Modern Statistics with R

From wrangling and exploring data to inference and predictive modelling

Måns Thulin

2021-11-25 - Version 1.0.1

Contents

To cite this book, please use the following:

- Thulin, M. (2021). *Modern Statistics with R*. Eos Chasma Press. ISBN 9789152701515.

Chapter 1

Introduction

1.1 Welcome to R

Welcome to the wonderful world of R!

R is not like other statistical software packages. It is free, versatile, fast, and modern. It has a large and friendly community of users that help answer questions and develop new R tools. With more than 17,000 add-on packages available, R offers more functions for data analysis than any other statistical software. This includes specialised tools for disciplines as varied as political science, environmental chemistry, and astronomy, and new methods come to R long before they come to other programs. R makes it easy to construct reproducible analyses and workflows that allow you to easily repeat the same analysis more than once.

R is not like other programming languages. It was developed by statisticians as a tool for data analysis and not by software engineers as a tool for other programming tasks. It is designed from the ground up to handle data, and that shows. But it is also flexible enough to be used to create interactive web pages, automated reports, and APIs.

R is, simply put, currently the best tool there is for data analysis.

1.2 About this book

This book was born out of lecture notes and materials that I created for courses at the University of Edinburgh, Uppsala University, Dalarna University, the Swedish University of Agricultural Sciences, and Karolinska Institutet. It can be used as a textbook, for self-study, or as a reference manual for R. No background in programming is assumed.

This is not a book that has been written with the intention that you should read it back-to-back. Rather, it is intended to serve as a guide to what to do next as you explore R. Think of it as a conversation, where you and I discuss different topics related to data analysis and data wrangling. At times I'll do the talking, introduce concepts and pose questions. At times you'll do the talking, working with exercises and discovering all that R has to offer. The best way to learn R is to use R. You should strive for active learning, meaning that you should spend more time with R and less time stuck with your nose in a book. Together we will strive for an exploratory approach, where the text guides you to discoveries and the exercises challenge you to go further. This is how I've been teaching R since 2008, and I hope that it's a way that you will find works well for you.

The book contains more than 200 exercises. Apart from a number of open-ended questions about ethical issues, all exercises involve R code. These exercises all have worked solutions. It is highly recommended that you actually work with all the exercises, as they are central to the approach to learning that this book seeks to support: using R to solve problems is a much better way to learn the language than to just read about how to use R to solve problems. Once you have finished an exercise (or attempted but failed to finish it) read the proposed solution - it may differ from what you came up with and will sometimes contain comments that you may find interesting. Treat the proposed solutions as a part of our conversation. As you work with the exercises and compare your solutions to those in the back of the book, you will gain more and more experience working with R and build your own library of examples of how problems can be solved.

Some books on R focus entirely on data science - data wrangling and exploratory data analysis - ignoring the many great tools R has to offer for deeper data analyses. Others focus on predictive modelling or classical statistics but ignore data-handling, which is a vital part of modern statistical work. Many introductory books on statistical methods put too little focus on recent advances in computational statistics and advocate methods that have become obsolete. Far too few books contain discussions of ethical issues in statistical practice. This book aims to cover all of these topics and show you the state-of-the-art tools for all these tasks. It covers data science and (modern!) classical statistics as well as predictive modelling and machine learning, and deals with important topics that rarely appear in other introductory texts, such as simulation. It is written for R 4.0 or later and will teach you powerful add-on packages like `data.table`, `dplyr`, `ggplot2`, and `caret`.

The book is organised as follows:

Chapter 2 covers basic concepts and shows how to use R to compute descriptive statistics and create nice-looking plots.

Chapter 3 is concerned with how to import and handle data in R, and how to perform routine statistical analyses.

Chapter 4 covers exploratory data analysis using statistical graphics, as well as un-

supervised learning techniques like principal components analysis and clustering. It also contains an introduction to R Markdown, a powerful markup language that can be used e.g. to create reports.

Chapter 5 describes how to deal with messy data - including filtering, rearranging and merging datasets - and different data types.

Chapter 6 deals with programming in R, and covers concepts such as iteration, conditional statements and functions.

Chapters 4-6 can be read in any order.

Chapter 7 is concerned with classical statistical topics like estimation, confidence intervals, hypothesis tests, and sample size computations. Frequentist methods are presented alongside Bayesian methods utilising weakly informative priors. It also covers simulation and important topics in computational statistics, such as the bootstrap and permutation tests.

Chapter 8 deals with various regression models, including linear, generalised linear and mixed models. Survival models and methods for analysing different kinds of censored data are also included, along with methods for creating matched samples.

Chapter 9 covers predictive modelling, including regularised regression, machine learning techniques, and an introduction to forecasting using time series models. Much focus is given to cross-validation and ways to evaluate the performance of predictive models.

Chapter 10 gives an overview of more advanced topics, including parallel computing, matrix computations, and integration with other programming languages.

Chapter 11 covers debugging, i.e. how to spot and fix errors in your code. It includes a list of more than 25 common error and warning messages, and advice on how to resolve them.

Chapter 12 covers some mathematical aspects of methods used in Chapters 7-9.

Finally, Chapter 13 contains fully worked solutions to all exercises in the book.

The datasets that are used for the examples and exercises can be downloaded from

http://www.modernstatisticswithr.com/data.zip

I have opted not to put the datasets in an R package, because I want you to practice loading data from files, as this is what you'll be doing whenever you use R for real work.

for the book is available at it's GitHub page (https://github.com/mthulin/mswr-book).

I am indebted to the numerous readers who have provided feedback on drafts of this book. My sincerest thanks go out to all of you. Any remaining misprints are, obviously, entirely my own fault.

Finally, there are countless packages and statistical methods that deserve a mention but aren't included in the book. Like any author, I've had to draw the line somewhere. If you feel that something is missing, feel free to post an issue on the book's GitHub page, and I'll gladly consider it for future revisions.

Chapter 2

The basics

Let's start from the very beginning. This chapter acts as an introduction to R. It will show you how to install and work with R and RStudio.

After working with the material in this chapter, you will be able to:

- Create reusable R scripts,
- Store data in R,
- Use functions in R to analyse data,
- Install add-on packages adding additional features to R,
- Compute descriptive statistics like the mean and the median,
- Do mathematical calculations,
- Create nice-looking plots, including scatterplots, boxplots, histograms and bar charts,
- Find errors in your code.

2.1 Installing R and RStudio

To download R, go to the R Project website

https://cran.r-project.org/mirrors.html

Choose a *download mirror*, i.e. a server to download the software from. I recommend choosing a mirror close to you. You can then choose to download R for either Linux[1], Mac or Windows by following the corresponding links (Figure 2.1).

The version of R that you should download is called the (base) binary. Download and run it to install R. You may see mentions of 64-bit and 32-bit versions of R; if

[1]For many Linux distributions, R is also available from the package management system.

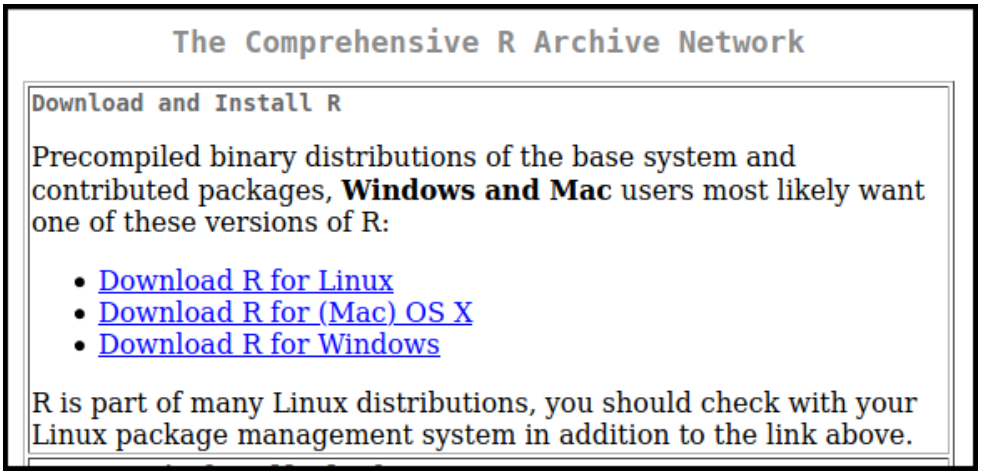

Figure 2.1: A screenshot from the R download page at https://ftp.acc.umu.se/mirror/CRAN/

you have a modern computer (which in this case means a computer from 2010 or later), you should go with the 64-bit version.

You have now installed the R programming language. Working with it is easier with an *integrated development environment*, or IDE for short, which allows you to easily write, run and debug your code. This book is written for use with the RStudio IDE, but 99.9 % of it will work equally well with other IDE's, like Emacs with ESS or Jupyter notebooks.

To download RStudio, go to the RStudio download page

https://rstudio.com/products/rstudio/download/#download

Click on the link to download the installer for your operating system, and then run it.

2.2 A first look at RStudio

When you launch RStudio, you will see three or four panels:

1. The *Environment* panel, where a list of the data you have imported and created can be found.
2. The *Files*, *Plots* and *Help* panel, where you can see a list of available files, will be able to view graphs that you produce, and can find help documents for different parts of R.
3. The *Console* panel, used for running code. This is where we'll start with the first few examples.

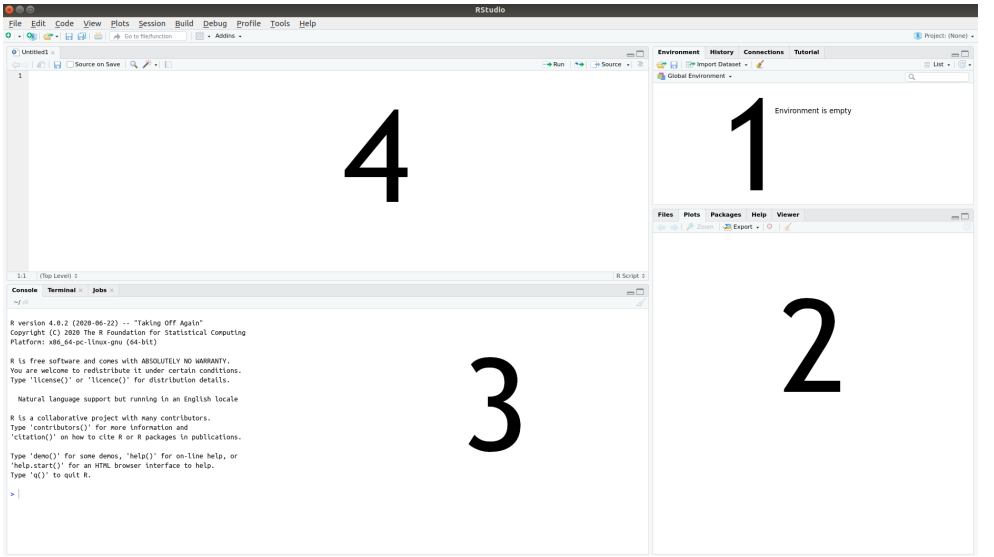

Figure 2.2: The four RStudio panels.

