_mode,time
train,25
train,20
walk,15
cycle,10
walk,17
train,30
```

In a CSV dataset, the first row includes the names of the variables. Each of the names are separated by commas, hence the name "comma-separated values". So the first line is `travel_model,time`. The second and following rows contain the values of each of the variables, again separated by commas. Each line needs to have the same number of commas so that the data can be read in as rectangular.

Commas as Part of Character Variables

Sometimes the values of the variables contain commas. For example, a travel mode could be `"train,cycle"` if someone both cycled and took the train. If we put in a line like `train,cycle,28` in the CSV file, it would think that there are 3 data points for that line, when there should be only 2. In order to distinguish when the comma is part of the data and when it separates columns, we can put the character variables in double quotation marks. So we could more safely store the dataset instead as follows:

```
"travel_mode","time"
"train",25
"train",20
"walk",15
"cycle",10
"walk",17
"train",30
```

Decimal Commas in CSV Files

Outside the Anglosphere (e.g. USA, UK, etc.), such as in the Netherlands, the decimal separator is a comma instead of a point. For example, "two and a half" is represented by 2,5 instead of 2.5. In this case we have two options:

1. We can surround all the numbers by double quotation marks.
2. We can use semicolons (;) to separate columns instead of commas, and inform R that we want to use semicolons as separators instead of commas.

For assignments and exams, however, we will always be dealing with datasets with period decimal separators.

8.2 Reading in a CSV file

We will learn how to read in the example dataset above from a file. To do so, you need to first do the following steps to save it:

1. Copy the text of the CSV file above using the clipboard icon.
2. Open a new text file with `File → New File → Text File`.
3. Paste in the contents of your clipboard so the file has the data in it.
4. Go to `File → Save As...` and save the file as `test.csv` somewhere on your computer.

Alternatively, download the file here.

Now, to read in the file into R, we need to tell R exactly where on your computer the file is. We need to give R either the *absolute* path (or full path) to the file, or the *relative* path from where R is currently working. We will now explain what these are.

8.2.1 Absolute Paths

The absolute path is the full path to the file. On Windows, the full path is something like `C:\Users\username\Documents\test.csv`. On a Mac, the full path is something like `/Users/username/Documents/test.csv`. On Linux, the full path is something like `/home/username/Documents/test.csv`.

There are many different ways you can find the full path of your file, but the most convenient way for our purposes is to use the `file.choose()` command. When you run the `file.choose()` command in your console, a file browser will appear:

You then navigate to the file on your computer and press "Open".

After doing so the file doesn't actually open, but instead the full path of the file will print in the console surrounded by quotes, which we can then use in our code:

We can then copy this (leaving out the `[1]`) and paste it into the `read.csv()` command like this:

```
df <- read.csv("e:\\Users\\cbtwalsh\\Documents\\test.csv")
```

Running this will read in the file as a `data.frame` called `df`. We will then be able to see `df` in our Environment and can view it by clicking on it there, or else typing `View(df)` in the console.

Technical note: Windows users will notice that the file path used in the `read.csv()` command uses "double backslashes" (\\) instead of single backslashes that you would normally see in a file path. We need to use these double backslashes in R because R uses the single backslash as an "escape character". To create a single actual backslash in R we always need to use two backslashes. This
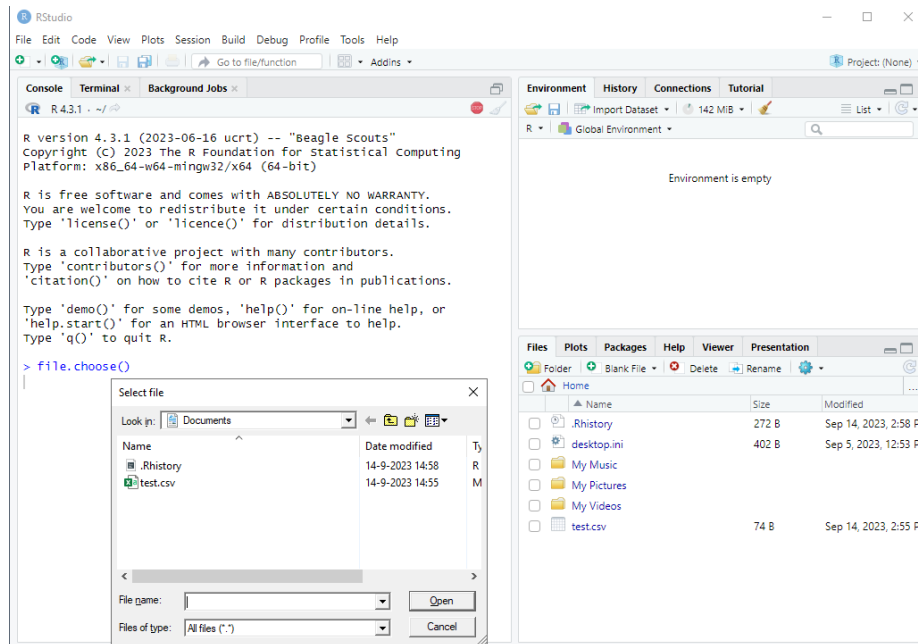
Figure 8.1: `file.choose()` command
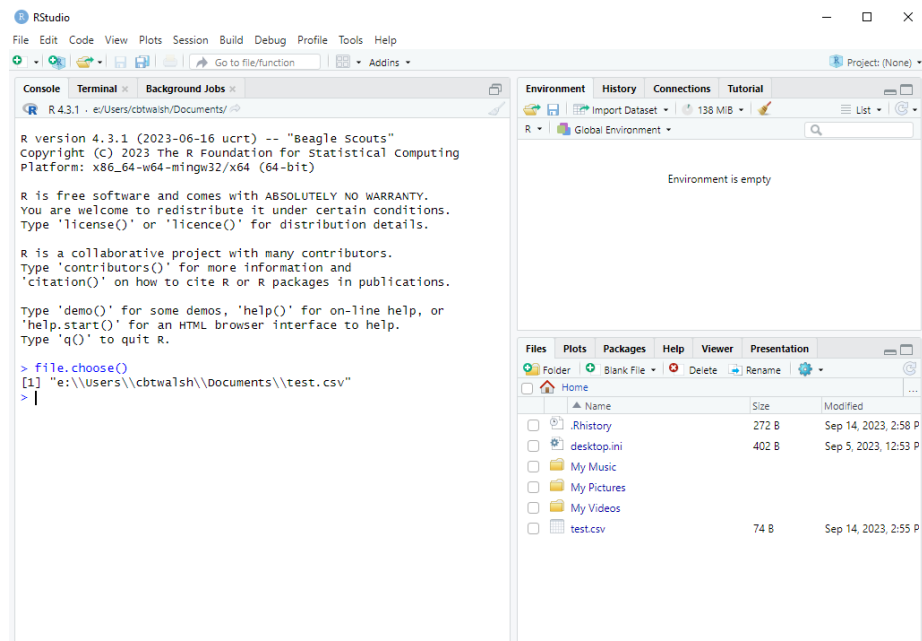
Figure 8.2: `file.choose()` command

is another reason why using the `file.choose()` command to get the file path is so useful. Otherwise we would have to add in all the extra backslashes in manually. It's also possible to use forward slashes (`/`) instead, in which case we only need to use one. For example: `"e:/Users/cbtwalsh/Documents/test.csv"`.

8.2.2 Relative Paths

Reading in a file using the absolute path like above works well until you start collaborating on a project with someone. Because you don't have the same username, the code that you write won't work on their computer because the file paths will be different. If two people are working on an R script together on Dropbox, they will have to constantly change the the lines in the R scripts that read in data.

A solution to this is to use relative paths for reading in data, or better yet, *projects* in RStudio (see below).

At any given time, R has a *current working directory* which is a folder somewhere on your computer. You can find out where this is using the `getwd()` command. If the file `test.csv` is in the current working directory, you can read it in without using the full path:

```
df <- read.csv("test.csv")
```

One way to change the current working directory to the location where you have saved your data is to use the `setwd()` command. You can use the `file.choose()` command like before to navigate to where that is. You would then copy the full file path except for the part containing the file name. In the example above, you would do the following:

```
setwd("e:\\Users\\cbtwalsh\\Documents\\")
df <- read.csv("test.csv")
```

This way, the collaborators would only need to set the current working directory once, and then the `read.csv()` commands wouldn't need to be changed. This is useful when commands like this are run several times.

Suppose the dataset wasn't in `e:\Users\cbtwalsh\Documents`, but rather `e:\Users\cbtwalsh\Documents\datasets`, a sub-folder of the Documents folder. We can still read in the file when the current directory is `"e:\\Users\\cbtwalsh\\Documents\\"` by using the *relative path*. That is, we only need to give the path *relative to* the current working directory. In this case, it would be `"datasets/test.csv"`. Thus we could read in the data with:

```
setwd("e:\\Users\\cbtwalsh\\Documents\\")
df <- read.csv("datasets/test.csv")
```

8.2.3 RStudio Projects

If you are collaborating with someone, setting the current working directory in the way above still requires changing a line when you run it on a different computer, which is inconvenient. A way that avoids this entirely is the *Project* feature in RStudio.

8.2.3.1 Creating an RStudio Project

To make use of this feature, we first need to create a project. The easiest way to do this is to first ensure that the folder that you want to be the project already exists somewhere on your computer. You can go to File → New Project... (or click the Project toolbar button) select "Existing Directory" and then browse to the location on your computer where it is. Then click "Create Project" and R will switch to the project.

Here are screenshots of the steps:
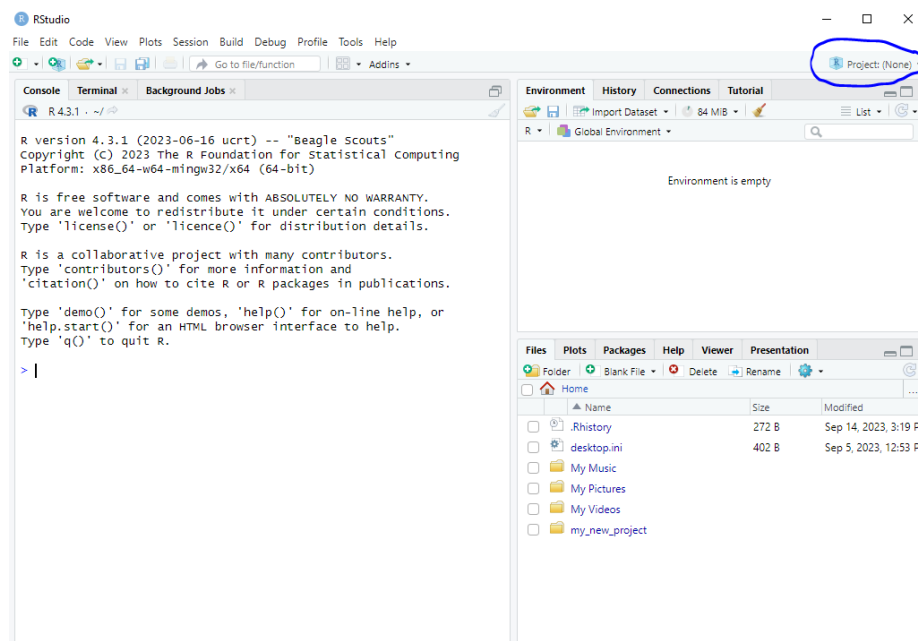
Figure 8.3: Step 1: Click on the Project toolbar button.

R then switches to the project. We can see that:

- The **Files** tab now shows the files from the project folder.
- RStudio creates a file called `my_new_project.Rproj` in your project folder.
- We can also see that R has switched its current working directory to the project folder. We can check this with the `getwd()` command:
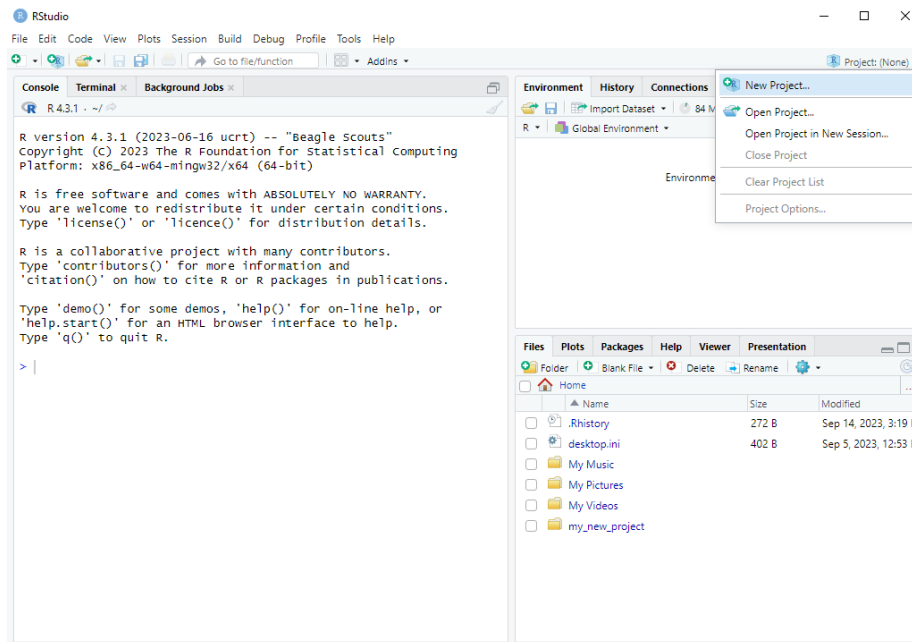
Figure 8.4: Step 2: Click on "New Project".
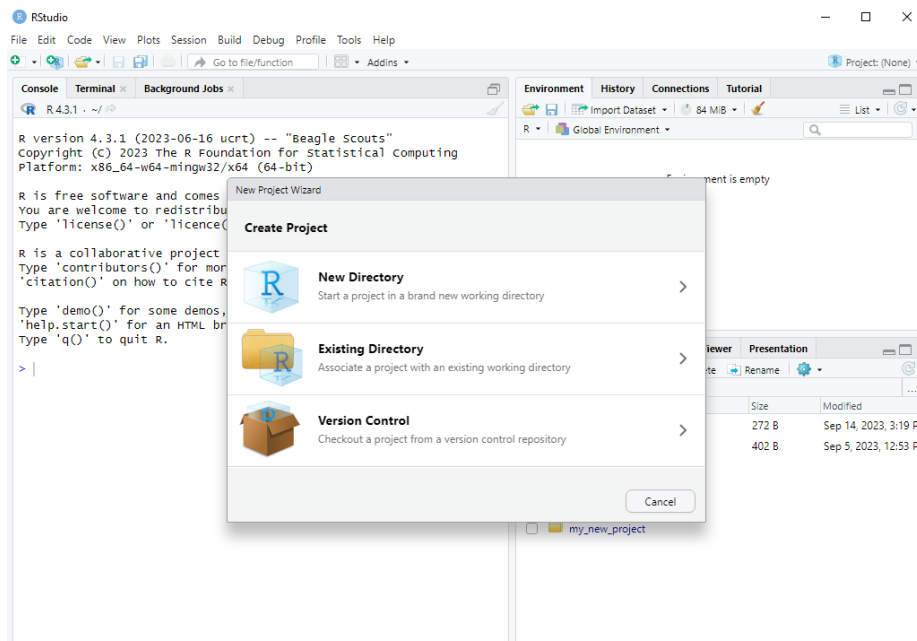
Figure 8.5: Step 3: Click on "Existing Directory".
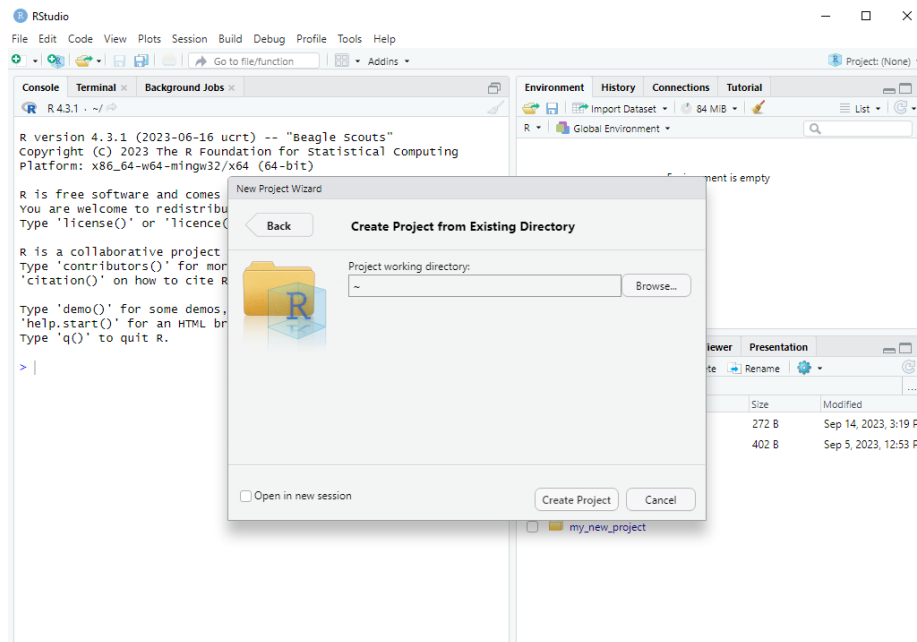
Figure 8.6: Step 4: Click "Browse".
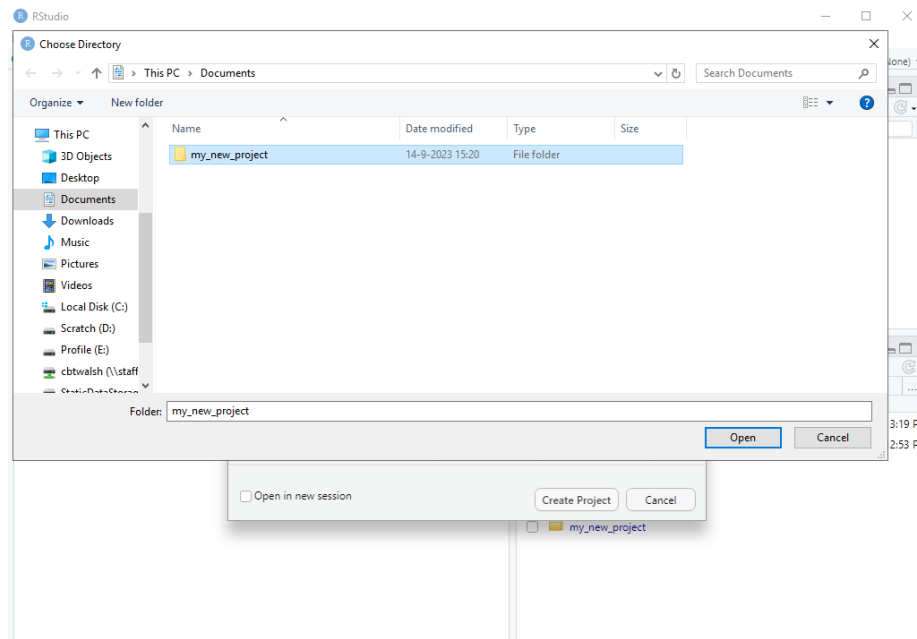
Figure 8.7: Step 5: Navigate to the project folder, click on it, and press "Open".
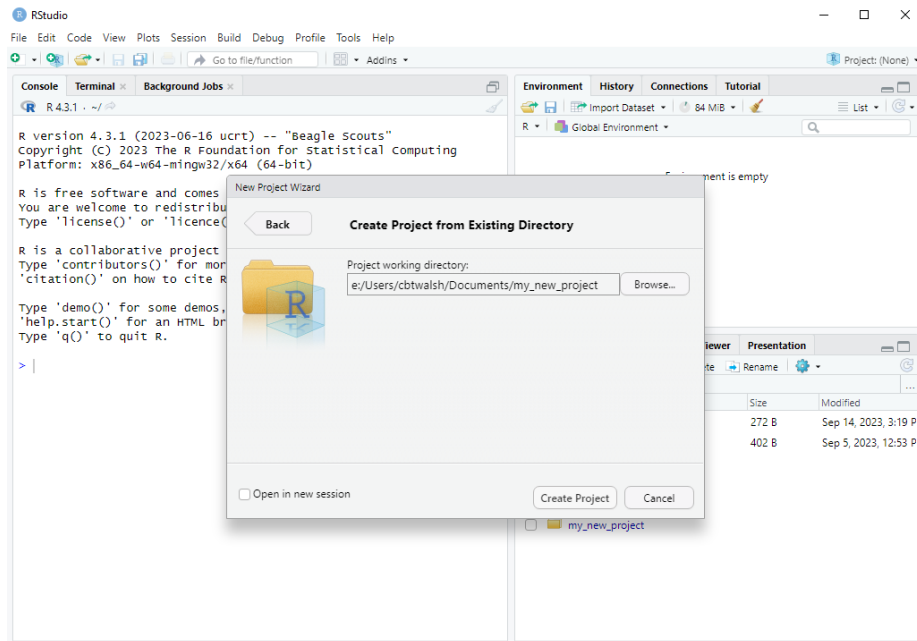
Figure 8.8: Step 6: Click "Create Project".
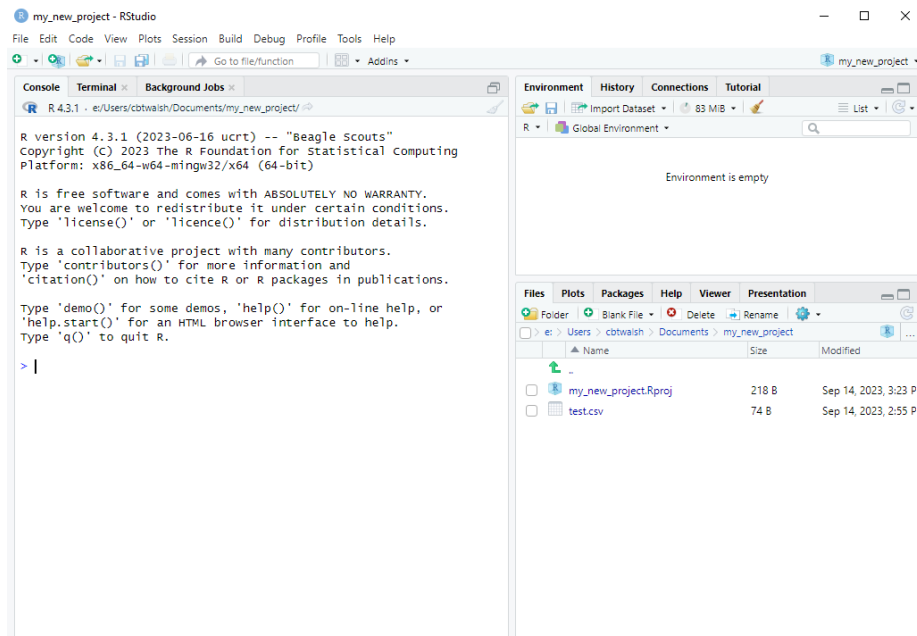
Figure 8.9: RStudio after creating the project.
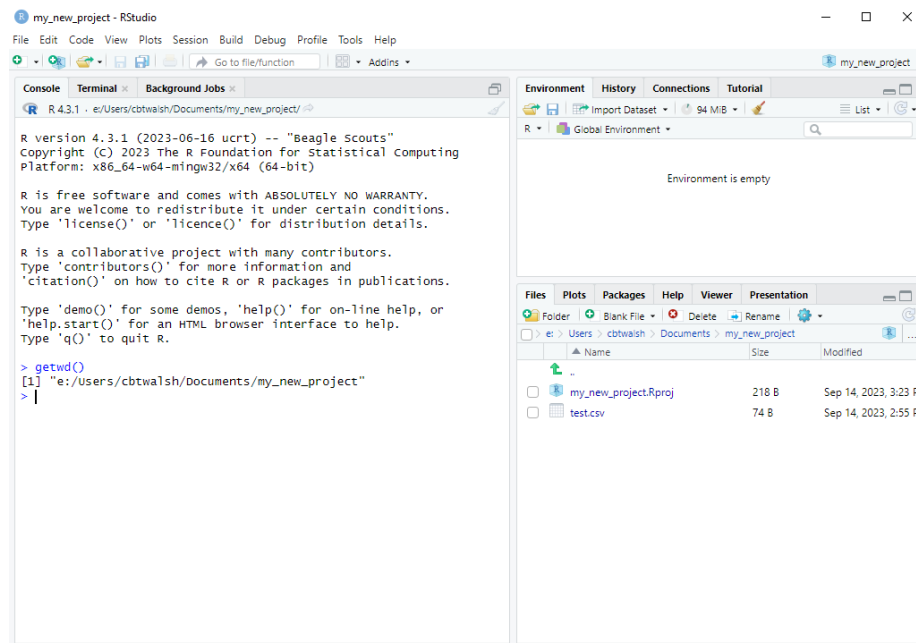
Figure 8.10: Confirming the new working directory with `getwd()`

8.2.3.2 Reading in Data within an R Project

To read in a file in the project directory, such as `test.csv`, all we need to do is this:

```
df <- read.csv("test.csv")
```

This way, we don't need to find and paste in the full file path, we don't need to change the working directory, and if we are collaborating with someone we don't ever need to change any of the lines. There are also a number of other (more advanced) benefits from using the Projects feature. For these reasons, I recommend **always** using the Project feature in RStudio.

Chapter 9

R Packages

Up to now we have been using the standard functions that come by default with R. However, certain operations that we want to do are not available as functions in "base R" (the default functionality within R). Whenever this happens, we need to load functions from other "packages" that allow us to do the operation we want to do.

Anyone in the R-using community is free to write their own functions and publish them as packages for other people to use. This is great because it means there is a package for almost everything you could think of doing. When academics develop new statistical models they often also publish an R package with their journal publication, allowing other researchers to use their models easily.

In this chapter we will learn how to install and load packages through an example.

9.1   Example Setting: Reading Excel Files

There is no function in base R that allows you to easily read in an Excel file into R as a `data.frame`. One solution to this problem would be to export the data from the Excel file into a CSV file. In Excel you would do `File → Save As` and choose "CSV" under `Save as type`). Then we can just read in the data using the `read.csv()` function we learned about in Chapter 8.

This approach would work fine in many circumstances, but you might have a situation where you need to read many Excel files, or the Excel file is frequently being updated. Then always having to convert the file to CSV becomes very time-consuming and annoying. Also, we want to easily be able to replicate the steps in our work. By having all the steps you do in your R script, anyone can see exactly what you have done from the initial "raw data". This makes your work replicable and transparent.

There are several R packages that allow you to read in Excel files, but we will focus our attention on one of them: the `readxl` package.

To test this out, open Excel and populate it with the data below and save it as `test.xlsx` in your project directory:

x	y
3	5
8	7
2	1

9.2   Installing packages

9.2.1   From the command line

One way to install a package is from the command line using the `install.packages()` function.   You just need to put the package name in quotation marks as the argument:

```
install.packages("readxl")
```

If you include this line in your R script, it will re-install the package every time you run/source your code. For that reason it's better to type it in the command line.

9.2.2   From RStudio

In RStudio you can go to `Tools → Install Packages...`, type the name of the packages in the "Packages" box, and press "Install" (leaving all the other options as default). Here is an example:

Note: the computers on campus have a very wide range of packages already installed. In the exam you won't have to install any packages.

9.3   Loading packages

The function from the `readxl` package that we want to use is called `read_excel()`. We want to use it to read in the `test.xlsx` file we created earlier. If we try to use the function with the command `read_excel("test.xlsx")`, we will get the following error:

```
> read_excel("test.xlsx")
Error in read_excel("test.xlsx") : could not find function "read_excel"
>
```
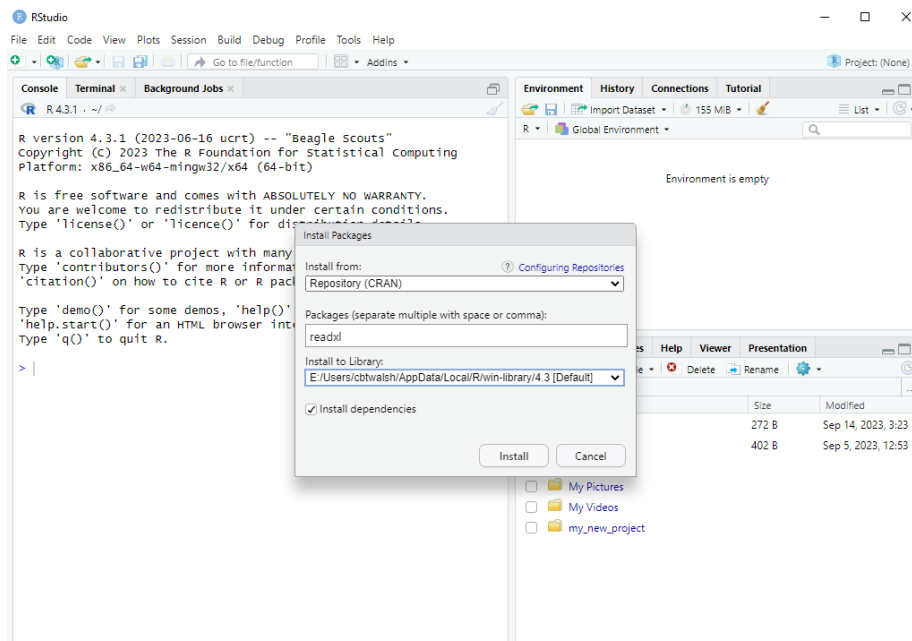
Figure 9.1: Installing a package using the RStudio dialog box.

This is because simply installing a package doesn't mean the functions are available to use. We need to load the package first. This is done using the `library()` function. When using the `library()` function, we don't need to put the package name in quotes (but we still can – both work). This is unlike the `install.packages()` function, where quotes are required.

After loading the package, we can read in the data from the Excel file:

```
library(readxl)
df <- read_excel("test.xlsx")
df
```

```
# A tibble: 4 x 2
      x     y
  <dbl> <dbl>
1     1     2
2     4     4
3     5     6
4     3     3
```

Note that the `read_excel()` function read in the data as a `tibble`, which is like a `data.frame` with a few extra features, like printing the dimensions of the data. For this course we can just think of tibbles and dataframes as the same

thing. If you want to convert this `tibble` into a pure `data.frame` you can use the `data.frame()` function around `read_excel()`:

```
df <- data.frame(read_excel("test.xlsx"))
df
```

```
  x y
1 1 2
2 4 4
3 5 6
4 3 3
```

Finally, note that if a package is not installed you will receive an error message. For example, if you tried to load the `readxl` package before installing it you would get the error:

```
Error in library(readxl) : there is no package called 'readxl'
```

When you get this error, you know you need to install the package first.

9.4   Data Formats from other Software (Optional)

In the future you may encounter data files stored in other formats saved from other statistical software programs, such as Stata, SPSS or SAS. The `haven` package in R is able to read in each of these. You can install the `haven` package and load it like before:

```
install.packages("haven")
library(haven)
```

The functions to read in each of these data types are shown in the table below:

Statistical Software	File extension	Function from `haven` package
Stata	.dta	`read_stata()`
SPSS	.sav	`read_spss()`
SAS	.sas7bdat	`read_sas()`

In the assignments and exam, however, we will not use any of these three data formats (`.dta`, `.sav` or `.sas7bdat`).

Chapter 10

Dataframes: Indexing

In Chapter 4 and Chapter 8 we encountered dataframes. In the following three chapters we will learn more about how to work with them. In this chapter will focus on indexing with dataframes.

10.1 Running Example: The Eredivisie Results from 2022/23

The dataset we will work with in the next three chapters contains the team, number of wins, draws, losses, goals for, and goals against for all 18 teams and 38 matches from the 2022/23 season of the Eredivisie. The Eredivisie is the top Dutch association football league. Here, "goals for" means the total number of goals the team scored that season, whereas "goals against" is the total number of goals the team conceded that season.

You can copy the code chunk below directly into R and it will read the data in as a `data.frame`. This is another way of reading in datasets into R. You can provide the contents of a CSV file directly into the `read.csv()` function we have seen before instead of giving the filename. To do this we need to use the `text` option. I am doing it this way to save you time having to copy the data and save a new file on your computer, but it's also good to see other ways of reading in data.