4. The *Script* panel, used for writing code. This is where you'll spend most of your time working.

If you launch RStudio by opening a file with R code, the *Script* panel will appear, otherwise it won't. Don't worry if you don't see it at this point - you'll learn how to open it soon enough.

The *Console* panel will contain R's startup message, which shows information about which version of R you're running[2]:

```
R version 4.1.0 (2021-05-18) -- "Camp Pontanezen"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

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

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
```

[2]In addition to the version number, each relase of R has a nickname referencing a Peanuts comic by Charles Schulz. The "Camp Pontanezen" nickname of R 4.1.0 is a reference to the Peanuts comic from February 12, 1986.

```
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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

You can resize the panels as you like, either by clicking and dragging their borders or using the minimise/maximise buttons in the upper right corner of each panel.

When you exit RStudio, you will be asked if you wish to *save your workspace*, meaning that the data that you've worked with will be stored so that it is available the next time you run R. That might sound like a good idea, but in general, I recommend that you don't save your workspace, as that often turns out to cause problems down the line. It is almost invariably a much better idea to simply rerun the code you worked with in your next R session.

2.3 Running R code

Everything that we do in R revolves around *code*. The code will contain instructions for how the computer should treat, analyse and manipulate[3] data. Thus each line of code tells R to do something: compute a mean value, create a plot, sort a dataset, or something else.

Throughout the text, there will be code chunks that you can paste into the Console panel. Here is the first example of such a code chunk. Type or copy the code into the Console and press Enter on your keyboard:

```
1+1
```

Code chunks will frequently contain multiple lines. You can select and copy both lines from the digital version of this book and simultaneously paste them directly into the Console:

```
2*2
1+2*3-5
```

As you can see, when you type the code into the Console panel and press Enter, R *runs* (or *executes*) the code and returns an answer. To get you started, the first exercise will have you write a line of code to perform a computation. You can find a solution to this and other exercises at the end of the book, in Chapter 13.

~

[3]The word manipulate has different meanings. Just to be perfectly clear: whenever I speak of *manipulating data* in this book, I will mean *handling and transforming the data*, not tampering with it.

Exercise 2.1. Use R to compute the product of the first ten integers: $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot 8 \cdot 9 \cdot 10$.

2.3.1 R scripts

When working in the Console panel[4], you can use the up arrow ↑ on your keyboard to retrieve lines of code that you've previously used. There is however a much better way of working with R code: to put it in *script files*. These are files containing R code, that you can save and then run again whenever you like.

To create a new script file in RStudio, press Ctrl+Shift+N on your keyboard, or select *File > New File > R Script* in the menu. This will open a new Script panel (or a new tab in the Script panel, in case it was already open). You can then start writing your code in the Script panel. For instance, try the following:

```
1+1
2*2
1+2*3-5
(1+2)*3-5
```

In the Script panel, when you press Enter, you insert a new line instead of running the code. That's because the Script panel is used for *writing* code rather than *running* it. To actually run the code, you must send it to the Console panel. This can be done in several ways. Let's give them a try to see which you prefer.

To run the entire script do one of the following:

- Press the Source button in the upper right corner of the Script panel.
- Press Ctrl+Shift+Enter on your keyboard.
- Press Ctrl+Alt+Enter on your keyboard to run the code without printing the code and its output in the Console.

To run a part of the script, first select the lines you wish to run, e.g. by highlighting them using your mouse. Then do one of the following:

- Press the Run button at the upper right corner of the Script panel.
- Press Ctrl+Enter on your keyboard (this is how I usually do it!).

To save your script, click the Save icon, choose *File > Save* in the menu or press Ctrl+S. R script files should have the file extension `.R`, e.g. `My first R script.R`. Remember to save your work often, and to save your code for all the examples and exercises in this book - you will likely want to revisit old examples in the future, to see how something was done.

[4] I.e. when the Console panel is active and you see a blinking text cursor in it.

2.4 Variables and functions

Of course, R is so much more than just a fancy calculator. To unlock its full potential, we need to discuss two key concepts: *variables* (used for storing data) and *functions* (used for doing things with the data).

2.4.1 Storing data

Without data, no data analytics. So how can we store and read data in R? The answer is that we use *variables*. A variable is a name used to store data, so that we can refer to a dataset when we write code. As the name *variable* implies, what is stored can change over time[5].

The code

```
x <- 4
```

is used to *assign* the value `4` to the *variable* `x`. It is read as "assign `4` to `x`". The `<-` part is made by writing a less than sign (`<`) and a hyphen (`-`) with no space between them[6].

If we now type `x` in the Console, R will return the answer `4`. Well, almost. In fact, R returns the following rather cryptic output:

```
[1] 4
```

The meaning of the `4` is clear - it's a 4. We'll return to what the `[1]` part means soon.

Now that we've created a variable, called `x`, and assigned a value (4) to it, `x` will have the value 4 whenever we use it again. This works just like a mathematical formula, where we for instance can insert the value $x = 4$ into the formula $x+1$. The following two lines of code will compute $x + 1 = 4 + 1 = 5$ and $x + x = 4 + 4 = 8$:

```
x + 1
x + x
```

Once we have assigned a value to `x`, it will appear in the Environment panel in RStudio, where you can see both the variable's name and its value.

The left-hand side of the assignment `x <- 4` is always the name of a variable, but the right-hand side can be any piece of code that creates some sort of object to be stored in the variable. For instance, we could perform a computation on the right-hand side and then store the result in the variable:

[5] If you are used to programming languages like C or Java, you should note that R is *dynamically typed*, meaning that the data type of an R variable also can change over time. This also means that there is no need to declare variable types in R (which is either liberating or terrifying, depending on what type of programmer you are).

[6] In RStudio, you can also create the assignment operator `<-` by using the keyboard shortcut Alt+- (i.e. press Alt and the - button at the same time).

```
x <- 1 + 2 + 3 + 4
```

R first evaluates the entire right-hand side, which in this case amounts to computing 1+2+3+4, and then assigns the result (10) to `x`. Note that the value previously assigned to `x` (i.e. `4`) now has been replaced by `10`. After a piece of code has been run, the values of the variables affected by it will have changed. There is no way to revert the run and get that `4` back, save to rerun the code that generated it in the first place.

You'll notice that in the code above, I've added some spaces, for instance between the numbers and the plus signs. This is simply to improve readability. The code works just as well without spaces:

```
x<-1+2+3+4
```

or with spaces in some places but not in others:

```
x<- 1+2+3 + 4
```

However, you can not place a space in the middle of the `<-` arrow. The following will not assign a value to `x`:

```
x < - 1 + 2 + 3 + 4
```

Running that piece of code rendered the output `FALSE`. This is because `< -` with a space has a different meaning than `<-` in R, one that we shall return to in the next chapter.

In rare cases, you may want to switch the direction of the arrow, so that the variable names is on the right-hand side. This is called right-assignment and works just fine too:

```
2 + 2 -> y
```

Later on, we'll see plenty of examples where right-assignment comes in handy.

~

Exercise 2.2. Do the following using R:

1. Compute the sum $924 + 124$ and assign the result to a variable named `a`.
2. Compute $a \cdot a$.

2.4.2 What's in a name?

You now know how to assign values to variables. But what should you call your variables? Of course, you can follow the examples in the previous section and give your variables names like `x`, `y`, `a` and `b`. However, you don't have to use single-letter

names, and for the sake of readability, it is often preferable to give your variables more informative names. Compare the following two code chunks:

```
y <- 100
z <- 20
x <- y - z
```

and

```
income <- 100
taxes <- 20
net_income <- income - taxes
```

Both chunks will run without any errors and yield the same results, and yet there is a huge difference between them. The first chunk is opaque - in no way does the code help us conceive *what it actually computes*. On the other hand, it is perfectly clear that the second chunk is used to compute a net income by subtracting taxes from income. You don't want to be a chunk-one type R user, who produces impenetrable code with no clear purpose. You want to be a chunk-two type R user, who writes clear and readable code where the intent of each line is clear. Take it from me - for years I was a chunk-one guy. I managed to write a lot of useful code, but whenever I had to return to my old code to reuse it or fix some bug, I had difficulties understanding what each line was supposed to do. My new life as a chunk-two guy is better in every way.

So, what's in a name? Shakespeare's balcony-bound Juliet would have us believe that that which we call a rose by any other name would smell as sweet. Translated to R practice, this means that your code will run just fine no matter what names you choose for your variables. But when you or somebody else reads your code, it will help greatly if you call a rose a rose and not `x` or `my_new_variable_5`.

You should note that R is case-sensitive, meaning that `my_variable`, `MY_VARIABLE`, `My_Variable`, and `mY_VariABle` are treated as different variables. To access the data stored in a variable, you must use its exact name - including lower- and uppercase letters in the right places. Writing the wrong variable name is one of the most common errors in R programming.

You'll frequently find yourself wanting to compose variable names out of multiple words, as we did with `net_income`. However, R does not allow spaces in variable names, and so `net income` would not be a valid variable name. There are a few different naming conventions that can be used to name your variables:

- `snake_case`, where words are separated by an underscore (`_`). Example: `househould_net_income`.
- `camelCase` or `CamelCase`, where each new word starts with a capital letter. Example: `househouldNetIncome` or `HousehouldNetIncome`.
- `period.case`, where each word is separated by a period (`.`). You'll find this used a lot in R, but I'd advise that you don't use it for naming variables, as a

period in the middle of a name can have a different meaning in more advanced cases[7]. Example: `household.net.income`.

- `concatenatedwordscase`, where the words are concatenated using only lowercase letters. Adownsidetothisconventionisthatitcanmakevariablenamesverydifficultoreadsousethisatyourownrisk. Example: `householdnetincome`
- `SCREAMING_SNAKE_CASE`, which mainly is used in Unix shell scripts these days. You can use it in R if you like, although you will run the risk of making others think that you are either angry, super excited or stark staring mad[8]. Example: `HOUSEHOULD_NET_INCOME`.

Some characters, including spaces, `-`, `+`, `*`, `:`, `=`, `!` and `$` are not allowed in variable names, as these all have other uses in R. The plus sign `+`, for instance, is used for addition (as you would expect), and allowing it to be used in variable names would therefore cause all sorts of confusion. In addition, variable names can't start with numbers. Other than that, it is up to you how you name your variables and which convention you use. Remember, your variable will smell as sweet regardless of what name you give it, but using a good naming convention will improve readability[9].

Another great way to improve the readability of your code is to use *comments*. A comment is a piece of text, marked by `#`, that is ignored by R. As such, it can be used to explain what is going on to people who read your code (including future you) and to add instructions for how to use the code. Comments can be placed on separate lines or at the end of a line of code. Here is an example:

```
#################################################################
#  This lovely little code snippet can be used to compute   #
#                   your net income.                        #
#################################################################

# Set income and taxes:
income <- 100  # Replace 100 with your income
taxes <- 20    # Replace 20 with how much taxes you pay

# Compute your net income:
net_income <- income - taxes
# Voilà!
```

In the Script panel in RStudio, you can comment and uncomment (i.e. remove the `#` symbol) a row by pressing Ctrl+Shift+C on your keyboard. This is particularly useful if you wish to comment or uncomment several lines - simply select the lines and press Ctrl+Shift+C.

[7]Specifically, the period is used to separate methods and classes in object-oriented programming, which is hugely important in R (although you can use R for several years without realising this).

[8]I find myself using screaming snake case on occasion. Make of that what you will.

[9]I recommend `snake_case` or `camelCase`, just in case that wasn't already clear.

~

Exercise 2.3. Answer the following questions:

1. What happens if you use an invalid character in a variable name? Try e.g. the following:

```
net income <- income - taxes
net-income <- income - taxes
ca$h <- income - taxes
```

2. What happens if you put R code as a comment? E.g.:

```
income <- 100
taxes <- 20
net_income <- income - taxes
# gross_income <- net_income + taxes
```

3. What happens if you remove a line break and replace it by a semicolon `;`? E.g.:

```
income <- 200; taxes <- 30
```

4. What happens if you do two assignments on the same line? E.g.:

```
income2 <- taxes2 <- 100
```

2.4.3 Vectors and data frames

Almost invariably, you'll deal with more than one figure at a time in your analyses. For instance, we may have a list of the ages of customers at a bookstore:

$$28, 48, 47, 71, 22, 80, 48, 30, 31$$

Of course, we could store each observation in a separate variable:

```
age_person_1 <- 28
age_person_2 <- 48
age_person_3 <- 47
# ...and so on
```

…but this quickly becomes awkward. A much better solution is to store the entire list in just one variable. In R, such a list is called a *vector*. We can create a vector using the following code, where `c` stands for *combine*:

```
age <- c(28, 48, 47, 71, 22, 80, 48, 30, 31)
```

The numbers in the vector are called *elements*. We can treat the vector variable `age` just as we treated variables containing a single number. The difference is that the

operations will apply to all elements in the list. So for instance, if we wish to express the ages in months rather than years, we can convert all ages to months using:

```
age_months <- age * 12
```

Most of the time, data will contain measurements of more than one quantity. In the case of our bookstore customers, we also have information about the amount of money they spent on their last purchase:

$$20, 59, 2, 12, 22, 160, 34, 34, 29$$

First, let's store this data in a vector:

```
purchase <- c(20, 59, 2, 12, 22, 160, 34, 34, 29)
```

It would be nice to combine these two vectors into a table, like we would do in a spreadsheet software such as Excel. That would allow us to look at relationships between the two vectors - perhaps we could find some interesting patterns? In R, tables of vectors are called *data frames*. We can combine the two vectors into a data frame as follows:

```
bookstore <- data.frame(age, purchase)
```

If you type `bookstore` into the Console, it will show a simply formatted table with the values of the two vectors (and row numbers):

```
> bookstore
  age purchase
1  28       20
2  48       59
3  47        2
4  71       12
5  22       22
6  80      160
7  48       34
8  30       34
9  31       29
```

A better way to look at the table may be to click on the variable name `bookstore` in the Environment panel, which will open the data frame in a spreadsheet format.

You will have noticed that R tends to print a `[1]` at the beginning of the line when we ask it to print the value of a variable:

```
> age
[1] 28 48 47 71 22 80 48 30 31
```

Why? Well, let's see what happens if we print a longer vector:

```
# When we enter data into a vector, we can put line breaks between
# the commas:
distances <- c(687, 5076, 7270, 967, 6364, 1683, 9394, 5712, 5206,
               4317, 9411, 5625, 9725, 4977, 2730, 5648, 3818, 8241,
               5547, 1637, 4428, 8584, 2962, 5729, 5325, 4370, 5989,
               9030, 5532, 9623)
distances
```

Depending on the size of your Console panel, R will require a different number of rows to display the data in `distances`. The output will look something like this:

```
> distances
 [1]  687 5076 7270  967 6364 1683 9394 5712 5206 4317 9411 5625 9725
[14] 4977 2730 5648 3818 8241 5547 1637 4428 8584 2962 5729 5325 4370
[27] 5989 9030 5532 9623
```

or, if you have a narrower panel,

```
> distances
 [1]  687 5076 7270  967 6364 1683 9394
 [8] 5712 5206 4317 9411 5625 9725 4977
[15] 2730 5648 3818 8241 5547 1637 4428
[22] 8584 2962 5729 5325 4370 5989 9030
[29] 5532 9623
```

The numbers within the square brackets - `[1]`, `[8]`, `[15]`, and so on - tell us which *elements* of the vector that are printed first on each row. So in the latter example, the first element in the vector is `687`, the 8th element is `5712`, the 15th element is `2730`, and so forth. Those numbers, called the *indices* of the elements, aren't exactly part of your data, but as we'll see later they are useful for keeping track of it.

This also tells you something about the inner workings of R. The fact that

```
x <- 4
x
```

renders the output

```
> x
[1] 4
```

tells us that `x` in fact is a vector, albeit with a single element. Almost everything in R is a vector, in one way or another.

Being able to put data on multiple lines when creating vectors is hugely useful, but can also cause problems if you forget to include the closing bracket `)`. Try running the following code, where the final bracket is missing, in your Console panel:

```
distances <- c(687, 5076, 7270, 967, 6364, 1683, 9394, 5712, 5206,
               4317, 9411, 5625, 9725, 4977, 2730, 5648, 3818, 8241,
               5547, 1637, 4428, 8584, 2962, 5729, 5325, 4370, 5989,
               9030, 5532, 9623
```

When you hit Enter, a new line starting with a `+` sign appears. This indicates that R doesn't think that your statement has finished. To finish it, type `)` in the Console and then press Enter.

Vectors and data frames are hugely important when working with data in R. Chapters 3 and 5 are devoted to how to work with these objects.

~

Exercise 2.4. Do the following:

1. Create two vectors, `height` and `weight`, containing the heights and weights of five fictional people (i.e. just make up some numbers!).
2. Combine your two vectors into a data frame.

You will use these vectors in Exercise 2.6.

Exercise 2.5. Try creating a vector using `x <- 1:5`. What happens? What happens if you use `5:1` instead? How can you use this notation to create the vector $(1, 2, 3, 4, 5, 4, 3, 2, 1)$?

2.4.4 Functions

You have some data. Great. But simply having data is not enough - you want to *do* something with it. Perhaps you want to draw a graph, compute a mean value or apply some advanced statistical model to it. To do so, you will use a *function*.

A function is a ready-made set of instructions - code - that tells R to do something. There are thousands of functions in R. Typically, you insert a variable into the function, and it returns an answer. The code for doing this follows the pattern `function_name(variable_name)`. As a first example, consider the function `mean`, which computes the mean of a variable:

```
# Compute the mean age of bookstore customers
age <- c(28, 48, 47, 71, 22, 80, 48, 30, 31)
mean(age)
```

Note that the code follows the pattern `function_name(variable_name)`: the function's name is `mean` and the variable's name is `age`.

Some functions take more than one variable as input, and may also have additional *arguments* (or *parameters*) that you can use to control the behaviour of the function. One such example is `cor`, which computes the correlation between two variables:

```
# Compute the correlation between the variables age and purchase
age <- c(28, 48, 47, 71, 22, 80, 48, 30, 31)
purchase <- c(20, 59, 2, 12, 22, 160, 34, 34, 29)
cor(age, purchase)
```

The answer, 0.59 means that there appears to be a fairly strong positive correlation between age and the purchase size, which implies that older customers tend to spend more. On the other hand, just by looking at the data we can see that the oldest customer - aged 80 - spent much more than anybody else - 160 monetary units. It can happen that such *outliers* strongly influence the computation of the correlation. By default, `cor` uses the Pearson correlation formula, which is known to be sensitive to outliers. It is therefore of interest to also perform the computation using a formula that is more robust to outliers, such as the Spearman correlation. This can be done by passing an additional *argument* to `cor`, telling it which method to use for the computation:

```
cor(age, purchase, method = "spearman")
```

The resulting correlation, 0.35 is substantially lower than the previous result. Perhaps the correlation isn't all that strong after all.

So, how can we know what arguments to pass to a function? Luckily, we don't have to memorise all possible arguments for all functions. Instead, we can look at the *documentation*, i.e. help file, for a function that we are interested in. This is done by typing `?function_name` in the Console panel, or doing a web search for `R function_name`. To view the documentation for the `cor` function, type:

```
?cor
```

The documentation for R functions all follow the same pattern:

- *Description*: a short (and sometimes quite technical) description of what the function does.
- *Usage*: an abstract example of how the function is used in R code.
- *Arguments*: a list and description of the input arguments for the function.
- *Details*: further details about how the function works.
- *Value*: information about the output from the function.
- *Note*: additional comments from the function's author (not always included).
- *References*: references to papers or books related to the function (not always included).
- *See Also*: a list of related functions.
- *Examples*: practical (and sometimes less practical) examples of how to use the function.

The first time that you look at the documentation for an R function, all this information can be a bit overwhelming. Perhaps even more so for `cor`, which is a bit unusual in that it shares its documentation page with three other (heavily related) functions: `var`, `cov` and `cov2cor`. Let the section headlines guide you when you look at the documentation. What information are you looking for? If you're just looking for an example of how the function is used, scroll down to Examples. If you want to know what arguments are available, have a look at Usage and Arguments.

Finally, there are a few functions that don't require any input at all, because they don't do anything with your variables. One such example is `Sys.time()` which prints the current time on your system:

```
Sys.time()
```

Note that even though `Sys.time` doesn't require any input, you still have to write the parentheses `()`, which tells R that you want to run a function.

~

Exercise 2.6. Using the data you created in Exercise 2.4, do the following:

1. Compute the mean height of the people.
2. Compute the correlation between height and weight.

Exercise 2.7. Do the following:

1. Read the documentation for the function `length`. What does it do? Apply it to your `height` vector.
2. Read the documentation for the function `sort`. What does it do? What does the argument `decreasing` (the values of which can be either `FALSE` or `TRUE`) do? Apply the function to your `weight` vector.

2.4.5 Mathematical operations

To perform addition, subtraction, multiplication and division in R, we can use the standard symbols `+`, `-`, `*`, `/`. As in mathematics, expressions within parentheses are evaluated first, and multiplication is performed before addition. So `1 + 2*(8/2)` is $1 + 2 \cdot (8/2) = 1 + 2 \cdot 4 = 1 + 8 = 9$.

In addition to these basic arithmetic operators, R has a number of mathematical functions that you can apply to your variables, including square roots, logarithms and trigonometric functions. Below is an incomplete list, showing the syntax for using the functions on a variable `x`. Throughout, `a` is supposed to be a number.

- `abs(x)`: computes the absolute value $|x|$.
- `sqrt(x)`: computes $\sqrt{x}$.

- `log(x)`: computes the logarithm of x with the natural number e as the base.
- `log(x, base = a)`: computes the logarithm of x with the number a as the base.
- `a^x`: computes a^x.
- `exp(x)`: computes e^x.
- `sin(x)`: computes $\sin(x)$.
- `sum(x)`: when `x` is a vector $x = (x_1, x_2, x_3, \dots, x_n)$, computes the sum of the elements of `x`: $\sum_{i=1}^{n} x_i$.
- `prod(x)`: when `x` is a vector $x = (x_1, x_2, x_3, \dots, x_n)$, computes the product of the elements of `x`: $\prod_{i=1}^{n} x_i$.
- `pi`: a built-in variable with value π, the ratio of the circumference of a circle to its diameter.
- `x %% a`: computes x modulo a.
- `factorial(x)`: computes $x!$.
- `choose(n,k)`: computes $\binom{n}{k}$.

~

Exercise 2.8. Compute the following:

1. $\sqrt{\pi}$
2. $e^2 \cdot log(4)$

Exercise 2.9. R will return non-numerical answers if you try to perform computations where the answer is infinite or undefined. Try the following to see some possible results:

1. Compute 1/0.
2. Compute 0/0.
3. Compute $\sqrt{-1}$.

2.5 Packages

R comes with a ton of functions, but of course these cannot cover all possible things that you may want to do with your data. That's where *packages* come in. Packages are collections of functions and datasets that add new features to R. Do you want to apply some obscure statistical test to your data? Plot your data on a map? Run C++ code in R? Speed up some part of your data handling process? There are R packages for that. In fact, with more than 17,000 packages and counting, there are R packages for just about anything that you could possibly want to do. All packages have been contributed by the R community - that is, by users like you and me.

Most R packages are available from CRAN, the official R repository - a network of servers (so-called *mirrors*) around the world. Packages on CRAN are checked before they are published, to make sure that they do what they are supposed to do and don't contain malicious components. Downloading packages from CRAN is therefore generally considered to be safe.

In the rest of this chapter, we'll make use of a package called `ggplot2`, which adds additional graphical features to R. To install the package from CRAN, you can either select *Tools > Install packages* in the RStudio menu and then write `ggplot2` in the text box in the pop-up window that appears, or use the following line of code:

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

A menu may appear where you are asked to select the location of the CRAN mirror to download from. Pick the one the closest to you, or just use the default option - your choice can affect the download speed, but will in most cases not make much difference. There may also be a message asking whether to create a folder for your packages, which you should agree to do.

As R downloads and installs the packages, a number of technical messages are printed in the Console panel (an example of what these messages can look like during a successful installation is found in Section 11.4). `ggplot2` depends on a number of packages that R will install for you, so expect this to take a few minutes. If the installation finishes successfully, it will finish with a message saying:

```
* DONE (ggplot2)
```

Or, on some systems,

```
package ‘ggplot2’ successfully unpacked and MD5 sums checked
```

If the installation fails for some reason, there will usually be a (sometimes cryptic) error message. You can read more about troubleshooting errors in Section 2.10. There is also a list of common problems when installing packages available on the RStudio support page at https://support.rstudio.com/hc/en-us/articles/200554786-Problem-Installing-Packages.

After you've installed the package, you're still not finished quite yet. The package may have been installed, but its functions and datasets won't be available until you *load* it. This is something that you need to do each time that you start a new R session. Luckily, it is done with a single short line of code using the `library` function[10], that I recommend putting at the top of your script file:

```
library(ggplot2)
```

[10]The use of `library` causes people to erroneously refer to R packages as *libraries*. Think of the library as the place where you store your packages, and calling `library` means that you go to your library to fetch the package.

We'll discuss more details about installing and updating R packages in Section 10.1.

2.6 Descriptive statistics

In the remainder of this chapter, we will study two datasets that are shipped with the `ggplot2` package:

- `diamonds`: describing the prices of more than 50,000 cut diamonds.
- `msleep`: describing the sleep times of 83 mammals.

These, as well as some other datasets, are automatically loaded as data frames when you load `ggplot2`:

```
library(ggplot2)
```

To begin with, let's explore the `msleep` dataset. To have a first look at it, type the following in the Console panel:

```
msleep
```

That shows you the first 10 rows of the data, and some of its columns. It also gives another important piece of information: `83 x 11`, meaning that the dataset has 83 rows (i.e. 83 observations) and 11 columns (with each column corresponding to a variable in the dataset).

There are however better methods for looking at the data. To view all 83 rows and all 11 variables, use:

```
View(msleep)
```

You'll notice that some cells have the value `NA` instead of a proper value. `NA` stands for Not Available, and is a placeholder used by R to point out *missing data*. In this case, it means that the value is unknown for the animal.

To find information about the data frame containing the data, some useful functions are:

```
head(msleep)
tail(msleep)
dim(msleep)
str(msleep)
names(msleep)
```

`dim` returns the numbers of rows and columns of the data frame, whereas `str` returns information about the 11 variables. Of particular importance are the *data types* of the variables (`chr` and `num`, in this instance), which tells us what kind of data we are dealing with (numerical, categorical, dates, or something else). We'll delve deeper

into data types in Chapter 3. Finally, `names` returns a vector containing the names of the variables.

Like functions, datasets that come with packages have documentation describing them. The documentation for `msleep` gives a short description of the data and its variables. Read it to learn a bit more about the variables:

```
?msleep
```

Finally, you'll notice that `msleep` isn't listed among the variables in the Environment panel in RStudio. To include it there, you can run:

```
data(msleep)
```

2.6.1 Numerical data

Now that we know what each variable represents, it's time to compute some statistics. A convenient way to get some descriptive statistics giving a summary of each variable is to use the `summary` function:

```
summary(msleep)
```

For the text variables, this doesn't provide any information at the moment. But for the numerical variables, it provides a lot of useful information. For the variable `sleep_rem`, for instance, we have the following:

```
  sleep_rem
Min.   :0.100
1st Qu.:0.900
Median :1.500
Mean   :1.875
3rd Qu.:2.400
Max.   :6.600
NA's   :22
```

This tells us that the mean of `sleep_rem` is `1.875`, that smallest value is `0.100` and that the largest is `6.600`. The 1st quartile[11] is `0.900`, the median is `1.500` and the third quartile is `2.400`. Finally, there are 22 animals for which there are no values (missing data - represented by `NA`).

Sometimes we want to compute just one of these, and other times we may want to compute summary statistics not included in `summary`. Let's say that we want to compute some descriptive statistics for the `sleep_total` variable. To access a vector inside a data frame, we use a dollar sign: `data_frame_name$vector_name`. So to access the `sleep_total` vector in the `msleep` data frame, we write:

[11] The first quartile is a value such that 25 % of the observations are smaller than it; the 3rd quartile is a value such that 25 % of the observations are larger than it.

```
msleep$sleep_total
```

Some examples of functions that can be used to compute descriptive statistics for this vector are:

```
mean(msleep$sleep_total)       # Mean
median(msleep$sleep_total)     # Median
max(msleep$sleep_total)        # Max
min(msleep$sleep_total)        # Min
sd(msleep$sleep_total)         # Standard deviation
var(msleep$sleep_total)        # Variance
quantile(msleep$sleep_total)   # Various quantiles
```

To see how many animals sleep for more than 8 hours a day, we can use the following:

```
sum(msleep$sleep_total > 8)    # Frequency (count)
mean(msleep$sleep_total > 8)   # Relative frequency (proportion)
```

`msleep$sleep_total > 8` checks whether the total sleep time of each animal is greater than 8. We'll return to expressions like this in Section 3.2.

Now, let's try to compute the mean value for the length of REM sleep for the animals:

```
mean(msleep$sleep_rem)
```

The above call returns the answer `NA`. The reason is that there are `NA` values in the `sleep_rem` vector (22 of them, as we saw before). What we actually wanted was the mean value among the animals for which we know the REM sleep. We can have a look at the documentation for `mean` to see if there is some way we can get this:

```
?mean
```

The argument `na.rm` looks promising - it is "a logical value indicating whether NA values should be stripped before the computation proceeds". In other words, it tells R whether or not to ignore the `NA` values when computing the mean. In order to ignore `NA`:s in the computation, we set `na.rm = TRUE` in the function call:

```
mean(msleep$sleep_rem, na.rm = TRUE)
```

Note that the `NA` values have not been removed from `msleep`. Setting `na.rm = TRUE` simply tells R to ignore them in a particular computation, not to delete them.

We run into the same problem if we try to compute the correlation between `sleep_total` and `sleep_rem`:

```
cor(msleep$sleep_total, msleep$sleep_rem)
```

A quick look at the documentation (`?cor`), tells us that the argument used to ignore `NA` values has a different name for `cor` - it's not `na.rm` but `use`. The reason will

become evident later on, when we study more than two variables at a time. For now, we set `use = "complete.obs"` to compute the correlation using only observations with complete data (i.e. no missing values):

```
cor(msleep$sleep_total, msleep$sleep_rem, use = "complete.obs")
```

2.6.2 Categorical data

Some of the variables, like `vore` (feeding behaviour) and `conservation` (conservation status) are *categorical* rather than *numerical*. It therefore makes no sense to compute means or largest values. For categorical variables (often called *factors* in R), we can instead create a table showing the frequencies of different categories using `table`:

```
table(msleep$vore)
```

To instead show the proportion of different categories, we can apply `proportions` to the table that we just created:

```
proportions(table(msleep$vore))
```

The `table` function can also be used to construct a cross table that shows the counts for different combinations of two categorical variables:

```
# Counts:
table(msleep$vore, msleep$conservation)

# Proportions, per row:
proportions(table(msleep$vore, msleep$conservation),
            margin = 1)

# Proportions, per column:
proportions(table(msleep$vore, msleep$conservation),
            margin = 2)
```

~

Exercise 2.10. Load `ggplot2` using `library(ggplot2)` if you have not already done so. Then do the following:

1. View the documentation for the `diamonds` data and read about different the variables.
2. Check the data structures: how many observations and variables are there and what type of variables (numeric, categorical, etc.) are there?
3. Compute summary statistics (means, median, min, max, counts for categorical variables). Are there any missing values?

2.7 Plotting numerical data

There are several different approaches to creating plots with R. In this book, we will mainly focus on creating plots using the `ggplot2` package, which allows us to create good-looking plots using the so-called *grammar of graphics*. The grammar of graphics is a set of structural rules that helps us establish a language for graphics. The beauty of this is that (almost) all plots will be created with functions that all follow the same logic, or grammar. That way, we don't have to learn new arguments for each new plot. You can compare this to the problems we encountered when we wanted to ignore `NA` values when computing descriptive statistics - `mean` required the argument `na.rm` whereas `cor` required the argument `use`. By using a common grammar for all plots, we reduce the number of arguments that we need to learn.

The three key components to grammar of graphics plots are:

- **Data**: the observations in your dataset,
- **Aesthetics**: mappings from the data to visual properties (like axes and sizes of geometric objects), and
- **Geoms**: geometric objects, e.g. lines, representing what you see in the plot.

When we create plots using `ggplot2`, we must define what data, aesthetics and geoms to use. If that sounds a bit strange, it will hopefully become a lot clearer once we have a look at some examples. To begin with, we will illustrate how this works by visualising some continuous variables in the `msleep` data.

2.7.1 Our first plot

As a first example, let's make a scatterplot by plotting the total sleep time of an animal against the REM sleep time of an animal.

Using base R, we simply do a call to the `plot` function in a way that is analogous to how we'd use e.g. `cor`:

```
plot(msleep$sleep_total, msleep$sleep_rem)
```

The code for doing this using `ggplot2` is more verbose:

```
library(ggplot2)
ggplot(msleep, aes(x = sleep_total, y = sleep_rem)) + geom_point()
```

The code consists of three parts:

- **Data**: given by the first argument in the call to `ggplot`: `msleep`
- **Aesthetics**: given by the second argument in the `ggplot` call: `aes`, where we map `sleep_total` to the x-axis and `sleep_rem` to the y-axis.
- **Geoms**: given by `geom_point`, meaning that the observations will be represented by points.

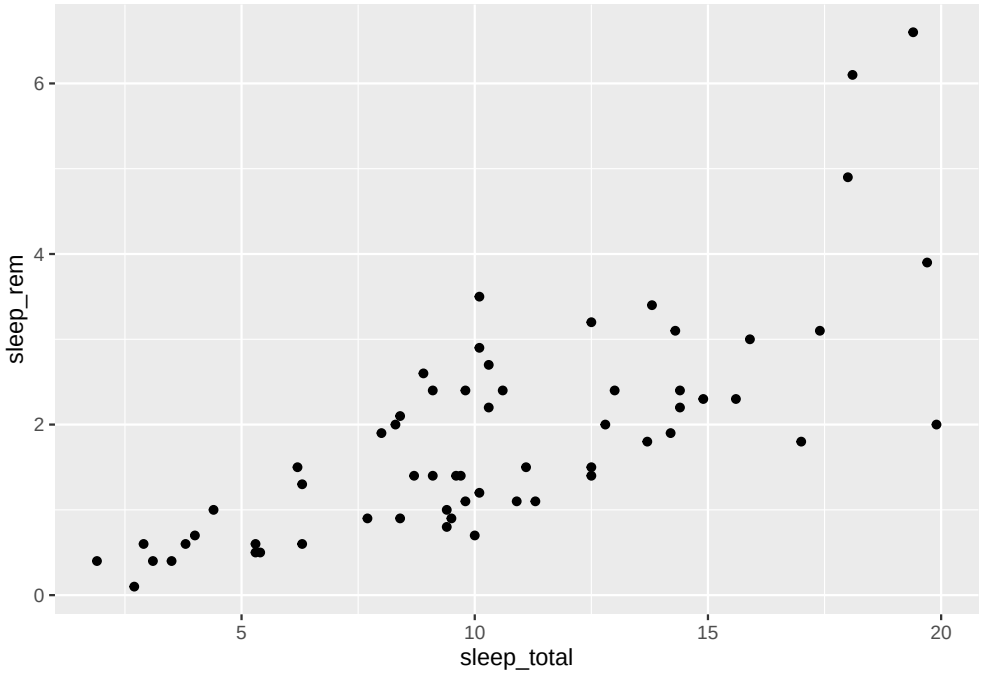

Figure 2.3: A scatterplot of mammal sleeping times.

At this point you may ask why on earth anyone would ever want to use `ggplot2` code for creating plots. It's a valid question. The base R code looks simpler, and is consistent with other functions that we've seen. The `ggplot2` code looks… different. This is because it uses the *grammar of graphics*, which in many ways is a language of its own, different from how we otherwise work with R.

But, the plot created using `ggplot2` also looked different. It used filled circles instead of empty circles for plotting the points, and had a grid in the background. In both base R graphics and `ggplot2` we can changes these settings, and many others. We can create something similar to the `ggplot2` plot using base R as follows, using the `pch` argument and the `grid` function:

```
plot(msleep$sleep_total, msleep$sleep_rem, pch = 16)
grid()
```

Some people prefer the look and syntax of base R plots, while others argue that `ggplot2` graphics has a prettier default look. I can sympathise with both groups. Some types of plots are easier to create using base R, and some are easier to create using ggplot2. I like base R graphics for their simplicity, and prefer them for quick-and-dirty visualisations as well as for more elaborate graphs where I want to combine many different components. For everything in between, including exploratory data analysis where graphics are used to explore and understand datasets, I prefer `ggplot2`. In this book, we'll use base graphics for some quick-and-dirty plots, but put more emphasis on `ggplot2` and how it can be used to explore data.

The syntax used to create the `ggplot2` scatterplot was in essence `ggplot(data, aes) + geom`. All plots created using `ggplot2` follow this pattern, regardless of whether they are scatterplots, bar charts or something else. The plus sign in `ggplot(data, aes) + geom` is important, as it implies that we can add more geoms to the plot, for instance a trend line, and perhaps other things as well. We will return to that shortly.

Unless the user specifies otherwise, the first two arguments to `aes` will always be mapped to the `x` and `y` axes, meaning that we can simplify the code above by removing the `x =` and `y =` bits (at the cost of a slight reduction in readability). Moreover, it is considered good style to insert a line break after the `+` sign. The resulting code is:

```
ggplot(msleep, aes(sleep_total, sleep_rem)) +
      geom_point()
```

Note that this does not change the plot in any way - the difference is merely in the style of the code.

~

Exercise 2.11. Create a scatterplot with total sleeping time along the x-axis and time awake along the y-axis (using the `msleep` data). What pattern do you see? Can you explain it?

2.7.2 Colours, shapes and axis labels

You now know how to make scatterplots, but if you plan to show your plot to someone else, there are probably a few changes that you'd like to make. For instance, it's usually a good idea to change the label for the x-axis from the variable name "sleep_total" to something like "Total sleep time (h)". This is done by using the `+` sign again, adding a call to `xlab` to the plot:

```
ggplot(msleep, aes(sleep_total, sleep_rem)) +
      geom_point() +
      xlab("Total sleep time (h)")
```

Note that the plus signs must be placed at the end of a row rather than at the beginning. To change the y-axis label, add `ylab` instead.

To change the colour of the points, you can set the colour in `geom_point`:

```
ggplot(msleep, aes(sleep_total, sleep_rem)) +
      geom_point(colour = "red") +
      xlab("Total sleep time (h)")
```

In addition to `"red"`, there are a few more colours that you can choose from. You can run `colors()` in the Console to see a list of the 657 colours that have names in R (examples of which include `"papayawhip"`, `"blanchedalmond"`, and `"cornsilk4"`), or use colour hex codes like `"#FF5733"`.

Alternatively, you may want to use the colours of the point to separate different categories. This is done by adding a `colour` argument to `aes`, since you are now mapping a data variable to a visual property. For instance, we can use the variable `vore` to show differences between herbivores, carnivores and omnivores:

```
ggplot(msleep, aes(sleep_total, sleep_rem, colour = vore)) +
      geom_point() +
      xlab("Total sleep time (h)")
```

What happens if we use a continuous variable, such as the sleep cycle length `sleep_cycle` to set the colour?

```
ggplot(msleep, aes(sleep_total, sleep_rem, colour = sleep_cycle)) +
      geom_point() +
      xlab("Total sleep time (h)")
```

You'll learn more about customising colours (and other parts) of your plots in Section 4.2.

~

Exercise 2.12. Using the `diamonds` data, do the following:

1. Create a scatterplot with carat along the x-axis and price along the y-axis. Change the x-axis label to read "Weight of the diamond (carat)" and the y-axis label to "Price (USD)". Use `cut` to set the colour of the points.

2. Try adding the argument `alpha = 1` to `geom_point`, i.e. `geom_point(alpha = 1)`. Does anything happen? Try changing the `1` to `0.5` and `0.25` and see how that affects the plot.

Exercise 2.13. Similar to how you changed the colour of the points, you can also change their size and shape. The arguments for this are called `size` and `shape`.

1. Change the scatterplot from Exercise 2.12 so that diamonds with different cut qualities are represented by different shapes.

2. Then change it so that the size of each point is determined by the diamond's length, i.e. the variable `x`.

2.7.3 Axis limits and scales

Next, assume that we wish to study the relationship between animals' brain sizes and their total sleep time. We create a scatterplot using:

```
ggplot(msleep, aes(brainwt, sleep_total, colour = vore)) +
      geom_point() +
      xlab("Brain weight") +
      ylab("Total sleep time")
```

There are two animals with brains that are much heavier than the rest (African elephant and Asian elephant). These outliers distort the plot, making it difficult to spot any patterns. We can try changing the x-axis to only go from 0 to 1.5 by adding `xlim` to the plot, to see if that improves it:

```
ggplot(msleep, aes(brainwt, sleep_total, colour = vore)) +
      geom_point() +
      xlab("Brain weight") +
      ylab("Total sleep time") +
      xlim(0, 1.5)
```

This is slightly better, but we still have a lot of points clustered near the y-axis, and some animals are now missing from the plot. If instead we wished to change the limits of the y-axis, we would have used `ylim` in the same fashion.

Another option is to resacle the x-axis by applying a log transform to the brain weights, which we can do directly in `aes`:

```
ggplot(msleep, aes(log(brainwt), sleep_total, colour = vore)) +
      geom_point() +
      xlab("log(Brain weight)") +
      ylab("Total sleep time")
```

This is a better-looking scatterplot, with a weak declining trend. We didn't have to remove the outliers (the elephants) to create it, which is good. The downside is that the x-axis now has become difficult to interpret. A third option that mitigates this is to add `scale_x_log10` to the plot, which changes the scale of the x-axis to a $\log_{10}$ scale (which increases interpretability because the values shown at the ticks still are on the original x-scale).