```
df <- read.csv(text = "
         team, wins, draws, losses, goals_for, goals_against
           AZ,   20,     7,      7,        68,            35
         Ajax,   20,     9,      5,        86,            38
     Excelsior,    9,     5,     20,        32,            71
      FC Emmen,    6,    10,     18,        33,            65
```

```
      FC Groningen,    4,     6,    24,        31,          75
        FC Twente,    18,    10,     6,        66,          27
        FC Utrecht,   15,     9,    10,        55,          50
       FC Volendam,   10,     6,    18,        42,          71
         Feyenoord,   25,     7,     2,        81,          30
   Fortuna Sittard,   10,     6,    18,        39,          62
   Go Ahead Eagles,   10,    10,    14,        46,          56
               NEC,    8,    15,    11,        42,          45
               PSV,   23,     6,     5,        89,          40
      RKC Waalwijk,   11,     8,    15,        50,          64
        SC Cambuur,    5,     4,    25,        26,          69
   Sparta Rotterdam,  17,     8,     9,        60,          37
           Vitesse,   10,    10,    14,        45,          50
      sc Heerenveen,  12,    10,    12,        44,          50
", strip.white = TRUE)
```

Note: The `strip.white` option I have used in this command is to remove the empty spaces before and after the team names.

10.2   Indexing with Dataframes

In Chapter 5 we learned that we can get the 3rd element of a vector `a` with `a[3]`. We can also extract elements of a dataframe in a similar way. To get the 2nd row and 3rd column of a dataframe, we do:

```
df[2, 3]
```

```
[1] 9
```

Inside the square bracket we first specify the rows, then after a comma we specify the columns.

We can also put multiple indexes in each part. Suppose we want a smaller dataframe of only the rows with Ajax, Feyenoord and PSV and only the columns with the team name and number of wins. We first check which rows those teams occupy (2, 9 and 13) and which columns those variables are in (1 and 2). We then do:

```
df[c(2, 9, 13), c(1, 2)]
```

```
        team wins
2       Ajax   20
9  Feyenoord   25
13       PSV   23
```

If we leave the columns part blank, it will give us the entire row. For example, to get all the results for just Ajax we just get the 2nd row:

```
df[2, ]
```

```
  team wins draws losses goals_for goals_against
2 Ajax   20     9      5        86            38
```

Similarly, if we leave the row part blank and only give column indices, it will give us all rows for those columns. If we just want the column of wins we can do:

```
df[, 2]
```

```
 [1] 20 20  9  6  4 18 15 10 25 10 10  8 23 11  5 17 10 12
```

We can also get a column of a dataframe using the name of the variable. For example, if we want to get the `goals_for` column, we can do:

```
df$goals_for
```

```
 [1] 68 86 32 33 31 66 55 42 81 39 46 42 89 50 26 60 45 44
```

The dollar symbol here is what is called an *extraction operator*. The dollar symbol is required because `goals_for` is part of `df`. The variable `goals_for` is not a standalone vector in our environment. The `df$` tells R to look for `goals_for` inside `df`.

We can also use the name of the variable in the part where we specify the column indices:

```
df[, "goals_for"]
```

```
 [1] 68 86 32 33 31 66 55 42 81 39 46 42 89 50 26 60 45 44
```

We can also place multiple variable names in there:

```
df[, c("team", "goals_for")]
```

```
           team goals_for
1            AZ        68
2          Ajax        86
3      Excelsior        32
4      FC Emmen        33
5  FC Groningen        31
6     FC Twente        66
7    FC Utrecht        55
```

```
8         FC Volendam       42
9           Feyenoord       81
10   Fortuna Sittard        39
11   Go Ahead Eagles        46
12               NEC        42
13               PSV        89
14      RKC Waalwijk        50
15        SC Cambuur        26
16  Sparta Rotterdam        60
17           Vitesse        45
18      sc Heerenveen       44
```

Finally, another way to get a single variable from a dataframe is to place the name of the variable in quotes inside *double square brackets*:

```
df[["goals_for"]]
```

```
 [1] 68 86 32 33 31 66 55 42 81 39 46 42 89 50 26 60 45 44
```

We can also subset rows of a dataframe using logical operators, just like we saw in Chapter 5. For example, suppose we wanted to only see the results for teams that won at least 20 matches. The following will return a logical vector which is TRUE if the team won at least 20 matches, and FALSE if they won 19 or fewer matches:

```
df$wins >= 20
```

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

The first two are TRUE, because AZ and Ajax won at least 20 matches (they both won exactly 20). The next two are FALSE because Excelsior and FC Emmen won less than 20 matches (they won 9 and 6, respectively).

If we use this inside the square brackets where we specify the row indices, we get the desired result:

```
df[df$wins >= 20, ]
```

```
        team wins draws losses goals_for goals_against
1         AZ   20     7      7        68            35
2       Ajax   20     9      5        86            38
9  Feyenoord   25     7      2        81            30
13       PSV   23     6      5        89            40
```

Chapter 11

Dataframes: Creating Variables

Here we will continue to work with the example dataframe from Chapter 10:

```
df <- read.csv(text = "
            team, wins, draws, losses, goals_for, goals_against
              AZ,   20,     7,      7,        68,            35
            Ajax,   20,     9,      5,        86,            38
        Excelsior,    9,     5,     20,        32,            71
         FC Emmen,    6,    10,     18,        33,            65
     FC Groningen,    4,     6,     24,        31,            75
        FC Twente,   18,    10,      6,        66,            27
       FC Utrecht,   15,     9,     10,        55,            50
      FC Volendam,   10,     6,     18,        42,            71
        Feyenoord,   25,     7,      2,        81,            30
  Fortuna Sittard,   10,     6,     18,        39,            62
  Go Ahead Eagles,   10,    10,     14,        46,            56
              NEC,    8,    15,     11,        42,            45
              PSV,   23,     6,      5,        89,            40
     RKC Waalwijk,   11,     8,     15,        50,            64
        SC Cambuur,    5,     4,     25,        26,            69
  Sparta Rotterdam,  17,     8,      9,        60,            37
          Vitesse,   10,    10,     14,        45,            50
     sc Heerenveen,   12,    10,     12,        44,            50
", strip.white = TRUE)
```

We can see that we are missing some of the columns we usually see when we look at the standings of an association football league. These are:

1. The goal difference.
2. The number of points the team has.
3. The rankings of the teams in the league.
4. The relegation status of each team.

What we will learn in this chapter is how to create these variables from the existing ones we already have.

11.1   Goal Difference

The goal difference in association football is "goals for" minus "goals against". To get the goal difference then, we subtract `df$goals_against` from `df$goals_for`:

```
df$goals_for - df$goals_against
```

```
 [1]  33  48 -39 -32 -44  39   5 -29  51 -23 -10  -3  49 -14 -43  23  -5  -6
```

This prints out the goal difference for all 18 teams, but it does not save it in the dataset. To do that, we need to assign this output to a new variable in `df`. We do that using the dollar symbol again:

```
df$goal_diff <- df$goals_for - df$goals_against
```

Now when we look at our dataset, there is a new variable in it:

```
df
```

```
               team wins draws losses goals_for goals_against goal_diff
1                AZ   20     7      7        68            35        33
2              Ajax   20     9      5        86            38        48
3          Excelsior    9     5     20        32            71       -39
4          FC Emmen    6    10     18        33            65       -32
5      FC Groningen    4     6     24        31            75       -44
6          FC Twente   18    10      6        66            27        39
7         FC Utrecht   15     9     10        55            50         5
8        FC Volendam   10     6     18        42            71       -29
9          Feyenoord   25     7      2        81            30        51
10  Fortuna Sittard   10     6     18        39            62       -23
11  Go Ahead Eagles   10    10     14        46            56       -10
12               NEC    8    15     11        42            45        -3
13               PSV   23     6      5        89            40        49
14      RKC Waalwijk   11     8     15        50            64       -14
15        SC Cambuur    5     4     25        26            69       -43
16  Sparta Rotterdam   17     8      9        60            37        23
17           Vitesse   10    10     14        45            50        -5
```

```
18    sc Heerenveen   12   10    12        44            50          -6
```

11.2 Total Points

In association football leagues, a team gets 3 points for a win and 1 point for a draw. They get 0 points for a loss. The formula for calculating the total number of points is then:

$$\text{Points} = 3 \times \text{Wins} + 1 \times \text{Draws} + 0 \times \text{Losses}$$

In our data we observe the number of wins, draws and losses, but not the total points. So we need to create this variable from the other ones using the formula above. We can do this with:

```
df$total_points <- 3 * df$wins + df$draws
```

We didn't need to include `df$losses` in the formula because when we multiply it by zero it won't make a difference.

Let's take a look at what we've created:

```
df[, c("team", "wins", "draws", "losses", "total_points")]
```

```
                team wins draws losses total_points
1                 AZ   20     7      7           67
2               Ajax   20     9      5           69
3          Excelsior    9     5     20           32
4           FC Emmen    6    10     18           28
5       FC Groningen    4     6     24           18
6          FC Twente   18    10      6           64
7         FC Utrecht   15     9     10           54
8         FC Volendam  10     6     18           36
9          Feyenoord   25     7      2           82
10   Fortuna Sittard   10     6     18           36
11   Go Ahead Eagles   10    10     14           40
12               NEC    8    15     11           39
13               PSV   23     6      5           75
14       RKC Waalwijk   11     8     15           41
15        SC Cambuur    5     4     25           19
16 Sparta Rotterdam   17     8      9           59
17           Vitesse   10    10     14           40
18     sc Heerenveen   12    10     12           46
```

11.3   Team Ranking

At the moment, the data are sorted alphabetically by team. But usually we see them sorted by their ranking in the league. What we will do now is sort the data by their ranking in the league, and also create a variable that shows the team's rank in the league.

In association football leagues, teams are ranked by the number of points they accumulated throughout the season. If two teams have the same number of points, we rank teams by their goal difference.[1]

We can create a ranking by *sorting* the data by the number of points and goal difference. We can do that using the `order()` function in R. The first argument is what we want to sort by (total points). For breaking ties we can include additional arguments. By default, `order()` sorts ascending so to sort descending we need to use the option `decreasing = TRUE`.

If we use the order function by itself, we get:

```
order(df$total_points, decreasing = TRUE)
```

```
 [1]  9 13  2  1  6 16  7 18 14 11 17 12  8 10  3  4 15  5
```

The 9 at the beginning means that the team with the most points is the one in the 9th position. Let's check which one that was:

```
df[9, ]
```

```
       team wins draws losses goals_for goals_against goal_diff total_points
9 Feyenoord   25     7      2        81            30        51           82
```

This is correct, because Feyenoord won the competition. The 2nd number 13 means the team in the 13th position (PSV) came second.

To actually sort the data, we need to use this function when specifying the row indices:

```
df <- df[order(df$total_points, decreasing = TRUE), ]
df[, c("team", "total_points", "goal_diff")]
```

```
           team total_points goal_diff
9     Feyenoord           82        51
13          PSV           75        49
2          Ajax           69        48
```

---

[1]If the date is `01/01/69`, the format `%d/%m/%y` will interpret it as January 1 **1969**. But if the date is `01/01/68`, it will interpret it as January 1 **2068**. All short-format years after 69 are put in the 1900s and all short-format years before 69 are put in the 2000s. You don't need to remember these details for the exam though because we won't ever use dates outside of 1969-2068.

```
1                 AZ        67        33
6          FC Twente        64        39
16 Sparta Rotterdam        59        23
7         FC Utrecht        54         5
18     sc Heerenveen        46        -6
14      RKC Waalwijk        41       -14
11   Go Ahead Eagles        40       -10
17            Vitesse        40        -5
12                NEC        39        -3
8         FC Volendam        36       -29
10    Fortuna Sittard       36       -23
3           Excelsior        32       -39
4           FC Emmen         28       -32
15         SC Cambuur        19       -43
5        FC Groningen        18       -44
```

If there are ties in total points, the function will keep the initial ordering. Here, both Go Ahead Eagles and Vitesse have 40 points, but Vitesse has a better goal difference (-5 instead of -10). But for the ranking to be correct, we need Vitesse to be ahead of Go Ahead Eagles. To do this, we add `df$goal_diff` as another argument to the `order()` function. This orders by goal difference whenever there is a tie in points:

```r
df <- df[order(df$total_points, df$goal_diff, decreasing = TRUE), ]
df[, c("team", "total_points", "goal_diff")]
```

```
              team total_points goal_diff
9         Feyenoord           82        51
13              PSV           75        49
2              Ajax           69        48
1                AZ           67        33
6         FC Twente           64        39
16 Sparta Rotterdam           59        23
7        FC Utrecht           54         5
18     sc Heerenveen          46        -6
14      RKC Waalwijk          41       -14
17            Vitesse          40        -5
11   Go Ahead Eagles          40       -10
12                NEC          39        -3
10    Fortuna Sittard          36       -23
8         FC Volendam          36       -29
3           Excelsior          32       -39
4           FC Emmen           28       -32
15         SC Cambuur          19       -43
5        FC Groningen          18       -44
```

Now we get the right ordering. We can also confirm that there are no ties in both total points and goal difference.

To create the ranking variable we can simply create a sequence from 1 to 18. We can do this with `1:18`. But another way to get 18 is to use `nrow(df)`, which is the number of rows in `df`:

```
df$ranking <- 1:nrow(df)
df[, c("team", "total_points", "goal_diff", "ranking")]
```

```
                team total_points goal_diff ranking
9          Feyenoord           82        51       1
13               PSV           75        49       2
2               Ajax           69        48       3
1                 AZ           67        33       4
6          FC Twente           64        39       5
16 Sparta Rotterdam           59        23       6
7         FC Utrecht           54         5       7
18     sc Heerenveen           46        -6       8
14     RKC Waalwijk           41       -14       9
17           Vitesse           40        -5      10
11   Go Ahead Eagles           40       -10      11
12               NEC           39        -3      12
10   Fortuna Sittard           36       -23      13
8        FC Volendam           36       -29      14
3          Excelsior           32       -39      15
4           FC Emmen           28       -32      16
15         SC Cambuur           19       -43      17
5       FC Groningen           18       -44      18
```

You can confirm that this is the correct points, goal difference and rankings by checking the table here.

11.4   Relegation Status

The last variable we will create is the relegation status. In the Eredivisie, the teams ranked 17th and 18th are automatically relegated to the lower "Keuken Kampioen" (Kitchen Champion) league (where Tilburg's Willem II competed that year). The 16th team enters into a playoff with teams in the Keuken Kampioen league. We will create a character variable with this information.

To do this we first create a variable which is blank everywhere: (`""`). We then fill in values depending on the rank of the team using indexing:

```
df$relegation_status <- ""
df$relegation_status[df$ranking < 16] <- "No relegation"
df$relegation_status[df$ranking == 16] <- "Relegation playoffs"
df$relegation_status[df$ranking == 17 |
                       df$ranking == 18] <- "Automatic relegation"
df[, c("team", "ranking", "relegation_status")]
```

```
                 team ranking    relegation_status
9           Feyenoord       1         No relegation
13                PSV       2         No relegation
2                Ajax       3         No relegation
1                  AZ       4         No relegation
6           FC Twente       5         No relegation
16  Sparta Rotterdam       6         No relegation
7          FC Utrecht       7         No relegation
18      sc Heerenveen       8         No relegation
14      RKC Waalwijk       9         No relegation
17            Vitesse      10         No relegation
11   Go Ahead Eagles      11         No relegation
12                NEC      12         No relegation
10   Fortuna Sittard      13         No relegation
8         FC Volendam      14         No relegation
3           Excelsior      15         No relegation
4           FC Emmen      16   Relegation playoffs
15         SC Cambuur      17  Automatic relegation
5       FC Groningen      18  Automatic relegation
```

For teams in rank 17 or 18, we use the logical OR operator: If the ranking is equal to 17 or equal to 18, we set the status to "Automatic relegation".

Writing `df$ranking == 17 | df$ranking == 18` inside the brackets is quite long. If you had more numbers you wanted to compare the ranking to, it would become a really long command. Fortunately R has a special operator we can use as a shortcut: the `%in%` operator. We can use the `%in%` operator to do the same thing as follows:

```
df$relegation_status[df$ranking %in% 17:18] <- "Automatic relegation"
```

What is the `%in%` doing? When we write `a %in% b` we are checking for each element in `a` if there is a *matching* element *somewhere* in `b`. To see this at work, consider the following example:

```
a <- 1:6
b <- c(3, 5, 7)
a %in% b
```

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

Here the 3rd and 5th element are `TRUE`, because the 3rd and 5th element of `a` (which are 3 and 5) are *somewhere* in `b` (3 and 5 are in `b`, but 1, 2, 4 and 6 aren't). An equivalent way of doing it (but with more typing) would be to see if $a = 3$ or $a = 5$ or $a = 7$ for each element (i.e. check for a match in any of the elements of `b`):