```
ggplot(msleep, aes(brainwt, sleep_total, colour = vore)) +
      geom_point() +
      xlab("Brain weight (logarithmic scale)") +
      ylab("Total sleep time") +
      scale_x_log10()
```

~

Exercise 2.14. Using the `msleep` data, create a plot of log-transformed body weight versus log-transformed brain weight. Use total sleep time to set the colours of the points. Change the text on the axes to something informative.

2.7.4 Comparing groups

We frequently wish to make visual comparison of different groups. One way to display differences between groups in plots is to use *facetting*, i.e. to create a grid of plots corresponding to the different groups. For instance, in our plot of animal brain weight versus total sleep time, we may wish to separate the different feeding behaviours (omnivores, carnivores, etc.) in the `msleep` data using facetting instead of different coloured points. In `ggplot2` we do this by adding a call to `facet_wrap` to the plot:

```
ggplot(msleep, aes(brainwt, sleep_total)) +
      geom_point() +
      xlab("Brain weight (logarithmic scale)") +
      ylab("Total sleep time") +
      scale_x_log10() +
      facet_wrap(~ vore)
```

Note that the x-axes and y-axes of the different plots in the grid all have the same

scale and limits.

~

Exercise 2.15. Using the `diamonds` data, do the following:

1. Create a scatterplot with `carat` along the x-axis and `price` along the y-axis, facetted by `cut`.
2. Read the documentation for `facet_wrap` (`?facet_wrap`). How can you change the number of rows in the plot grid? Create the same plot as in part 1, but with 5 rows.

2.7.5 Boxplots

Another option for comparing groups is boxplots (also called box-and-whiskers plots). Using `ggplot2`, we create boxplots for animal sleep times, grouped by feeding behaviour, with `geom_boxplot`. Using base R, we use the `boxplot` function instead:

```
# Base R:
boxplot(sleep_total ~ vore, data = msleep)

# ggplot2:
ggplot(msleep, aes(vore, sleep_total)) +
      geom_boxplot()
```

The boxes visualise important descriptive statistics for the different groups, similar to what we got using `summary`:

- *Median*: the thick black line inside the box.
- *First quartile*: the bottom of the box.
- *Third quartile*: the top of the box.
- *Minimum*: the end of the line ("whisker") that extends from the bottom of the box.
- *Maximum*: the end of the line that extends from the top of the box.
- *Outliers*: observations that deviate too much[12] from the rest are shown as separate points. These outliers are not included in the computation of the median, quartiles and the extremes.

Note that just as for a scatterplot, the code consists of three parts:

- **Data**: given by the first argument in the call to `ggplot`: `msleep`

[12] In this case, *too much* means that they are more than 1.5 times the height of the box away from the edges of the box.

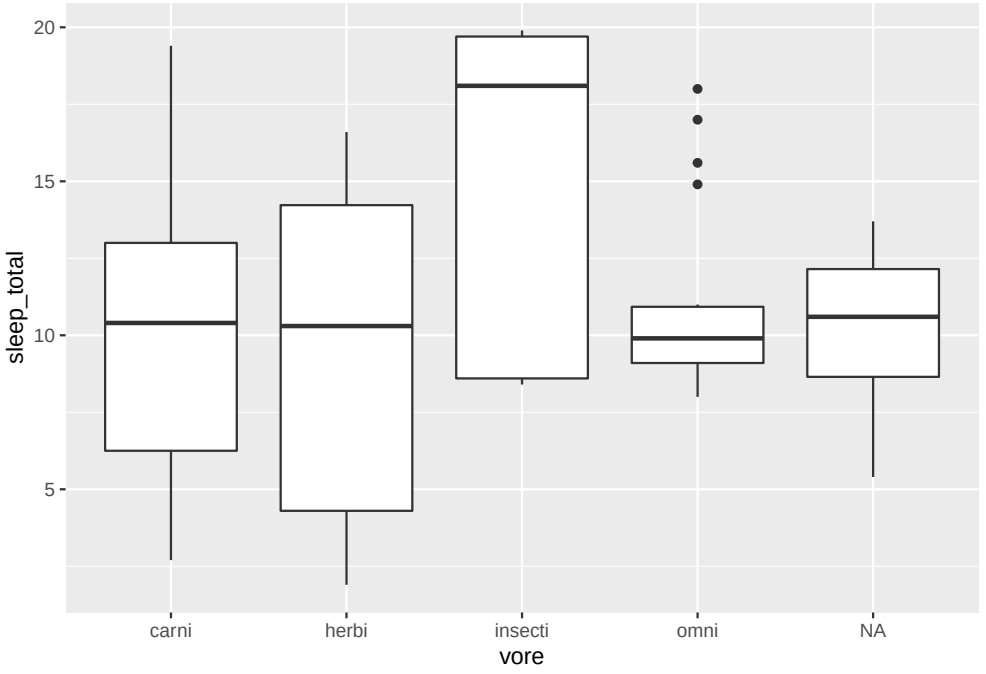

Figure 2.4: Boxplots showing mammal sleeping times.

- **Aesthetics**: given by the second argument in the `ggplot` call: `aes`, where we map the group variable `vore` to the x-axis and the numerical variable `sleep_total` to the y-axis.
- **Geoms**: given by `geom_boxplot`, meaning that the data will be visualised with boxplots.

~

Exercise 2.16. Using the `diamonds` data, do the following:

1. Create boxplots of diamond prices, grouped by `cut`.
2. Read the documentation for `geom_boxplot`. How can you change the colours of the boxes and their outlines?
3. Replace `cut` by `reorder(cut, price, median)` in the plot's aestethics. What does `reorder` do? What is the result?
4. Add `geom_jitter(size = 0.1, alpha = 0.2)` to the plot. What happens?

2.7.6 Histograms

To show the distribution of a continuous variable, we can use a histogram, in which the data is split into a number of bins and the number of observations in each bin is shown by a bar. The `ggplot2` code for histograms follows the same pattern as other plots, while the base R code uses the `hist` function:

```
# Base R:
hist(msleep$sleep_total)

# ggplot2:
ggplot(msleep, aes(sleep_total)) +
      geom_histogram()
```

As before, the three parts in the `ggplot2` code are:

- **Data**: given by the first argument in the call to `ggplot`: `msleep`
- **Aesthetics**: given by the second argument in the `ggplot` call: `aes`, where we map `sleep_total` to the x-axis.
- **Geoms**: given by `geom_histogram`, meaning that the data will be visualised by a histogram.

~

Exercise 2.17. Using the `diamonds` data, do the following:

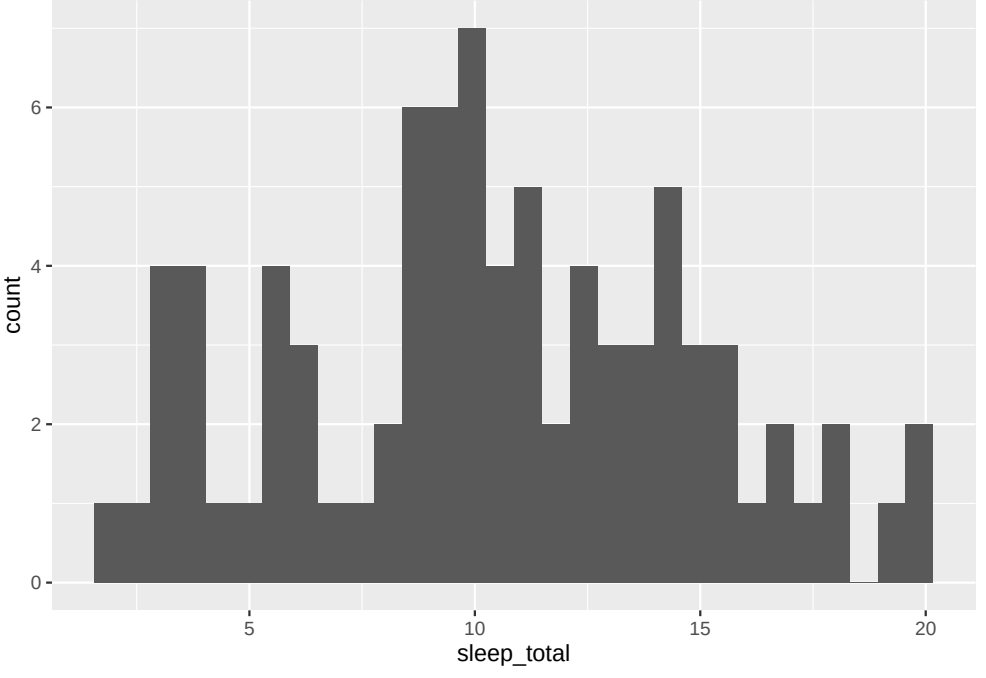

Figure 2.5: A histogram for mammal sleeping times.

1. Create a histogram of diamond prices.
2. Create histograms of diamond prices for different cuts, using facetting.
3. Add a suitable argument to `geom_histogram` to add black outlines around the bars[13].

2.8 Plotting categorical data

When visualising categorical data, we typically try to show the counts, i.e. the number of observations, for each category. The most common plot for this type of data is the bar chart.

2.8.1 Bar charts

Bar charts are discrete analogues to histograms, where the category counts are represented by bars. The code for creating them is:

```
# Base R
barplot(table(msleep$vore))

# ggplot2
ggplot(msleep, aes(vore)) +
      geom_bar()
```

As always, the three parts in the `ggplot2` code are:

- **Data**: given by the first argument in the call to `ggplot`: `msleep`
- **Aesthetics**: given by the second argument in the `ggplot` call: `aes`, where we map `vore` to the x-axis.
- **Geoms**: given by `geom_bar`, meaning that the data will be visualised by a bar chart.

To create a stacked bar chart using `ggplot2`, we use map all groups to the same value on the x-axis and then map the different groups to different colours. This can be done as follows:

```
ggplot(msleep, aes(factor(1), fill = vore)) +
      geom_bar()
```

~

Exercise 2.18. Using the `diamonds` data, do the following:

[13] Personally, I don't understand why anyone would ever plot histograms without outlines!

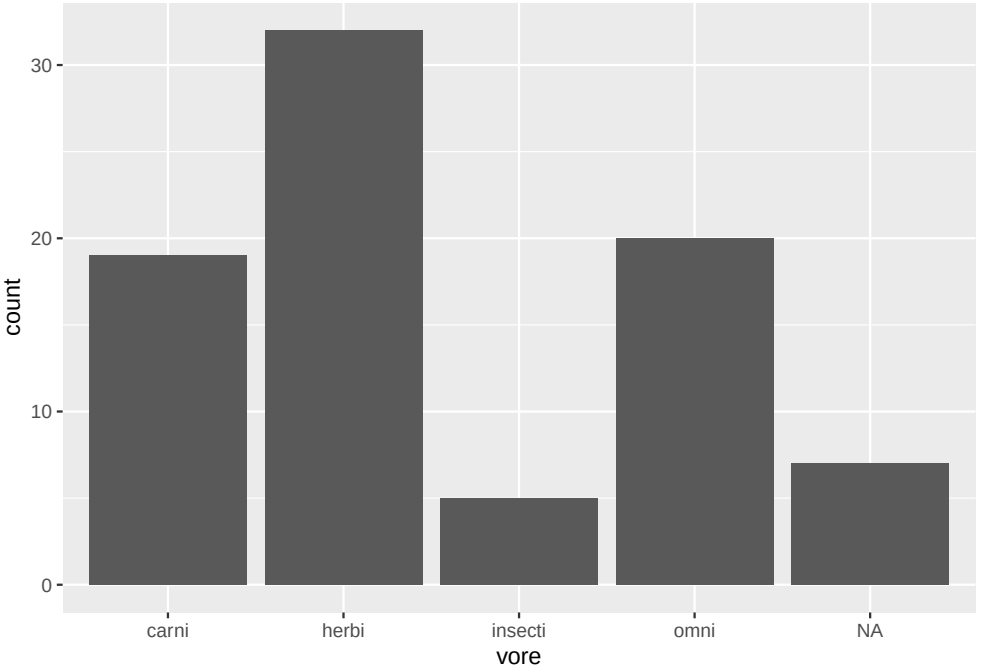

Figure 2.6: A bar chart for the mammal sleep data.

1. Create a bar chart of diamond cuts.
2. Add different colours to the bars by adding a `fill` argument to `geom_bar`.
3. Check the documentation for `geom_bar`. How can you decrease the width of the bars?
4. Return to the code you used for part 1. Add `fill = clarity` to the `aes`. What happens?
5. Next, add `position = "dodge"` to `geom_bar`. What happens?
6. Return to the code you used for part 1. Add `coord_flip()` to the plot. What happens?

2.9 Saving your plot

When you create a `ggplot2` plot, you can save it as a plot object in R:

```
library(ggplot2)
myPlot <- ggplot(msleep, aes(sleep_total, sleep_rem)) +
      geom_point()
```

To plot a saved plot object, just write its name:

```
myPlot
```

If you like, you can add things to the plot, just as before:

```
myPlot + xlab("I forgot to add a label!")
```

To save your plot object as an image file, use `ggsave`. The `width` and `height` arguments allows us to control the size of the figure (in inches, unless you specify otherwise using the `units` argument).

```
ggsave("filename.pdf", myPlot, width = 5, height = 5)
```

If you don't supply the name of a plot object, `ggsave` will save the last `ggplot2` plot you created.

In addition to pdf, you can save images e.g. as jpg, tif, eps, svg, and png files, simply by changing the file extension in the filename. Alternatively, graphics from both base R and `ggplot2` can be saved using the `pdf` and `png` functions, using `dev.off` to mark the end of the file:

```
pdf("filename.pdf", width = 5, height = 5)
myPlot
dev.off()

png("filename.png", width = 500, height = 500)
```

```
plot(msleep$sleep_total, msleep$sleep_rem)
dev.off()
```

Note that you also can save graphics by clicking on the Export button in the Plots panel in RStudio. Using code to save your plot is usually a better idea, because of reproducibility. At some point you'll want to go back and make changes to an old figure, and that will be much easier if you already have the code to export the graphic.

~

Exercise 2.19. Do the following:

1. Create a plot object and save it as a 4 by 4 inch png file.
2. When preparing images for print, you may want to increase their resolution. Check the documentation for `ggsave`. How can you increase the resolution of your png file to 600 dpi?

You've now had a first taste of graphics using R. We have however only scratched the surface, and will return to the many uses of statistical graphics in Chapter 4.

2.10 Troubleshooting

Every now and then R will throw an error message at you. Sometimes these will be informative and useful, as in this case:

```
age <- c(28, 48, 47, 71, 22, 80, 48, 30, 31)
means(age)
```

where R prints:

```
> means(age)
Error in means(age) : could not find function "means"
```

This tells us that the function that we are trying to use, `means` does not exist. There are two possible reasons for this: either we haven't loaded the package in which the function exists, or we have misspelt the function name. In our example the latter is true, the function that we really wanted to use was of course `mean` and not `means`.

At other times interpreting the error message seems insurmountable, like in these examples:

```
Error in if (str_count(string = f[[j]], pattern = \"\\\\S+\") == 1) { :
  \n  argument is of length zero
```

and

```
Error in if (requir[y] &gt; supply[x]) { : \nmissing value where
  TRUE/FALSE needed
```

When you encounter an error message, I recommend following these steps:

1. Read the error message carefully and try to decipher it. Have you seen it before? Does it point to a particular variable or function? Check Section 11.2 of this book, which deals with common error messages in R.
2. Check your code. Have you misspelt any variable or function names? Are there missing brackets, strange commas or invalid characters?
3. Copy the error message and do a web search using the message as your search term. It is more than likely that somebody else has encountered the same problem, and that you can find a solution to it online. This is a great shortcut for finding solutions to your problem. In fact, **this may well be the single most important tip in this entire book.**
4. Read the documentation for the function causing the error message, and look at some examples of how to use it (both in the documentation and online, e.g. in blog posts). Have you used it correctly?
5. Use the debugging tools presented in Chapter 11, or try to simplify the example that you are working with (e.g. removing parts of the analysis or the data) and see if that removes the problem.
6. If you still can't find a solution, post a question at a site like Stack Overflow or the RStudio community forums. Make sure to post your code and describe the context in which the error message appears. If at all possible, post a reproducible example, i.e. a piece of code that others can run, that causes the error message. This will make it a lot easier for others to help you.

Chapter 3

Transforming, summarising, and analysing data

Most datasets are stored as tables, with rows and columns. In this chapter we'll see how you can import and export such data, and how it is stored in R. We'll also discuss how you can transform, summarise, and analyse your data.

After working with the material in this chapter, you will be able to use R to:

- Distinguish between different data types,
- Import data from Excel spreadsheets and csv text files,
- Compute descriptive statistics for subgroups in your data,
- Find interesting points in your data,
- Add new variables to your data,
- Modify variables in your data,
- Remove variables from your data,
- Save and export your data,
- Work with RStudio projects,
- Run t-tests and fit linear models,
- Use `%>%` pipes to chain functions together.

The chapter ends with a discussion of ethical guidelines for statistical work.

3.1 Data frames and data types

3.1.1 Types and structures

We have already seen that different kinds of data require different kinds of statistical methods. For numeric data we create boxplots and compute means, but for categor-

ical data we don't. Instead we produce bar charts and display the data in tables. It is no surprise then, that what R also treats different kinds of data differently.

In programming, a variable's_data type_ describes what kind of object is assigned to it. We can assign many different types of objects to the variable `a`: it could for instance contain a number, text, or a data frame. In order to treat `a` correctly, R needs to know what data type its assigned object has. In some programming languages, you have to explicitly state what data type a variable has, but not in R. This makes programming R simpler and faster, but can cause problems if a variable turns out to have a different data type than what you thought[1].

R has six basic data types. For most people, it suffices to know about the first three in the list below:

- `numeric`: numbers like `1` and `16.823` (sometimes also called `double`).
- `logical`: true/false values (boolean): either `TRUE` or `FALSE`.
- `character`: text, e.g. `"a"`, `"Hello! I'm Ada."` and `"name@domain.com"`.
- `integer`: integer numbers, denoted in R by the letter `L`: `1L`, `55L`.
- `complex`: complex numbers, like `2+3i`. Rarely used in statistical work.
- `raw`: used to hold raw bytes. Don't fret if you don't know what that means. You can have a long and meaningful career in statistics, data science, or pretty much any other field without ever having to worry about raw bytes. We won't discuss `raw` objects again in this book.

In addition, these can be combined into special data types sometimes called *data structures*, examples of which include vectors and data frames. Important data structures include `factor`, which is used to store categorical data, and the awkwardly named `POSIXct` which is used to store date and time data.

To check what type of object a variable is, you can use the `class` function:

```
x <- 6
y <- "Scotland"
z <- TRUE

class(x)
class(y)
class(z)
```

What happens if we use `class` on a vector?

```
numbers <- c(6, 9, 12)
class(numbers)
```

`class` returns the data type of the elements of the vector. So what happens if we put objects of different type together in a vector?

[1] And the subsequent troubleshooting makes programming R more difficult and slower.