```r
a == b[1] | a == b[2] | a == b[3]
```

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

In the example above, we had `df$ranking %in% 17:18`. This returns `TRUE` if the team's ranking was one of 17 or 18 and is `FALSE` otherwise.

Chapter 12

Dataframes: Summary Statistics

In this chapter we will learn some techniques for summarizing a dataframe using the same running example.

We load up the data and create some of the missing variables again (summarizing what we did in the last chapter):

```
df <- read.csv(text = "
            team, wins, draws, losses, goals_for, goals_against
              AZ,   20,     7,      7,        68,            35
            Ajax,   20,     9,      5,        86,            38
        Excelsior,    9,     5,     20,        32,            71
        FC Emmen,    6,    10,     18,        33,            65
    FC Groningen,    4,     6,     24,        31,            75
       FC Twente,   18,    10,      6,        66,            27
      FC Utrecht,   15,     9,     10,        55,            50
     FC Volendam,   10,     6,     18,        42,            71
       Feyenoord,   25,     7,      2,        81,            30
 Fortuna Sittard,   10,     6,     18,        39,            62
 Go Ahead Eagles,   10,    10,     14,        46,            56
             NEC,    8,    15,     11,        42,            45
             PSV,   23,     6,      5,        89,            40
    RKC Waalwijk,   11,     8,     15,        50,            64
      SC Cambuur,    5,     4,     25,        26,            69
Sparta Rotterdam,   17,     8,      9,        60,            37
         Vitesse,   10,    10,     14,        45,            50
    sc Heerenveen,   12,    10,     12,        44,            50
```

```r
", strip.white = TRUE)

# Create goal difference:
df$goal_diff <- df$goals_for - df$goals_against

# Create total points scored over the season:
df$total_points <- 3 * df$wins + df$draws

# Order teams by season rank and create season ranking variable:
df <- df[order(df$total_points, df$goal_diff, decreasing = TRUE), ]
df$ranking <- 1:nrow(df)
```

12.1   `summary()` for Dataframes

To get a broad overview of a dataset, you can use the `summary()` function
that we used in Chapter 5 before for vectors. When we use this function on a
dataframe, it will show the summary statistics for all variables in the dataframe:

```r
summary(df)
```

```
     team                 wins             draws            losses
 Length:18           Min.   : 4.00   Min.   : 4.000   Min.   : 2.00
 Class :character    1st Qu.: 9.25   1st Qu.: 6.000   1st Qu.: 7.50
 Mode  :character    Median :10.50   Median : 8.000   Median :13.00
                     Mean   :12.94   Mean   : 8.111   Mean   :12.94
                     3rd Qu.:17.75   3rd Qu.:10.000   3rd Qu.:18.00
                     Max.   :25.00   Max.   :15.000   Max.   :25.00
   goals_for       goals_against      goal_diff       total_points
 Min.   :26.00    Min.   :27.00    Min.   :-44.0    Min.   :18.00
 1st Qu.:39.75    1st Qu.:38.50    1st Qu.:-27.5    1st Qu.:36.00
 Median :45.50    Median :50.00    Median : -5.5    Median :40.50
 Mean   :51.94    Mean   :51.94    Mean   :  0.0    Mean   :46.94
 3rd Qu.:64.50    3rd Qu.:64.75    3rd Qu.: 30.5    3rd Qu.:62.75
 Max.   :89.00    Max.   :75.00    Max.   : 51.0    Max.   :82.00
    ranking
 Min.   : 1.00
 1st Qu.: 5.25
 Median : 9.50
 Mean   : 9.50
 3rd Qu.:13.75
 Max.   :18.00
```

For the team name, it just says `character`, because we cannot find the mean
of a character. The only information we get is the number of observations (18).

All the other variables are numeric, and the summary statistics are shown for each one.

12.2 `head()` and `tail()`

Another way to get a broad overview of a dataset is to just "eyeball" it by displaying it in the console with `df`, or browsing it in RStudio with `View(df)`. For datasets with many observations, however, it may be easier to just look at the first few rows. We can do that with the `head()` function:

```
head(df)
```

```
                 team wins draws losses goals_for goals_against goal_diff
9           Feyenoord   25     7      2        81            30        51
13                PSV   23     6      5        89            40        49
2                Ajax   20     9      5        86            38        48
1                 AZ   20     7      7        68            35        33
6           FC Twente   18    10      6        66            27        39
16  Sparta Rotterdam   17     8      9        60            37        23
   total_points ranking
9            82       1
13           75       2
2            69       3
1            67       4
6            64       5
16           59       6
```

By default, `head()` shows the first 6 rows. We can look at a different number by specifying the option `n`. For example, to see the first 4 rows we would do:

```
head(df, n = 4)
```

```
        team wins draws losses goals_for goals_against goal_diff total_points
9  Feyenoord   25     7      2        81            30        51           82
13       PSV   23     6      5        89            40        49           75
2       Ajax   20     9      5        86            38        48           69
1         AZ   20     7      7        68            35        33           67
   ranking
9        1
13       2
2        3
1        4
```

The function `tail()` does the exact opposite. It shows the last `n` rows of the dataset, with 6 rows by default. To see the two teams that are automatically relegated (the bottom 2) we would do:

```
tail(df, n = 2)
```

```
           team wins draws losses goals_for goals_against goal_diff
15   SC Cambuur    5     4     25        26            69       -43
5  FC Groningen    4     6     24        31            75       -44
   total_points ranking
15           19      17
5            18      18
```

12.3 nrow() and ncol()

Something we are often interested in is the total number of observations. We can find this by checking the number of rows in the dataframe with the **nrow()** function. In this case it is the number of teams. The number of columns (found with **ncol()**) shows the total number of variables.

```
nrow(df)
```

```
[1] 18
```

```
ncol(df)
```

```
[1] 9
```

If we want to quickly find both of these numbers, we can also use the **dim()** function, which shows the dimensions of the dataframe (first the number of rows, then the number of columns):

```
dim(df)
```

```
[1] 18  9
```

12.4 names()

Sometimes we are just interested in what variables are included in the dataset. To see this, we can use the **names()** function:

```
names(df)
```

```
[1] "team"         "wins"          "draws"      "losses"
[5] "goals_for"    "goals_against" "goal_diff"  "total_points"
[9] "ranking"
```

Chapter 13

Data Cleaning

Often when we get a dataset we need to analyze, it's not always exactly structured and formatted in a way that we can immediately work with it. Here are some examples of this:

- The data don't start at the top of the file because the first few rows contain some other information.
- The dates are not formatted correctly.
- Numbers are interpreted as characters.
- The data contain extra columns that we don't want.
- There are rows with missing data that we want to omit.
- The variable names are not what we want them to be.

In these cases we need to *clean* the data before we can work with it. By cleaning we don't mean modifying the underlying data. It just means bringing the dataset into a format such that we can more easily work with it in R.

Some datasets are "dirtier" than others, sometimes so "dirty" that it can take weeks or even months to clean. Fortunately we will stick to "lightly unkempt" data for this course that can be cleaned only a few lines of code.

In this chapter we will learn some basic data cleaning techniques to deal with the 6 example issues listed above. We will do this using stock price data for the company ASML from the Amsterdam Stock Exchange.

Download the following file: asml-trades.csv and put it in your project folder. The variable names and meanings are:

- `Date`: The date the data from that row are from.
- `Open`: The opening price of the stock on that day.
- `High`: The highest price the stock traded at on that day.
- `Low`: The lowest price the stock traded at on that day.
- `Last`: The price of the last-traded stock at on that day.

- `Close`: The closing price of the stock on that day.
- `Number.of.Shares`: The number of shares traded that day.
- `Number.of.Trades`: The number of trades made that day.

13.1 Skipping Rows

Before reading the dataset into R, let's take a quick look at the text in the file itself. In the *Files* tab in RStudio, click on the `asml-trades.csv` file and click "View File". The text of the file opens in RStudio. The first few lines should look like:

```
Historical Data,,,,,,,,
From 2021-08-31 to 2023-08-30,,,,,,,,
NL0010273215,,,,,,,,
Date,Open,High,Low,Last,Close,Number of Shares,Number of Trades,Print table
31/8/2021,713.8,717.7,699.9,704.5,704.5,978555,36599,
1/9/2021,712.4,715.5,706.4,714.2,714.2,674506,38527,
2/9/2021,713.9,723.4,708.4,722.2,722.2,533840,29229,
```

We immediately notice that the first 3 rows contain information about the data that we don't want to include in our dataframe. The variable names are on line 4 instead of line 1.

One option would be to delete those rows and save the file. However, it is bad practice to directly edit a data file. One should always work with the raw CSV file as it was downloaded. This makes it easier to reproduce your work and show your steps through your R script. Fortunately the `read.csv()` function has an option to skip rows. We can use this with:

```r
df <- read.csv("asml-trades.csv", skip = 3)
```

We could also have read in the data directly from the URL with:

```r
df <- read.csv("https://walshc.github.io/pqs/asml-trades.csv", skip = 3)
```

This way we wouldn't have to set up an RStudio project or change the working directory.

Let's take a first look at the data with `summary()`:

```r
summary(df)
```

```
    Date                 Open            High            Low
 Length:521         Min.   :394.7   Min.   :408.2   Min.   :375.8
 Class :character   1st Qu.:535.5   1st Qu.:545.5   1st Qu.:525.8
 Mode  :character   Median :592.1   Median :597.4   Median :582.5
```

```
                      Mean   :589.2   Mean   :597.5   Mean   :579.6
                      3rd Qu.:645.5   3rd Qu.:652.5   3rd Qu.:636.1
                      Max.   :770.5   Max.   :777.5   Max.   :764.2
                      NA's   :6       NA's   :6       NA's   :6
      Last              Close       Number.of.Shares   Number.of.Trades
 Min.   :397.4   Min.   :397.4   Length:521         Length:521
 1st Qu.:535.9   1st Qu.:535.9   Class :character   Class :character
 Median :589.4   Median :589.4   Mode  :character   Mode  :character
 Mean   :588.4   Mean   :588.4
 3rd Qu.:644.0   3rd Qu.:644.0
 Max.   :770.5   Max.   :770.5
 NA's   :6       NA's   :6
 Print.table
 Mode:logical
 NA's:521
```

From this we can see a few problems:

- From the summary of the `Date` variable, we can see that R read it in as a character. It did not recognize that it was a date.
- The variables `Open`, `High`, `Low`, `Last` and `Close` contain 6 NAs. `NA` stands for "Not Available" and is what R uses to represent missing values.
- For the variables `Number.of.Shares` and `Number.of.Trades`, we can see that they were read in as characters instead of numbers.
- `Print.table` has 521 `NA`s (all values are `NA`), thus this variable is useless and should be deleted.
- We also want to change some of the variable names and also change the names to lower case and replace the dots (`.`) with underscores (`_`).

We will work through these problems for the rest of this chapter.

13.2   Formatting Dates

13.2.1   Converting Dates in the ASML Example

We will start by converting the date variable from a character to a date variable. This is useful for doing operations with the date (such as subsetting the data on observations before/after a particular date) and for plotting.

We can convert the character to a date using the `as.Date()` function. The first argument of the function is the vector of dates that need to be converted, and the `format` argument specifies the format the date is written in. Let's see what

format the dates are written in using the `head()` command to see the first few rows:

```r
head(df$Date)
```

```
[1] "31/8/2021" "1/9/2021"  "2/9/2021"  "3/9/2021"  "6/9/2021"  "7/9/2021"
```

When we look at the data, we can see that the format is dd/mm/yyyy. To specify this we need to write `format = "%d/%m/%Y"`.

```r
df$Date <- as.Date(df$Date, format = "%d/%m/%Y")
```

We can check what this did with the `head()` command again:

```r
head(df$Date)
```

```
[1] "2021-08-31" "2021-09-01" "2021-09-02" "2021-09-03" "2021-09-06"
[6] "2021-09-07"
```

We can see that when R has dates correctly formatted they show up in the format yyyy-mm-dd. This is the default format.

Now that the date is formatted correctly, the `summary()` command shows the first and last dates in the data (31 August 2021 and 29 August 2023):

```r
summary(df$Date)
```

```
        Min.      1st Qu.       Median         Mean      3rd Qu.         Max.
  "2021-08-31" "2022-03-01" "2022-08-30" "2022-08-29" "2023-02-28" "2023-08-29"
```

13.2.2 Converting Dates from Other Formats

Dates in the Netherlands are typically written like dd-mm-yyyy. If it was the case we would instead do `"%d-%m-%Y"`. You can try this out with:

```r
as.Date("1-9-2023", format = "%d-%m-%Y")
```

```
[1] "2023-09-01"
```

If the dates were in mm/dd/yyyy format, as is typical in USA, we would do `"%m/%d/%Y"`.

```r
as.Date("12/31/2022", format = "%m/%d/%Y")
```

```
[1] "2022-12-31"
```

Sometimes we omit the century from years. We might write the 1st of September 2023 as 01/09/23. In R we need to use `%y` instead of `%Y` for these abbreviated years:[1]

```
as.Date("01/09/22", format = "%d/%m/%y")
```

```
[1] "2022-09-01"
```

13.2.3 Converting Dates with Month Names (Optional)

Sometimes the dates have the month name in words. To convert this we need to use the `%b` option for abbreviated month names and `%B` for full month names:

```
as.Date("Sep 1 2023", format = "%b %d %Y")
```

```
[1] "2023-09-01"
```

```
as.Date("January 1 2023", format = "%B %d %Y")
```

```
[1] "2023-01-01"
```

Note that if your computer is not in English it might not work as R expects to read months in the language of your computer. To get around this, you can first set the language for dates to English using the `Sys.setlocale()` function before formatting the dates. Because of the complications with month names in different languages, I will not ask questions on the assignments or exam involving this. I am just providing this information in case it may be useful for you later.

13.3 Converting Characters to Numbers

For the variables `Number.of.Shares` and `Number.of.Trades`, we saw that they were read in as characters instead of numbers. Usually this happens when there are some letters or other non-numeric symbols (like the % symbol) somewhere in the data. This is because all elements of vectors in R (the individual columns of a `data.frame`) must have the same data type. If there are any character elements in a vector, the remaining elements are coerced into characters.

If we look through the data we can see that some rows have `"None"` written instead of `NA`.[2]

---

[1]If the date is `01/01/69`, the format `%d/%m/%y` will interpret it as January 1 **1969**. But if the date is `01/01/68`, it will interpret it as January 1 **2068**. All short-format years after 69 are put in the 1900s and all short-format years before 69 are put in the 2000s. You don't need to remember these details for the exam though because we won't ever use dates outside of 1969-2068.

[2]As a shortcut, we could have found all the non-numeric characters in a variable using the

To do this conversion we first replace the `"None"` values with `NA`. We do this by assigning `NA` to the subset of values where `df$Number.of.Shares == "None"`:[3]

```
df$Number.of.Shares[df$Number.of.Shares == "None"] <- NA
```

Once we have done this we use the `as.numeric()` function to convert the values from `character` to `numeric`:

```
df$Number.of.Shares <- as.numeric(df$Number.of.Shares)
```

Now when we summarize we see that it's treated as a number:

```
summary(df$Number.of.Shares)
```

```
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
 84141  571178  711067  783291  916000 2932273       6
```

We can do the same with the `Number.of.Trades`. We are also able to do this skipping the step of converting the `"None"` to `NA`:

```
df$Number.of.Trades <- as.numeric(df$Number.of.Trades)
```

```
Warning: NAs introduced by coercion
```

When we do this, however, we see that R warned us that it converted some observations to `NA`. A warning is different to an error in that R still completes the operation (in an error it will just stop). But it is warning us because we may not have expected some values to be forced to `NA`. In general it is better to code in a way that doesn't generate warnings, so I recommend setting the non-numeric values to `NA` first.

13.4 Deleting columns

The last column of the data, `Print.table`, contains no data. It has `NA` for all rows.

To delete a variable we write over the variable with `NULL`. This essentially replaces it with nothing:

```
df$Print.table <- NULL
```

---

following command: `unique(grep("[^0-9]", df$Number.of.Trades, value = TRUE))`. You don't need to remember this command for the exam.

[3]We could also have replaced all elements with non-numeric characters without knowing what they are with the following: `df$Number.of.Shares[grepl("[^0-9]", df$Number.of.Trades)] <- NA`. You don't need to remember this command for the exam.

An alternative approach would be to use the column index of the variable we want to drop. `Print.table` is the 9th column in the data, and to drop the 9th column we could use:

```r
df <- df[, -9]
```

If we view the data in RStudio, we can see that it is now deleted.

13.5 Dropping rows with missing data

Let's take a look at how our data look now:

```r
summary(df)
```

```
     Date                  Open            High             Low
 Min.   :2021-08-31   Min.   :394.7   Min.   :408.2   Min.   :375.8
 1st Qu.:2022-03-01   1st Qu.:535.5   1st Qu.:545.5   1st Qu.:525.8
 Median :2022-08-30   Median :592.1   Median :597.4   Median :582.5
 Mean   :2022-08-29   Mean   :589.2   Mean   :597.5   Mean   :579.6
 3rd Qu.:2023-02-28   3rd Qu.:645.5   3rd Qu.:652.5   3rd Qu.:636.1
 Max.   :2023-08-29   Max.   :770.5   Max.   :777.5   Max.   :764.2
                      NA's   :6       NA's   :6       NA's   :6
     Last            Close        Number.of.Shares  Number.of.Trades
 Min.   :397.4   Min.   :397.4   Min.   :  84141   Min.   :  4398
 1st Qu.:535.9   1st Qu.:535.9   1st Qu.: 571178   1st Qu.: 26290
 Median :589.4   Median :589.4   Median : 711067   Median : 33375
 Mean   :588.4   Mean   :588.4   Mean   : 783291   Mean   : 35830
 3rd Qu.:644.0   3rd Qu.:644.0   3rd Qu.: 916000   3rd Qu.: 42707
 Max.   :770.5   Max.   :770.5   Max.   :2932273   Max.   :108957
 NA's   :6       NA's   :6       NA's   :6         NA's   :6
```

We can see that there are 6 `NA`s for all variables except the `Date` variable. If we scroll through the data we notice that 6 rows with `NA`s are the same for all variables. Let's have a look at what dates these are. We can do this using the `is.na()` function. This function, when applied to a vector, returns `TRUE` if the element is `NA` and `FALSE` if not. To see the dates when the variables are missing, we can do:

```r
df$Date[is.na(df$Open)]
```

```
[1] "2022-04-15" "2022-04-18" "2022-12-26" "2023-04-07" "2023-04-10"
[6] "2023-05-01"
```

We can see that the missings were:

- 2022-04-15: Good Friday

- 2022-04-18: Easter Monday (*Tweede paasdag*)
- 2022-12-26: Day after Christmas (*Tweede kerstdag*)
- 2023-04-07: Good Friday
- 2023-04-10: Easter Monday (*Tweede paasdag*)
- 2023-05-01: Labor Day (*Dag van de Arbeid*)

These are weekdays where the Amsterdam stock market is closed. We can drop rows with any missing values using the `na.omit()` function. I will print the number of rows in `df` before and after the operation to show what is happening:

```
nrow(df)
```

```
[1] 521
```

```
df <- na.omit(df)
nrow(df)
```

```
[1] 515
```

We can see that we fell from 521 observations to 515 after deleting the 6 holidays from the data.

13.6   Renaming Variables

Although the variable names are quite okay, suppose we wanted to change some of them.

Suppose we wanted to change the name of `"Number.of.Shares"` to `"num_shares"` to make it shorter to type, and to replace the dot with an underscore. Because we know it occupies the 7th column, we can change the name with:

```
names(df)[7] <- "num_shares"
```

Suppose we also wanted to change the name of `"Number.of.Trades"` to `"num_trades"`. Counting columns like we did above increases our chances of making a mistake (besides, we need to do a lot of counting). We can instead change the name using the old name as follows:

```
names(df)[names(df) == "Number.of.Trades"] <- "num_trades"
```

How this works is `names(df) == "Number.of.Trades"` is `TRUE` only in the 8th column when the name is actually `"Number.of.Trades"`, and so it changes the name of only that column.

We can also change the names of multiple columns at the same time. Suppose we wanted to change Open, High, Low and Last to lower case. We can do:

```
names(df)[2:5] <- c("open", "high", "low", "last")
```

If we wanted to quickly change all variable names to lower case, we can use the `tolower()` function. The `tolower()` function converts upper case characters to lower case:

```
test <- c("hello!", "HELLO!", "Hello!", "HeLlO!")
tolower(test)
```

```
[1] "hello!" "hello!" "hello!" "hello!"
```

Let's use it to change the names of the data set:

```
names(df) <- tolower(names(df))
names(df)
```

```
[1] "date"       "open"        "high"       "low"        "last"
[6] "close"      "num_shares" "num_trades"
```

Chapter 14

Introduction to Plotting

14.1 Introduction

We will now learn some techniques to *visualize* your data. We will learn how to create histograms, bar charts, line plots, scatter plots, among others, and how to customize them.

Base R (R without any packages) has some basic plotting functions. These are easy to use but they are not easily customizable and don't look very elegant. For that reason we will also learn how to use the popular plotting package `ggplot2`. But in this chapter we stick to base R, leaving `ggplot` for later chapters.

14.2 Example Setting: Penguins

To get started on some basic plotting techniques we will use the famous "Palmer Penguins" dataset. This dataset contains several measurements of different penguins collected by researchers on Antwerp Island in the Palmer Archipelago of Antarctica. Interestingly, there is a smaller island next to this called Brabant Island.

The dataset contains data from three species of penguins: the Adelie, Chinstrap and Gentoo. A picture of each species is shown in the pictures below:

This dataset is convenient to use because we can load it into R straight from a package. First install the package with the dataset with:

```
install.packages("palmerpenguins")
```

Then load it with:

```r
library(palmerpenguins)
data(penguins)
```

Running the command `data(penguins)` loads up two datasets: `penguins` and `penguins_raw`. We will ignore the `penguins_raw` dataset and only work with the `penguins` one.

14.3   Data Inspection

Before getting started with plotting, it's good to first get a basic understanding of our data. Let's get some summary statistics with `summary()` and find out how many observations with have with `nrow()`:

```r
summary(penguins)
```

```
     species            island    bill_length_mm  bill_depth_mm
 Adelie   :152   Biscoe   :168   Min.   :32.10   Min.   :13.10
 Chinstrap: 68   Dream    :124   1st Qu.:39.23   1st Qu.:15.60
 Gentoo   :124   Torgersen: 52   Median :44.45   Median :17.30
                                 Mean   :43.92   Mean   :17.15
                                 3rd Qu.:48.50   3rd Qu.:18.70
                                 Max.   :59.60   Max.   :21.50
                                 NA's   :2       NA's   :2
 flipper_length_mm  body_mass_g       sex          year
 Min.   :172.0     Min.   :2700   female:165   Min.   :2007
 1st Qu.:190.0     1st Qu.:3550   male  :168   1st Qu.:2007
 Median :197.0     Median :4050   NA's  : 11   Median :2008
 Mean   :200.9     Mean   :4202                Mean   :2008
 3rd Qu.:213.0     3rd Qu.:4750                3rd Qu.:2009
 Max.   :231.0     Max.   :6300                Max.   :2009
 NA's   :2         NA's   :2
```

```r
nrow(penguins)
```

```
[1] 344
```

We see that we have data on 344 penguins with the following variables:

- `species`: A factor variable indicating which of the 3 species the penguin is.
- `island`: A factor variable indicating which island the penguin was on.
- `bill_length_mm`: A numerical variable indicating how long the penguin's bill (their beak) was (in mm).
- `bill_depth_mm`: A numerical variable indicating how deep the penguin's bill was (in mm). The depth is the distance between the top and bottom

of their beak.
- `flipper_length_mm`: A numerical variable indicating how long their flipper (wing) is (in mm).
- `body_mass_g`: A numerical variable indicating how heavy the penguin is (in grams).
- `sex`: A factor variable indicating the gender of the penguins (`male` or `female`).
- `year`: A numerical variable indicating what year the data point is from.

We also see that we have 2 missing values for 4 of the variables and 11 missing values for `sex`. For our purposes here it is fine to just leave these missing values in the dataset. We don't need to delete those rows.

14.4   Basic Plotting with Base R

We will now learn how to do some very simple plots with base R: the histogram, the bar plot and the scatter plot. The plots from base R are not very beautiful, but the idea is to learn how to make "quick and dirty" plots for you to quickly get a sense of your data, before making nicer customizable plots with `ggplot`.

14.4.1   Histograms

To describe the distribution of a single numeric variable, we can use a histogram. A histogram splits the data into "bins" and shows the number of observations in each bin. We can create a histogram by using the `hist()` function, putting the variable we want to plot as the argument inside:

```
hist(penguins$body_mass_g)
```

## Histogram of penguins$body_mass_g
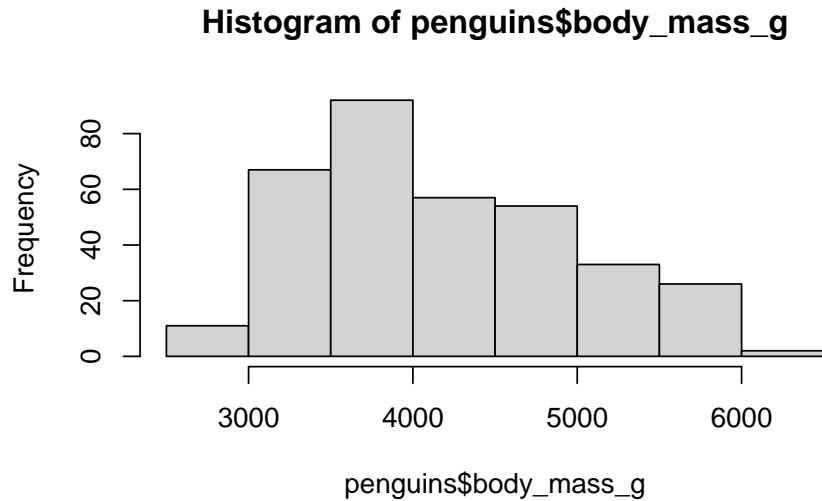
14.4.2   Bar Plot

For categorical variables, we can use a bar plot to visualize the relative frequencies of different categories. We already saw the `table()` function which counts the number of times each category appears:

```
table(penguins$species)
```

```
   Adelie Chinstrap    Gentoo
      152        68       124
```

If we want to plot these values, we can put this entire expression into the `barplot()` function:

```
barplot(table(penguins$species))
```
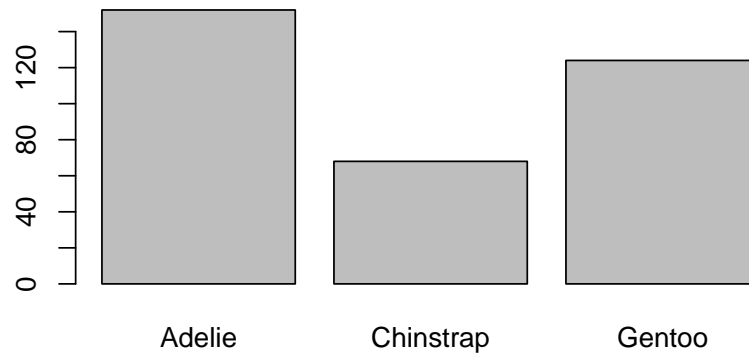
14.4.3 Scatter Plots

To quickly visualize the relationship between two variables we can make a scatter plot. We can do this by listing the two variables we want to plot as arguments in the `plot()` function:

```
plot(penguins$bill_length_mm, penguins$flipper_length_mm)
```
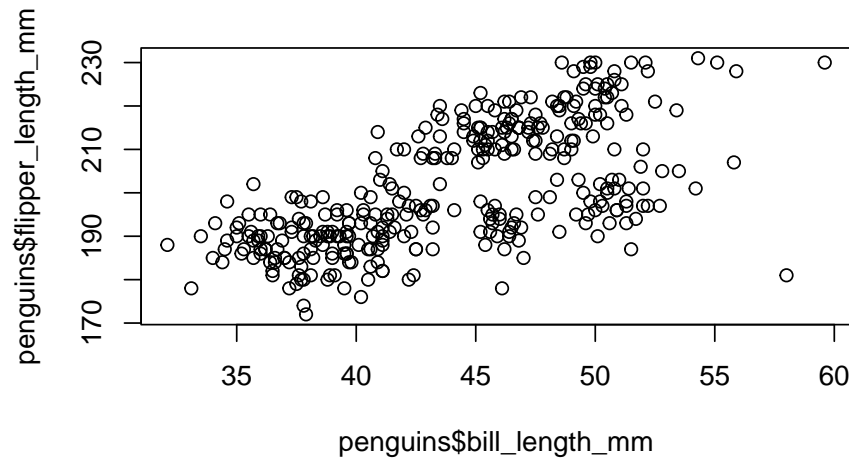
In each case, the base R commands to make plots are very short and easy to use. Therefore I use them very frequently in the console to learn what a dataset looks like. But because they do not look very nice I do not tend to use them in research papers. I prefer to use the plots from `ggplot`, which we will learn about next.

Chapter 15

Data Visualization with `ggplot`

15.1 Introduction

`ggplot` is the main data visualization package in R. The `gg` in the name refers to "Grammar of Graphics" which is a scheme of layering different parts of a plot. As we will learn, `ggplot()` works by adding layers.

To get started, we first need to install and load the `ggplot2` package. The name of the package is `ggplot2` (with a 2), but the function we use to make the plots is just `ggplot` (without a 2).

```
install.packages("ggplot2")
library(ggplot2)
```

We also load the same penguins data that we saw in Chapter 14:

```
library(palmerpenguins)
data(penguins)
```

We will begin by recreating the plots we saw in Chapter 14 and customizing them to our liking.

15.2   Histograms

15.2.1   Basic Histogram

Let's first show how to make a basic histogram (without customization) and then describe the different parts of the function:

```
ggplot(penguins, aes(body_mass_g)) +
  geom_histogram()
```

`` `stat_bin()` using `bins = 30`. Pick better value with `binwidth`. ``

Warning: Removed 2 rows containing non-finite outside the scale range (`stat_bin()`).
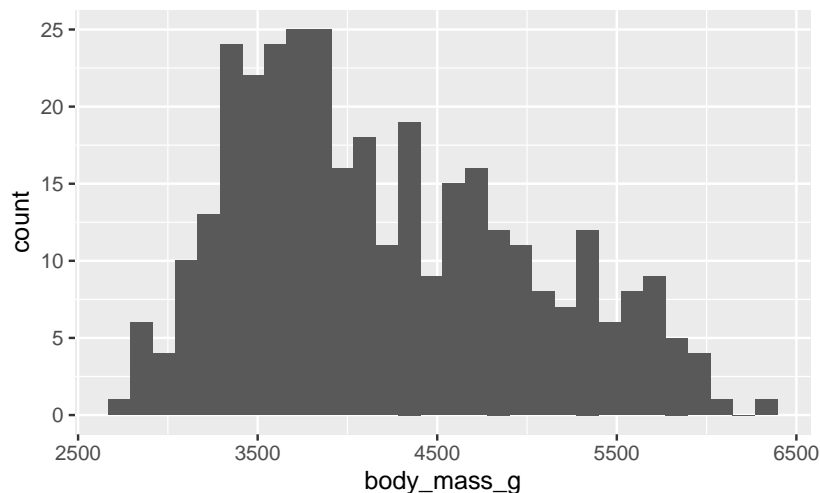
With `ggplot()`, the first argument you provide is the dataframe with the variables you want to plot. Here it's the `penguins` dataframe. The second argument is the "mapping" which we provide using the `aes()` (aesthetics) function. For a histogram we only need to provide one variable. Because we already tell ggplot the name of the dataframe, we only provide the variable name `body_mass_g` here instead of typing `penguins$body_mass_g`.

The next thing we do is add *layers*. We do this using the sum operator, `+`. To tell ggplot to add a histogram to the plot, we add `geom_histogram()`. It's good practice to put the layers on different lines to make them easier to read, as later on we'll see that `ggplot` commands can get very long as we add many many layers.

We can see that ggplot gave 2 warnings when we did this command:

1. It tells us it is using 30 bins and tells us how we can change this.
2. `Warning: Removed 2 rows containing non-finite values` is telling us that our data contain 2 `NA` values that weren't included in the plot. This is expected, because in Chapter 14 we saw that our dataset contains some missing observations.

15.2.2 Customizing a Histogram

To customize the plot, we can use options in the functions, and add more *layers* to the plot. There are many possibilities here. Let's see some of these:

15.2.2.1 Changing the number of bins:

Just like the warning above told us, we can change the number of bins to a different number using the `bins` option in `geom_histogram()`. Let's change it to use 15 bins instead:

```
ggplot(penguins, aes(body_mass_g)) +
  geom_histogram(bins = 15)
```

Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
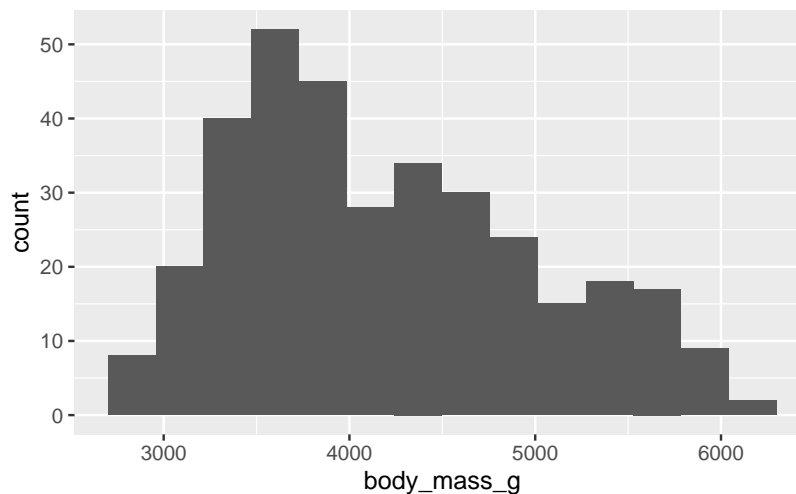
15.2.2.2 Changing the color of the bins:

We can change the color of the bins using the `fill` option in `geom_histogram()`, and we can change the outline color of the bins using the `color` option. We add this to the previous options:

```
ggplot(penguins, aes(body_mass_g)) +
  geom_histogram(bins = 15, fill = "navy", color = "white")
```

Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
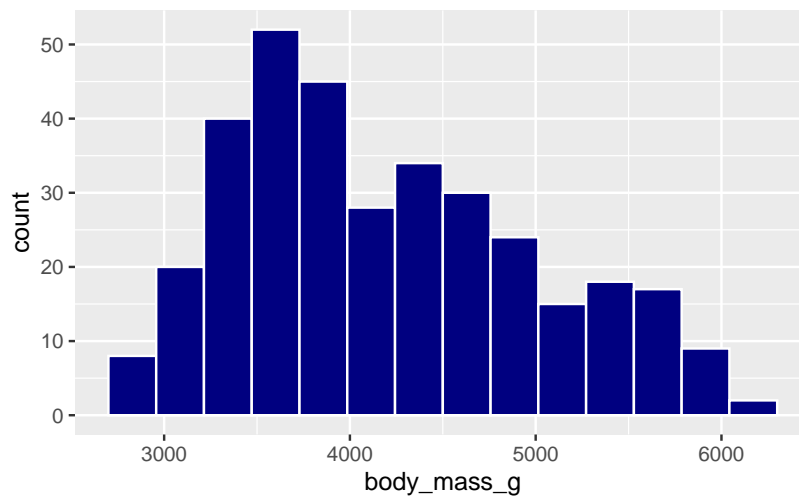
15.2.2.3   Changing the axis labels:

To change the names of the axis labels, we add *additional layers*. We can change
the horizontal axis label with the `xlab()` function, putting in quotes what we
want the new label to be. Similarly, we use the `ylab()` function for the vertical
axis label. Because these are additional layers, we add them to the plot using
the `+` operator:

```
ggplot(penguins, aes(body_mass_g)) +
  geom_histogram(bins = 15, fill = "navy", color = "white") +
  xlab("Penguin weight (grams)") +
  ylab("Count")
```

Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
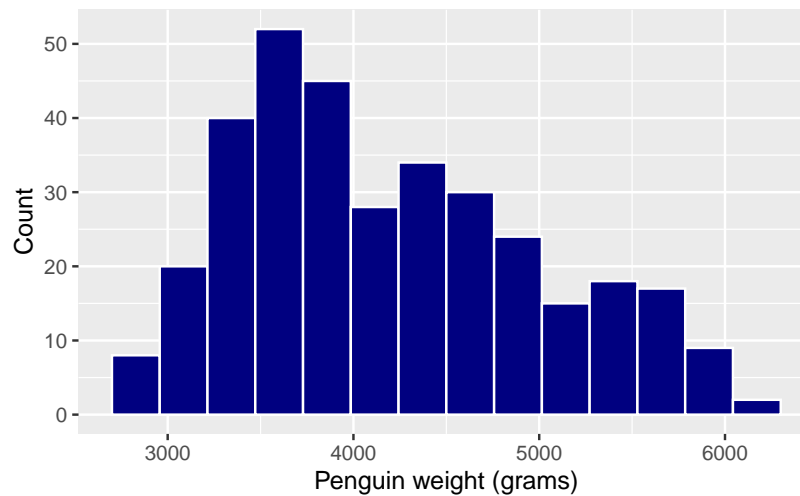
15.2.2.4 Changing the plot theme:

What if we want to get rid of the gray background? A white background is better for plots that get printed on paper. The easiest way to do this is to change the "theme" of the plot to a more minimalistic theme. We do this by adding the `theme_minimal()` layer:

```
ggplot(penguins, aes(body_mass_g)) +
  geom_histogram(bins = 15, fill = "navy", color = "white") +
  xlab("Penguin weight (grams)") +
  ylab("Count") +
  theme_minimal()
```
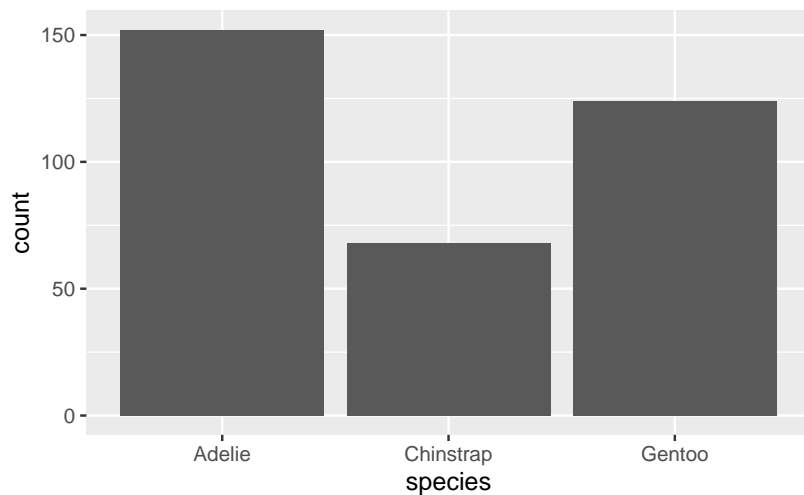
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
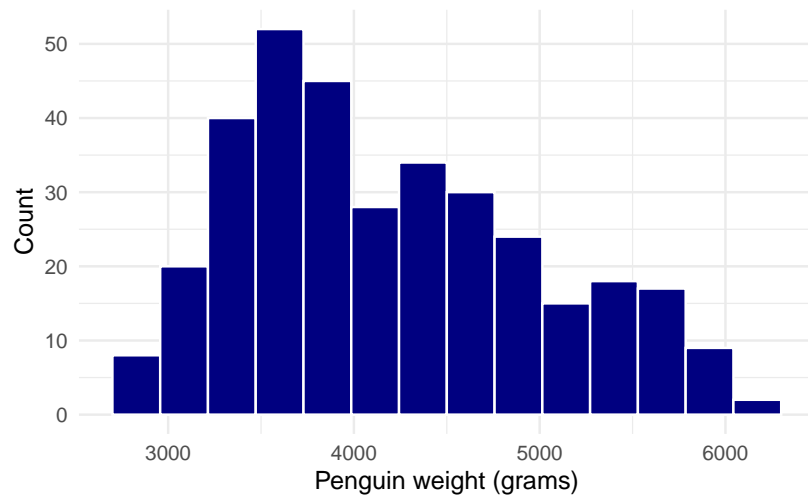
15.3   Bar Plots

We can create a bar plot in a very similar way. We just use the categorical variable in place of the numeric one and use `geom_bar()` as the additional layers instead:
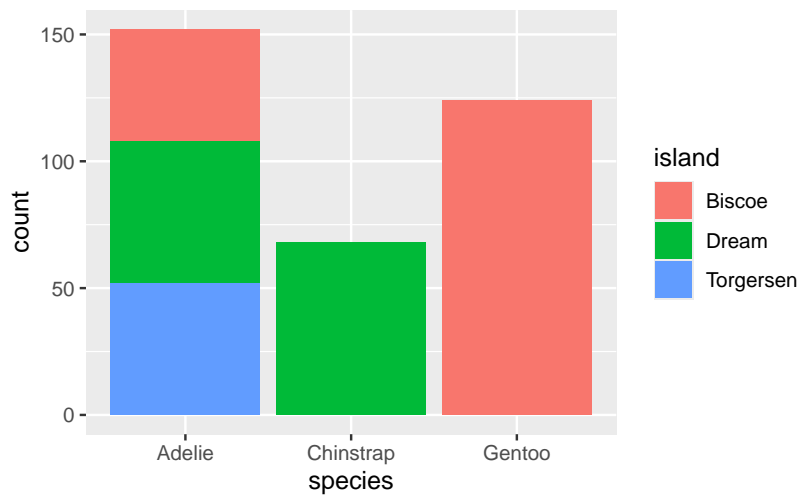
```
ggplot(penguins, aes(species)) +
  geom_bar()
```

We can also get a bar plot of the number of species on each island, which displays

even more information:

```
ggplot(penguins, aes(species, fill = island)) +
  geom_bar()
```


Here we can see that the Adelie is found on all 3 islands, but the Chinstrap is only on "Dream" island and the Gentoo is only on "Torgersen" island.

We can add layers of customization to this in a similar way. This time we will add all of the customization in one go. Here is one way to do it:

```
ggplot(penguins, aes(species, fill = island)) +
  geom_bar(color = "black") +
  xlab("Penguin species") +
  ylab("Count") +
  scale_fill_discrete(name = "Island",
                      type = c("#0B0405", "#357BA2", "#DEF5E5")) +
  theme_minimal()
```
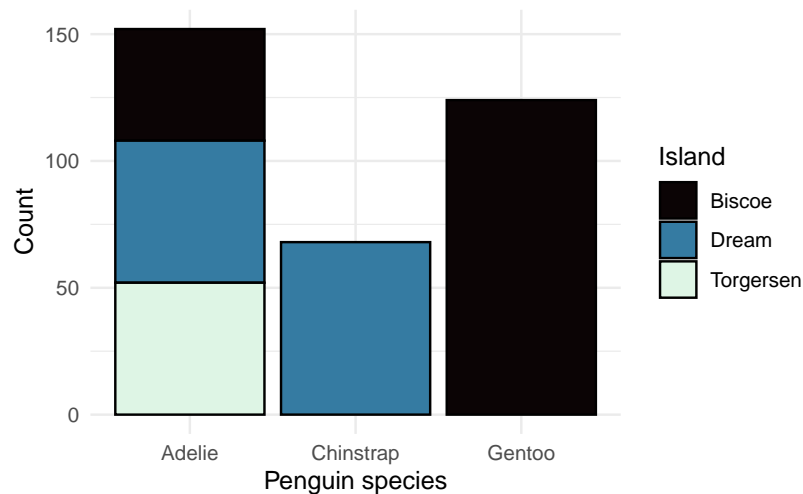
The `color` option in `geom_bar()` is for the outline of the bars (`fill` is for the fill color on the inside).

We can customize the colors and the legend name using the layer `scale_fill_discrete()`. This is the name of the function because this legend is for the "fill" variable, which is "discrete" because it's a factor (as opposed to a numerical variable which would be continuous). We provide the legend name with the `name` option and we provide the colors using the `type` option. Here I've provided the colors using the *hexidecimal format* as opposed to their names like we did earlier with "navy". This format allows you to choose exactly what shade you like. You can find the hex code for any color with many tools online. Google even has one built in if you search for "color picker". I have chosen these colors because the differences would still be clear even if you printed out the plot in black and white.

15.4 Scatter Plots

For scatter plots we need to provide both the x variable and the y variable in the `aes()` command. Let's plot the bill length against the flipper length like we did with base R:

```
ggplot(penguins, aes(bill_length_mm, flipper_length_mm)) +
  geom_point()
```

Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
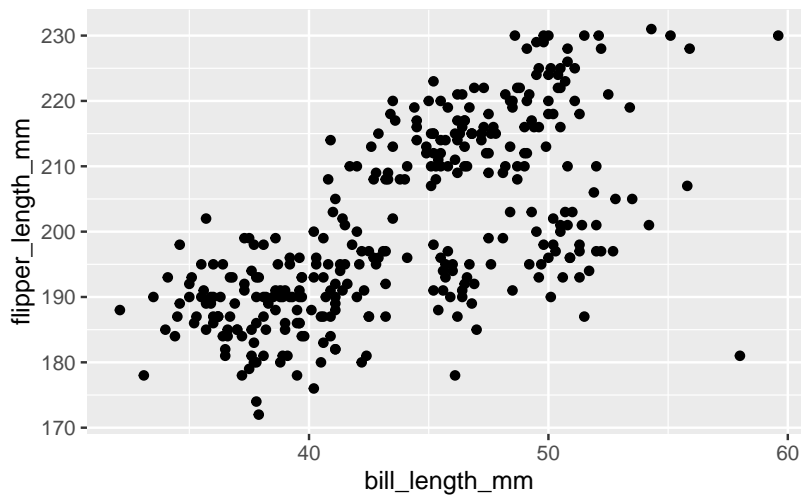
If we want to have the colors of the dots to change with the species, we can specify that with `color` in `aes()` as well:

```
ggplot(penguins, aes(bill_length_mm, flipper_length_mm, color = species)) +
  geom_point()
```

Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
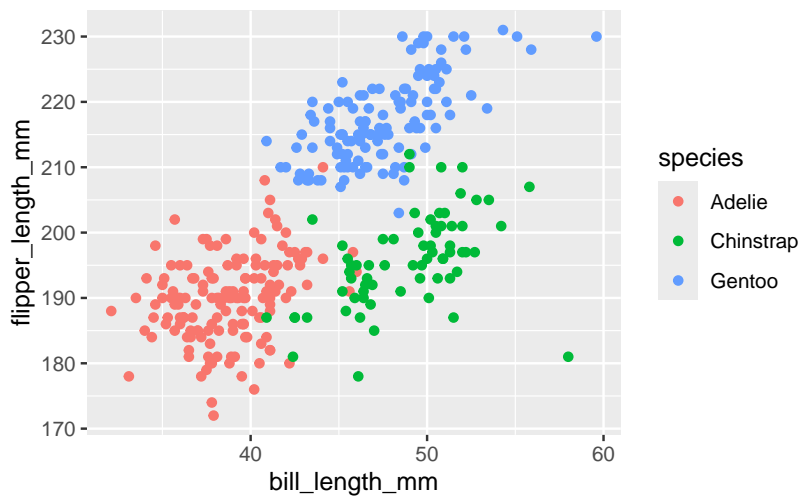
We can then add some more customization in the same way as before:

```r
ggplot(penguins, aes(bill_length_mm, flipper_length_mm, color = species)) +
  geom_point() +
  scale_color_discrete(name = "Species") +
  xlab("Bill length (in mm)") +
  ylab("Flipper length (in mm)") +
  theme_minimal()
```

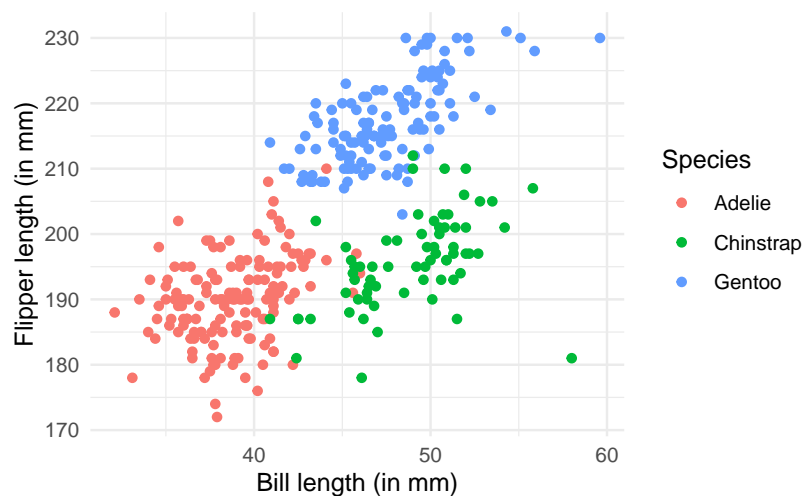Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).

15.5   Saving Plots

There are many different ways to save plots created by R, but a simple way to do so is to use the `ggsave()` function. If we want to save a plot as a PDF file we simply use the command after our `ggplot()` call giving the name we want to give to the plot with the appropriate file extension (such as `.pdf` or `.png`). For example:

```r
ggplot(penguins, aes(bill_length_mm, flipper_length_mm)) +
  geom_point()
ggsave("my-plot.pdf")
```

The plot is saved in the current working directory - the folder given by the `getwd()` command.

Chapter 16

Making Functions

16.1 Creating Simple Functions

It is very easy to make your own customized functions in R. Suppose, for example, you want to make an R function to calculate the output of the quadratic function:

$$f(x) = -8 - 2x + x^2$$

If we want to call this function `f()`, we would define it as follows:

```r
f <- function(x) {
  y <- -8 - 2 * x + x^2
  return(y)
}
```

The `f` is what we want to call the function. We assign to `f` using the assignment operator, `<-`, the "function" with a single argument `x` using `function(x)`. After that we specify what the function is supposed to do:

- Calculate `y <- -8 - 2 * x + x^2`
- Return `y` as the output. We use the `return()` function to specify what the output of the function is.

We need to wrap what the function does in curly brackets (`{ }`) because what the function does can span several lines. Using the curly brackets tells R that these commands belong together in the function.

Let's try out the function:

```r
f(2)
```

```
[1] -8
```

```
  f(3)
```

```
[1] -5
```

We can also pass a vector into the function to see the output for several values at once:

```
  f(c(2, 3, 4))
```

```
[1] -8 -5  0
```

16.2   Plotting Functions

We can also use `ggplot()` to plot the function. To do this we first create a sequence of values of `x` over the range that we want to see the function plotted. We then evaluate the function for each of these values of `x` and save it as `y`. We then combine `x` and `y` into a data.frame and plot it like we learned in Chapter 14.

Let's give it a try:

```
library(ggplot2)
x <- seq(from = -4, to = 6, length.out = 200)
y <- f(x)
df <- data.frame(x, y)
ggplot(df, aes(x, y)) + geom_line()
```

In this example, the sequence runs from -4 to $+6$. The `length.out` option specifies how many numbers in total there should be in the sequence between -4 and $+6$. 200 numbers is plenty to get a curve that looks smooth. Why did we use -4 and $+6$ here? This range includes minimum and gives a good idea of its shape. You can try out different ranges instead (using numbers different from -4 and $+6$). When making these plots the easiest thing to do is try out different numbers until the plot looks good.

Chapter 17

Univariate Unconstrained Optimization

In Chapter 16 we learned how to make our own functions. We learned how to write a function to calculate:

$$f(x) = -8 - 2x + x^2$$

The function was:

```
f <- function(x) {
  y <- -8 - 2 * x + x^2
  return(y)
}
```

In this chapter we will learn how to find the extreme point (maximum/minimum) of this univariate function (function with only one variable).

17.1 Plotting Approach

In Chapter 16, we also learned how to plot the function with `ggplot()`. We can get a visual view of the extreme point:

```
library(ggplot2)
x <- seq(from = -4, to = 6, length.out = 200)
df <- data.frame(x, y = f(x))
ggplot(df, aes(x, y)) +
  geom_line()
```

From the plot we can see the following that the function achieves a minimum at $x = 1$.

17.2   Analytic Solution

We could have found this number analytically using calculus. Let's do that before doing it in R. The first derivative of the function is:

$$f'(x) = -2 + 2x$$

To find the extreme point of the function we find the value of x where $f'(x) = 0$. This happens when:

$$-2 + 2x = 0$$

Solving for x yields $x = 1$. To see if this is a maximum or a minimum we check the second derivative:

$$f''(x) = +2$$

This is positive, so we know it is a minimum. A minimum at $x = 1$ is exactly what we see in the plot.

17.3   Using Optimization

We will now use R to find the extreme point using optimization. We can use the `optimize()` function to find the minimum of a univariate function in R. To do that we need to first specify the function we want to minimize and an interval

to search over. We specify the interval as a vector with two elements, the lower
bound and the upper bound. We will use a wide interval of $[-100, +100]$. We
also need to specify if we are looking for a maximum or a minimum. We do that
with the `maximum` option and set it to `FALSE` when looking for a minimum:

```
optimize(f, interval = c(-100, 100), maximum = FALSE)
```

```
$minimum
[1] 1
```

```
$objective
[1] -9
```

We can see that we get the same result as the plot and the analytic solution.
The minimum value occurs at $x = 1$ and the value of the function is -9 at that
point.

If you want to *maximize* a function instead, we need to set `maximum = TRUE`.

The `optimize()` function returns a named `list`. Suppose we assign the output
of the `optimize()` function to `f_min`:

```
f_min <- optimize(f, interval = c(-100, 100), maximum = FALSE)
class(f_min)
```

```
[1] "list"
```

To extract the minimum from this list we can use `f_min$minimum`. The `$` works
for extraction with named lists the same way as with dataframes. To extract
the value of the function at the minimum, we can use `f_min$objective`:

```
f_min$minimum
```

```
[1] 1
```

```
f_min$objective
```

```
[1] -9
```

Chapter 18

Conditional Statements

18.1 If-else statements

Conditional statements, or "If-else statements" are very useful and extremely common in programming. In an if-else statement, the code first checks a particular true/false condition. If the condition is true, it performs one action, and if the condition is false, it performs another action.

A simple example of this is the absolute value function we saw in Chapter 3. Let's define precisely what that function does:

$$|x| = \begin{cases} -x & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

If $x < 0$, it returns $-x$ (so that the number becomes positive). Otherwise, it returns just x: if x was positive it stays positive, and if x is zero it stays zero.

Although there already is an absolute value function in R that we saw in Chapter 3 (the `abs()` function), we can easily create our own function to do the same thing.

Let's call this function `my_abs()` (my absolute value function):

```
my_abs <- function(x) {
  if (x < 0) {
    return(-x)
  } else {
    return(x)
  }
}
```

After `if`, we need to write the condition to check in parentheses (here `x < 0`). Then we write between the curly brackets (`{` and `}`) what we want R to do if the condition is `TRUE` (here return `-x`). Then we write `else` and write between the curly brackets what we want R to do if the condition is `FALSE` (here return `x`).

Let's go through what R does here given an input `x`. First R checks the condition `x < 0`. If it is `TRUE` it returns $-x$ and it's done. If it is `FALSE` it goes to the `else` and returns x.

Let's test it out:

```
my_abs(-2)
```

```
[1] 2
```

```
my_abs(3)
```

```
[1] 3
```

```
my_abs(0)
```

```
[1] 0
```

18.2 The `ifelse()` function

The `my_abs()` function we wrote above only works with scalar inputs (vectors of length one). If we try use it with a vector it will return an error. A useful function in R is the `ifelse()` function, which can do if-else statements on vectors. The function takes 3 arguments:

1. A logical vector (such as a condition to check).
2. What to do when `TRUE`.
3. What to do when `FALSE`.

Let's use the `ifelse()` function to get the absolute value of the sequence $(-3, -2, -1, 0, 1, 2, 3)$:

```
x <- -3:3
ifelse(x < 0, -x, x)
```

```
[1] 3 2 1 0 1 2 3
```

The first argument checks the condition $x < 0$. This will be `TRUE` for the first 3 elements, and `FALSE` everywhere else. Let's see this:

```
x < 0
```

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

The second argument is what to do when the condition is `TRUE`. This is to turn the x to $-x$, which makes the negative values positive. We can see that it did precisely this for the first 3 elements.

The third argument is what to do when the condition is `FALSE`. By writing just `x`, we are telling R to leave those elements unchanged.

We can also use the `ifelse()` statement to create other types of variables. For example, we can use it to make character variables:

```
x <- -3:3
ifelse(x < 0, "Negative", "Non-negative")
```

```
[1] "Negative"     "Negative"     "Negative"     "Non-negative" "Non-negative"
[6] "Non-negative" "Non-negative"
```

When $x < 0$, the output element is `"Negative"` and when $x \geq 0$, the output element is `"Non-negative"`.

18.3 "If else-if else" statements

Sometimes we want to do one thing if a certain condition holds, another thing if a different condition holds, and something else in the remaining cases. An example of this is the "sign" function, which tells you the sign in front of a value:

$$sgn(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ +1 & \text{otherwise} \end{cases}$$

If the value is negative, we get -1. If it's zero we get 0. If it's positive (the remaining case), we get $+1$.

To do this in R, we can nest several if-else statements. We simply write `else if` for the intermediate case:

```
sgn <- function(x) {
  if (x < 0) {
    return(-1)
  } else if (x == 0) {
    return(0)
  } else {
```

```
      return(+1)
    }
  }
```

Like above, after `if` we write the condition to check in parentheses and in curly brackets what to do if the condition is `TRUE`. We then write `else if` and write another condition to check, as well as what to do when that condition is `TRUE` in curly brackets. We then write after `else` what to do if neither of the above conditions are `TRUE`.

Let's go through what R does here. Given an input `x`:

1. R checks the condition `x < 0`. If it is `TRUE` it returns -1. If it is `FALSE` it goes to the next step.
2. R checks the condition `x == 0`. If it is `TRUE` it returns 0. If it is `FALSE` it goes to the next step.
3. R returns $+1$ (happens if neither of the above conditions are `TRUE`).

Let's try it out:

```
sgn(-2)
```

```
[1] -1
```

```
sgn(3)
```

```
[1] 1
```

```
sgn(0)
```

```
[1] 0
```

"If else-if else" statements with vectors

The above approach only works for scalars. If we want to do this with vectors, we can nest the `ifelse()` function inside itself like this:

```
x <- -3:3
x
```

```
[1] -3 -2 -1  0  1  2  3
```

```
ifelse(x < 0, -1, ifelse(x == 0, 0, 1))
```

```
[1] -1 -1 -1  0  1  1  1
```

Let's take apart what's happening in `ifelse(x < 0, -1, ifelse(x == 0, 0, 1))` for an element in `x`:

- R first checks for each element in x if $x < 0$. If it is `TRUE`, it returns -1; if it is `FALSE`, it goes to the next `ifelse()`.
- If we go to the next `ifelse()`, it checks if the element satisfies $x = 0$. If this is `TRUE` it returns 0; if it is `FALSE`, it returns $+1$.

Chapter 19

Merging

Often we want to be able to join datasets together to analyze the relationship between variables. For example, suppose we are interested in the relationship between the price of crude oil on commodities markets and the average price of petrol at the pumps over time. We download data on daily petrol prices and daily data on crude oil prices. But if one dataset has more observations (spans a longer time period) than the other, or one has missing observations (such as weekend and holiday values missing), it's not so straightforward to match them up together.

Fortunately, the `merge()` function solves these problems. We will learn how to use that function here.

19.1 Data Cleaning

First we read in and clean the petrol price data. You can download the dataset here.

```
df1 <- read.csv("avg_daily_petrol_prices.csv")
# Format dates:
df1$date <- as.Date(df1$date, format = "%Y-%m-%d")
summary(df1)
```

```
      date                  e5              e10             diesel
 Min.   :2014-06-08   Min.   :1.159   Min.   :1.130   Min.   :0.9558
 1st Qu.:2016-07-03   1st Qu.:1.340   1st Qu.:1.318   1st Qu.:1.1322
 Median :2018-07-29   Median :1.402   Median :1.379   Median :1.2353
 Mean   :2018-07-29   Mean   :1.456   Mean   :1.423   Mean   :1.2811
 3rd Qu.:2020-08-23   3rd Qu.:1.522   3rd Qu.:1.479   3rd Qu.:1.3217
 Max.   :2022-09-18   Max.   :2.261   Max.   :2.203   Max.   :2.3343
```

```
nrow(df1)
```

```
[1] 3025
```

We now do the same with the Brent crude oil prices. You can download the dataset here.

```
df2 <- read.csv("Europe_Brent_Spot_Price_FOB.csv", skip = 4)
# Format dates:
df2$Day <- as.Date(df2$Day, format = "%m/%d/%Y")
# Rename variables:
names(df2) <- c("date", "crude_oil")
# Sort ascending:
df2 <- df2[order(df2$date), ]
summary(df2)
```

```
     date                crude_oil
Min.   :1987-05-20   Min.   :  9.10
1st Qu.:1996-03-06   1st Qu.: 19.03
Median :2005-01-04   Median : 38.08
Mean   :2005-01-12   Mean   : 48.22
3rd Qu.:2013-11-24   3rd Qu.: 69.67
Max.   :2022-09-19   Max.   :143.95
```

```
nrow(df2)
```

```
[1] 8970
```

We see that the petrol price data covers 2014-2022, and the crude oil price data covers 1987-2022. If we look closer at the dates, we notice that the crude oil price data doesn't include the weekends, whereas the petrol price data does:

```
head(df1$date, n = 10)
```

```
[1] "2014-06-08" "2014-06-09" "2014-06-10" "2014-06-11" "2014-06-12"
[6] "2014-06-13" "2014-06-14" "2014-06-15" "2014-06-16" "2014-06-17"
```

```
head(df2$date, n = 10)
```

```
[1] "1987-05-20" "1987-05-21" "1987-05-22" "1987-05-25" "1987-05-26"
[6] "1987-05-27" "1987-05-28" "1987-05-29" "1987-06-01" "1987-06-02"
```

19.2 Merging

19.2.1 The `merge()` Command

Now that the two datasets are clean, we merge the two using the `merge()` function. The first two arguments of the `merge()` function are the two datasets we want to merge. The third argument, `by`, specifies the variable name (in quotations) that link the two datasets. In our case, the variable linking the two is the date variable.

```
df <- merge(df1, df2, by = "date")
summary(df)
```

```
     date                  e5              e10             diesel
 Min.   :2014-06-09   Min.   :1.159   Min.   :1.130   Min.   :0.9558
 1st Qu.:2016-07-04   1st Qu.:1.339   1st Qu.:1.318   1st Qu.:1.1302
 Median :2018-07-26   Median :1.402   Median :1.379   Median :1.2345
 Mean   :2018-07-27   Mean   :1.455   Mean   :1.422   Mean   :1.2797
 3rd Qu.:2020-08-19   3rd Qu.:1.521   3rd Qu.:1.478   3rd Qu.:1.3216
 Max.   :2022-09-16   Max.   :2.261   Max.   :2.203   Max.   :2.3343
   crude_oil
 Min.   :  9.12
 1st Qu.: 48.54
 Median : 61.18
 Mean   : 63.47
 3rd Qu.: 72.97
 Max.   :133.18
```

```
nrow(df)
```

```
[1] 2107
```

We notice that the dataset becomes much smaller: only 2,107 observations instead of 3,025 in `df1` and 8,970 in `df2`. This is because `df1` only contained dates from 2014 onwards, and `df2` only contains data on weekdays. The merged datasets only includes weekdays between 2014-2022, and is thus much smaller.

19.2.2 Keeping Unmatched Observations

If we want to avoid dropping the observations where there is no match, we can use one of the following options:

- `all.x = TRUE` : Keeps all observations in the 1st dataset, but only merges data from the 2nd dataset when there is a match. When there is no match, variables in the 2nd dataset get assigned `NA` values.
- `all.y = TRUE` : Keeps all observations in the 2nd dataset, but only merges data from the 1st dataset when there is a match. When there is no match,

variables in the 1st dataset get assigned `NA` values.
- `all = TRUE` : This keeps all observations from both datasets, and variables get assigned `NA` values when there is no match. This is equivalent to setting both `all.x = TRUE` and `all.y = TRUE`.

For example, suppose we use the `all.x = TRUE` option:

```r
df <- merge(df1, df2, by = "date", all.x = TRUE)
summary(df)
```

```
     date                   e5               e10              diesel
 Min.   :2014-06-08   Min.   :1.159    Min.   :1.130    Min.   :0.9558
 1st Qu.:2016-07-03   1st Qu.:1.340    1st Qu.:1.318    1st Qu.:1.1322
 Median :2018-07-29   Median :1.402    Median :1.379    Median :1.2353
 Mean   :2018-07-29   Mean   :1.456    Mean   :1.423    Mean   :1.2811
 3rd Qu.:2020-08-23   3rd Qu.:1.522    3rd Qu.:1.479    3rd Qu.:1.3217
 Max.   :2022-09-18   Max.   :2.261    Max.   :2.203    Max.   :2.3343

   crude_oil
 Min.   :  9.12
 1st Qu.: 48.54
 Median : 61.18
 Mean   : 63.47
 3rd Qu.: 72.97
 Max.   :133.18
 NA's   :918
```

```r
nrow(df)
```

```
[1] 3025
```

We see that we have 3,025 rows, the same as the original `df1`. However, the variable `crude_oil`, which was merged from `df2` now has 918 missing values. Any time `df2$crude_oil` didn't have a value for a date in `df1`, we replace it with an `NA`. When we didn't specify `all.x = TRUE`, we ended up with 2,107 observations. This is because it dropped all the rows where there was no match. The 2,107 comes from the 3,025 original rows in `df1`, minus the 918 rows where there was no corresponding match in `df2` (3,025-918=2,107).

19.2.3   Other Merging Options

Finally, we end with some further remarks on the `merge()` function. First, if you are merging on multiple variables, you can include a vector of variable names in the `by` argument. For example, suppose you are merging the two datasets:

- `df1`: contains the revenue in each market area (variable `market_area`) and date (variable `date`).

- **df2**: contains the advertising expenditure in each market area (variable `market_area`) and date (variable `date`).

Let's create two example datasets for illustration purposes here:

```r
df1 <- data.frame(expand.grid(
  market_area = c("Market A", "Market B"),
  date = seq(as.Date("2022-01-01"), as.Date("2022-03-01"), by = "month")
))
df1$revenue <- runif(n = nrow(df1), min = 0, max = 1000)

df2 <- data.frame(expand.grid(
  market_area = c("Market A", "Market B"),
  date = seq.Date(as.Date("2022-01-01"), as.Date("2022-03-01"), by = "month")
))
df2$advertising_exp <- runif(n = nrow(df1), min = 0, max = 500)

df1
```

```
  market_area       date   revenue
1    Market A 2022-01-01 485.76845
2    Market B 2022-01-01 940.01892
3    Market A 2022-02-01  74.59912
4    Market B 2022-02-01 785.79297
5    Market A 2022-03-01 109.54504
6    Market B 2022-03-01 961.13105
```

```r
df2
```

```
  market_area       date advertising_exp
1    Market A 2022-01-01        480.6382
2    Market B 2022-01-01        426.6870
3    Market A 2022-02-01        236.2161
4    Market B 2022-02-01        302.1044
5    Market A 2022-03-01        415.5219
6    Market B 2022-03-01        212.2895
```

Some notes on the functions used here:

- The `expand.grid()` function here creates every combination of each market area and date (you won't be asked to use this function in the exam).
- The `seq()` function can also be used to create sequences of dates. We can specify the step length to be `"day" "month"`, `"quarter"` or `"year"`.
- The `runif()` function creates `n` random numbers between `min` and `max` (you won't be asked to use this function in the exam).

We can merge these two datasets by simply including a vector of the variable names in the `by` argument:

```
df <- merge(df1, df2, by = c("market_area", "date"))
df
```

```
  market_area        date   revenue advertising_exp
1    Market A 2022-01-01 485.76845          480.6382
2    Market A 2022-02-01  74.59912          236.2161
3    Market A 2022-03-01 109.54504          415.5219
4    Market B 2022-01-01 940.01892          426.6870
5    Market B 2022-02-01 785.79297          302.1044
6    Market B 2022-03-01 961.13105          212.2895
```

If the variable names are different in the two datasets, we could change the names of the variables to make them match before merging. But what we could do instead is use the `by.x` and `by.y` options in the `merge()` function. For instance, suppose in the previous example the market area variable was called `"market"` in `df2` instead of `"market_area"`. Let's change the name of the variable in `df2` to that:

```
names(df2)[names(df2) == "market_area"] <- "market"
```

If I want to merge the two datasets in this case I can do:

```
df <- merge(df1, df2, by.x = c("market_area", "date"),
            by.y = c("market", "date"))
df
```

```
  market_area        date   revenue advertising_exp
1    Market A 2022-01-01 485.76845          480.6382
2    Market A 2022-02-01  74.59912          236.2161
3    Market A 2022-03-01 109.54504          415.5219
4    Market B 2022-01-01 940.01892          426.6870
5    Market B 2022-02-01 785.79297          302.1044
6    Market B 2022-03-01 961.13105          212.2895
```

Finally, by default, `merge()` will sort the data by the merging variable(s). To avoid this behaviour you can use the `sort = FALSE` option.

Chapter 20

Reshaping

20.1 From Long to Wide

Suppose we have the following dataset:

```
long <- data.frame(
  id       = rep(1:3, each = 2),
  variable = rep(c("x", "y"), times = 3),
  value    = c(3, 5, 4, 8, 3, 1)
)
long
```

```
  id variable value
1  1        x     3
2  1        y     5
3  2        x     4
4  2        y     8
5  3        x     3
6  3        y     1
```

This dataset is in what is called "long" format. We have 3 individuals, with IDs 1, 2, 3. For each individual we have 2 variables, x and y, and for each individual and variable we observe the value in the `value` column.

If we want to reshape this dataset so that it has only 1 row per individual (3 rows in total), with the variables x and y as separate variables, we can use functions from the `reshape2` package. Install the package with `install.packages("reshape2")`. You can use the `dcast()` function from this package to reshape the data as follows:

```r
library(reshape2)
wide <- dcast(long, id ~ variable)
wide
```

```
  id x y
1  1 3 5
2  2 4 8
3  3 3 1
```

The first argument is the name of the dataset. The second argument is the formula for how to reshape. We put the ID variable that we want to represent the rows first, then we use the ~ symbol, and then we put the variable with the different variable names.

20.2   From Wide to Long

We can also go the other direction. Let's get back to our original data by reshaping the new `wide` dataset back to `long`. Let's call the output `long2`. We can do that with the `melt()` function:

```r
long2 <- melt(wide, id.vars = "id")
long2
```

```
  id variable value
1  1        x     3
2  2        x     4
3  3        x     3
4  1        y     5
5  2        y     8
6  3        y     1
```

Again, the first argument is the name of the dataset. The second is the variable is the varying representing the observation IDs.

20.3   Example Usage Case

Sometimes with `ggplot`, we need to have the data in long format. This happens when we want to plot multiple variables on the same plot with different colors. Let's use the petrol price dataset from Chapter 19 to demonstrate this:

```r
# Read in and clean petrol price data:
df <- read.csv("avg_daily_petrol_prices.csv")
df$date <- as.Date(df$date)  # format dates
head(df)
```

```
        date       e5      e10   diesel
1 2014-06-08 1.551987 1.477774 1.353583
2 2014-06-09 1.576623 1.483362 1.385182
3 2014-06-10 1.569619 1.478455 1.374060
4 2014-06-11 1.572578 1.481119 1.377091
5 2014-06-12 1.574652 1.480383 1.378247
6 2014-06-13 1.584659 1.492049 1.387577
```

The dataset is currently in "wide" format. The date runs down the dataset and the variables (petrol prices) at each date are stored horizontally from this. Let's go to "long" format with the `melt()` function, where `"date"` represents the observation IDs.

```
df2 <- melt(df, "date")
head(df2)
```

```
        date variable    value
1 2014-06-08       e5 1.551987
2 2014-06-09       e5 1.576623
3 2014-06-10       e5 1.569619
4 2014-06-11       e5 1.572578
5 2014-06-12       e5 1.574652
6 2014-06-13       e5 1.584659
```

Now the dataset is in long format: we have the date, a variable representing the variable names (`variable`), and the values of each variable (`value`).

Let's use this long-format data to plot the petrol prices over time for each type of petrol:

```
library(ggplot2)
ggplot(df2, aes(date, value, color = variable)) +
  geom_line()
```
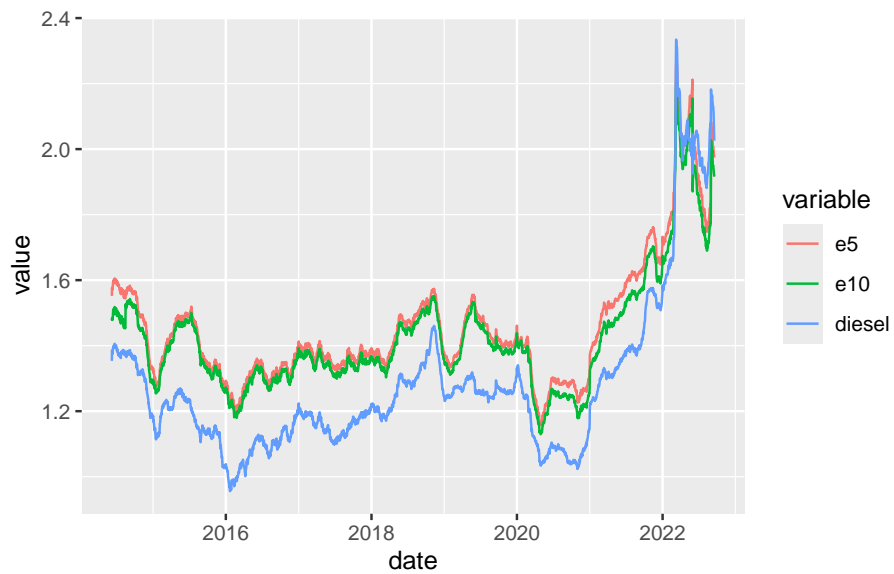
We can customize this plot a bit with:

```r
levels(df2$variable) <- c("E5", "E10", "Diesel")
ggplot(df2, aes(date, value, color = variable)) +
  geom_line() +
  xlab("") +
  ylab("Average Daily Petrol Price (in Euro)") +
  scale_color_discrete(name = "Petrol Type:") +
  theme_minimal() +
  theme(legend.direction = "horizontal",
        legend.position = "bottom")
```
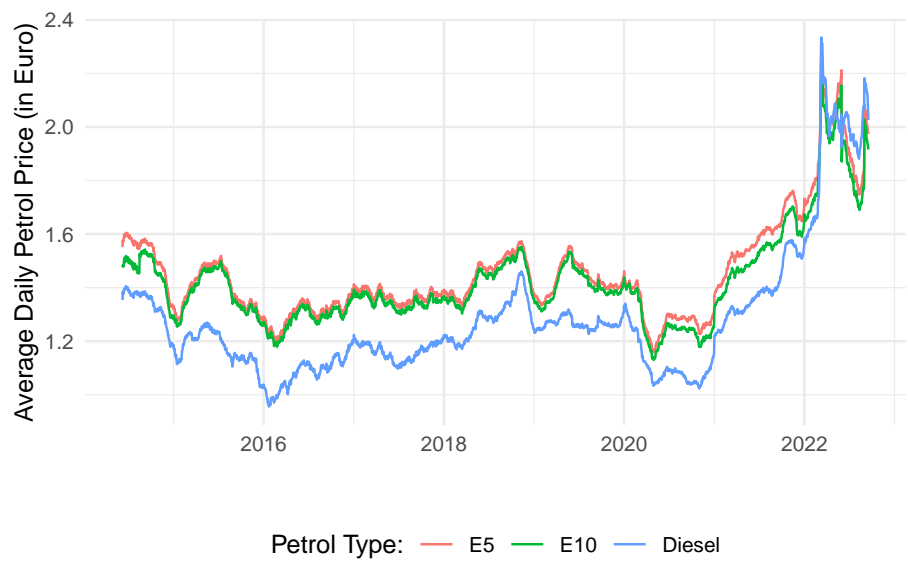
Chapter 21

Aggregating by Group

Sometimes we want to get the average or sum of a variable by group. We can do this in R using the `aggregate()` function. Let's learn how to use this function using an example. We will use the daily average petrol price data from before. Suppose we wanted to get the average petrol price by year from this. Let's load up the data again:

```r
df <- read.csv("avg_daily_petrol_prices.csv")
df$date <- as.Date(df$date)  # format dates
```

To aggregate by year, we first need to create a variable which gives the year corresponding to the date. There are several ways to do this.

One way to get the year is to re-format the date so that it only shows the year. Recall from Chapter 13, that `%Y` represents the year in R:

```r
df$year <- format(df$date, format = "%Y")
```

But an easier way is to use the `year()` function from the `lubridate` package. You can install it with `install.packages("lubridate")`. With this package loaded, we can use the function with:

```r
library(lubridate)
df$year <- year(df$date)
```

The next step is to use the `aggregate()` function. If I want the average price of E10 petrol by year, I can do the following:

```r
aggregate(e10 ~ year, FUN = mean, data = df)
```

```
  year      e10
1 2014 1.461632
2 2015 1.373425
3 2016 1.282375
4 2017 1.345430
5 2018 1.430955
6 2019 1.408177
7 2020 1.253603
8 2021 1.522833
9 2022 1.875897
```

Here we first provide the formula: We want the average of `e10` by `year` so we write `e10 ~ year`. The function we want to use is the `mean` (to get the average). Finally, we provide the name of the dataset, `df`.

We can calculate the average of all petrol prices by year by adding all the variables with `cbind()`:

```
aggregate(cbind(e5, e10, diesel) ~ year, FUN = mean, data = df)
```

```
  year      e5      e10   diesel
1 2014 1.519195 1.461632 1.334612
2 2015 1.393170 1.373425 1.173013
3 2016 1.302767 1.282375 1.081282
4 2017 1.368414 1.345430 1.161306
5 2018 1.454098 1.430955 1.287264
6 2019 1.430592 1.408177 1.265341
7 2020 1.288080 1.253603 1.111068
8 2021 1.579992 1.522833 1.387114
9 2022 1.933512 1.875897 1.941386
```

If you have a lot of variables, you can get the average of all of them by year by replacing the part before the `~` with a dot:

```
aggregate(. ~ year, FUN = mean, data = df)
```

```
  year    date      e5      e10   diesel
1 2014 16332.0 1.519195 1.461632 1.334612
2 2015 16618.0 1.393170 1.373425 1.173013
3 2016 16983.5 1.302767 1.282375 1.081282
4 2017 17349.0 1.368414 1.345430 1.161306
5 2018 17714.0 1.454098 1.430955 1.287264
6 2019 18079.0 1.430592 1.408177 1.265341
7 2020 18444.5 1.288080 1.253603 1.111068
8 2021 18810.0 1.579992 1.522833 1.387114
9 2022 19123.0 1.933512 1.875897 1.941386
```

You may notice that this also gives us the average of the date variable in the year as numbers that don't look like dates. Similar to Excel, underlying each date in R is a number which is the number of days since the 1/1/1970 (in Excel it's days since 1/1/1900). 1/1/2014 is 16,071 days since 1/1/1970, so that is why we see numbers this size there.

We can also replace `mean` with any function we like. For example, to get the maximum average daily price in a year:

```
aggregate(e10 ~ year, FUN = max, data = df)
```

```
  year       e10
1 2014 1.542430
2 2015 1.499718
3 2016 1.375595
4 2017 1.393057
5 2018 1.552005
6 2019 1.533291
7 2020 1.438141
8 2021 1.703694
9 2022 2.203362
```

To get the number of observations per year:

```
aggregate(e10 ~ year, FUN = length, data = df)
```

```
  year e10
1 2014 207
2 2015 365
3 2016 366
4 2017 365
5 2018 365
6 2019 365
7 2020 366
8 2021 365
9 2022 261
```

Tutorial Exercises

Below are links to the exercises and solutions for each of the tutorial exercises.

Week 2: Chapters 2-8

- Exercises
- Solutions

Week 3: Chapters 9-12

- Exercises
- Solutions

Week 4: Chapter 13

- Exercises
- Solutions

Week 5: Chapters 14-15

- Exercises
- Solutions

Week 6: Chapters 16-18

- Exercises
- Solutions

Week 7: Chapters 19-21

- Exercises
- Solutions

Sample Exam

- Exercises
- Solutions

Extra Practice Questions

- Exercises
- Solutions