```
all_together <- c(x, y, z)
all_together
class(all_together)
```

In this case, R has coerced the objects in the vector to all be of the same type. Sometimes that is desirable, and sometimes it is not. The lesson here is to be careful when you create a vector from different objects. We'll learn more about coercion and how to change data types in Section 5.1.

3.1.2 Types of tables

The basis for most data analyses in R are data frames: spreadsheet-like tables with rows and columns containing data. You encountered some data frames in the previous chapter. Have a quick look at them to remind yourself of what they look like:

```
# Bookstore example
age <- c(28, 48, 47, 71, 22, 80, 48, 30, 31)
purchase <- c(20, 59, 2, 12, 22, 160, 34, 34, 29)
bookstore <- data.frame(age, purchase)
View(bookstore)

# Animal sleep data
library(ggplot2)
View(msleep)

# Diamonds data
View(diamonds)
```

Notice that all three data frames follow the same format: each column represents a *variable* (e.g. age) and each row represents an *observation* (e.g. an individual). This is the standard way to store data in R (as well as the standard format in statistics in general). In what follows, we will use the terms column and variable interchangeably, to describe the columns/variables in a data frame.

This kind of table can be stored in R as different types of objects - that is, in several different ways. As you'd expect, the different types of objects have different properties and can be used with different functions. Here's the run-down of four common types:

- `matrix`: a table where all columns must contain objects of the same type (e.g. all `numeric` or all `character`). Uses less memory than other types and allows for much faster computations, but is difficult to use for certain types of data manipulation, plotting and analyses.
- `data.frame`: the most common type, where different columns can contain different types (e.g. one `numeric` column, one `character` column).
- `data.table`: an enhanced version of `data.frame`.
- `tbl_df` ("tibble"): another enhanced version of `data.frame`.

First of all, in most cases it doesn't matter which of these four that you use to store your data. In fact, they all look similar to the user. Have a look at the following datasets (`WorldPhones` and `airquality` come with base R):

```
# First, an example of data stored in a matrix:
?WorldPhones
class(WorldPhones)
View(WorldPhones)

# Next, an example of data stored in a data frame:
?airquality
class(airquality)
View(airquality)

# Finally, an example of data stored in a tibble:
library(ggplot2)
?msleep
class(msleep)
View(msleep)
```

That being said, in some cases it *really* matters which one you use. Some functions require that you input a matrix, while others may break or work differently from what was intended if you input a tibble instead of an ordinary data frame. Luckily, you can convert objects into other types:

```
WorldPhonesDF <- as.data.frame(WorldPhones)
class(WorldPhonesDF)

airqualityMatrix <- as.matrix(airquality)
class(airqualityMatrix)
```

~

Exercise 3.1. The following tasks are all related to data types and data structures:

1. Create a text variable using e.g. `a <- "A rainy day in Edinburgh"`. Check that it gets the correct type. What happens if you use single quotes marks instead of double quotes when you create the variable?
2. What data types are the sums `1 + 2`, `1L + 2` and `1L + 2L`?
3. What happens if you add a `numeric` to a `character`, e.g. `"Hello" + 1`?
4. What happens if you perform mathematical operations involving a `numeric` and a `logical`, e.g. `FALSE * 2` or `TRUE + 1`?

Exercise 3.2. What do the functions `ncol`, `nrow`, `dim`, `names`, and `row.names` return when applied to a data frame?

Exercise 3.3. `matrix` tables can be created from vectors using the function of the same name. Using the vector `x <- 1:6` use `matrix` to create the following matrices:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

and

$$\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}.$$

Remember to check `?matrix` to find out how to set the dimensions of the matrix, and how it is filled with the numbers from the vector!

3.2 Vectors in data frames

In the next few sections, we will explore the `airquality` dataset. It contains daily air quality measurements from New York during a period of five months:

- `Ozone`: mean ozone concentration (ppb),
- `Solar.R`: solar radiation (Langley),
- `Wind`: average wind speed (mph),
- `Temp`: maximum daily temperature in degrees Fahrenheit,
- `Month`: numeric month (May=5, June=6, and so on),
- `Day`: numeric day of the month (1-31).

There are lots of things that would be interesting to look at in this dataset. What was the mean temperature during the period? Which day was the hottest? Which was the windiest? What days were the temperature more than 90 degrees Fahrenheit? To answer these questions, we need to be able to access the vectors inside the data frame. We also need to be able to quickly and automatically screen the data in order to find interesting observations (e.g. the hottest day)

3.2.1 Accessing vectors and elements

In Section 2.6, we learned how to compute the mean of a vector. We also learned that to compute the mean of a vector *that is stored inside a data frame*[2] we could use a dollar sign: `data_frame_name$vector_name`. Here is an example with the `airquality` data:

[2]This works regardless of whether this is a regular `data.frame`, a `data.table` or a tibble.

```
# Extract the Temp vector:
airquality$Temp

# Compute the mean temperature:
mean(airquality$Temp)
```

If we want to grab a particular element from a vector, we must use its *index* within square brackets: `[index]`. The first element in the vector has index 1, the second has index 2, the third index 3, and so on. To access the fifth element in the `Temp` vector in the `airquality` data frame, we can use:

```
airquality$Temp[5]
```

The square brackets can also be applied directly to the data frame. The syntax for this follows that used for matrices in mathematics: `airquality[i, j]` means the element at the i:th row and j:th column of `airquality`. We can also leave out either `i` or `j` to extract an entire row or column from the data frame. Here are some examples:

```
# First, we check the order of the columns:
names(airquality)
# We see that Temp is the 4th column.

airquality[5, 4]     # The 5th element from the 4th column,
                     # i.e. the same as airquality$Temp[5]
airquality[5,]       # The 5th row of the data
airquality[, 4]      # The 4th column of the data, like airquality$Temp
airquality[[4]]      # The 4th column of the data, like airquality$Temp
airquality[, c(2, 4, 6)] # The 2nd, 4th and 6th columns of the data
airquality[, -2]     # All columns except the 2nd one
airquality[, c("Temp", "Wind")] # The Temp and Wind columns
```

~

Exercise 3.4. The following tasks all involve using the the `[i, j]` notation for extracting data from data frames:

1. Why does `airquality[, 3]` not return the third row of `airquality`?
2. Extract the first five rows from `airquality`. *Hint:* a fast way of creating the vector `c(1, 2, 3, 4, 5)` is to write `1:5`.
3. Compute the correlation between the `Temp` and `Wind` vectors of `airquality` without refering to them using `$`.
4. Extract all columns from `airquality` *except* `Temp` and `Wind`.

3.2.2 Use your dollars

The $ operator can be used not just to extract data from a data frame, but also to manipulate it. Let's return to our `bookstore` data frame, and see how we can make changes to it using the dollar sign.

```
age <- c(28, 48, 47, 71, 22, 80, 48, 30, 31)
purchase <- c(20, 59, 2, 12, 22, 160, 34, 34, 29)
bookstore <- data.frame(age, purchase)
```

Perhaps there was a data entry error - the second customer was actually 18 years old and not 48. We can assign a new value to that element by referring to it in either of two ways:

```
bookstore$age[2] <- 18
# or
bookstore[2, 1] <- 18
```

We could also change an entire column if we like. For instance, if we wish to change the `age` vector to months instead of years, we could use

```
bookstore$age <- bookstore$age * 12
```

What if we want to add another variable to the data, for instance the length of the customers' visits in minutes? There are several ways to accomplish this, one of which involves the dollar sign:

```
bookstore$visit_length <- c(5, 2, 20, 22, 12, 31, 9, 10, 11)
bookstore
```

As you see, the new data has now been added to a new column in the data frame.

~

Exercise 3.5. Use the `bookstore` data frame to do the following:

1. Add a new variable `rev_per_minute` which is the ratio between purchase and the visit length.
2. Oh no, there's been an error in the data entry! Replace the purchase amount for the 80-year old customer with `16`.

3.2.3 Using conditions

A few paragraphs ago, we were asking which was the hottest day in the `airquality` data. Let's find out! We already know how to find the maximum value in the `Temp` vector:

```
max(airquality$Temp)
```

But can we find out which day this corresponds to? We could of course manually go through all 153 days e.g. by using `View(airquality)`, but that seems tiresome and wouldn't even be possible in the first place if we'd had more observations. A better option is therefore to use the function `which.max`:

```
which.max(airquality$Temp)
```

`which.max` returns the index of the observation with the maximum value. If there is more than one observation attaining this value, it only returns the first of these.

We've just used `which.max` to find out that day `120` was the hottest during the period. If we want to have a look at the entire row for that day, we can use

```
airquality[120,]
```

Alternatively, we could place the call to `which.max` inside the brackets. Because `which.max(airquality$Temp)` returns the number `120`, this yields the same result as the previous line:

```
airquality[which.max(airquality$Temp),]
```

Were we looking for the day with the lowest temperature, we'd use `which.min` analogously. In fact, we could use any function or computation that returns an index in the same way, placing it inside the brackets to get the corresponding rows or columns. This is extremely useful if we want to extract observations with certain properties, for instance all days where the temperature was above 90 degrees. We do this using *conditions*, i.e. by giving statements that we wish to be fulfilled.

As a first example of a condition, we use the following, which checks if the temperature exceeds 90 degrees:

```
airquality$Temp > 90
```

For each element in `airquality$Temp` this returns either `TRUE` (if the condition is fulfilled, i.e. when the temperature is greater than 90) or `FALSE` (if the conditions isn't fulfilled, i.e. when the temperature is 90 or lower). If we place the condition inside brackets following the name of the data frame, we will extract only the rows corresponding to those elements which were marked with `TRUE`:

```
airquality[airquality$Temp > 90, ]
```

If you prefer, you can also store the `TRUE` or `FALSE` values in a new variable:

```
airquality$Hot <- airquality$Temp > 90
```

There are several logical operators and functions which are useful when stating conditions in R. Here are some examples:

```
a <- 3
b <- 8

a == b      # Check if a equals b
a > b       # Check if a is greater than b
a < b       # Check if a is less than b
a >= b      # Check if a is equal to or greater than b
a <= b      # Check if a is equal to or less than b
a != b      # Check if a is not equal to b
is.na(a)    # Check if a is NA
a %in% c(1, 4, 9) # Check if a equals at least one of 1, 4, 9
```

When checking a conditions for all elements in a vector, we can use `which` to get the indices of the elements that fulfill the condition:

```
which(airquality$Temp > 90)
```

If we want to know if all elements in a vector fulfill the condition, we can use `all`:

```
all(airquality$Temp > 90)
```

In this case, it returns `FALSE`, meaning that not all days had a temperature above 90 (phew!). Similarly, if we wish to know whether *at least one* day had a temperature above 90, we can use `any`:

```
any(airquality$Temp > 90)
```

To find how many elements that fulfill a condition, we can use `sum`:

```
sum(airquality$Temp > 90)
```

Why does this work? Remember that `sum` computes the sum of the elements in a vector, and that when `logical` values are used in computations, they are treated as `0` (`FALSE`) or `1` (`TRUE`). Because the condition returns a vector of `logical` values, the sum of them becomes the number of 1's - the number of `TRUE` values - i.e. the number of elements that fulfill the condition.

To find the proportion of elements that fulfill a condition, we can count how many elements fulfill it and then divide by how many elements are in the vector. This is exactly what happens if we use `mean`:

```
mean(airquality$Temp > 90)
```

Finally, we can combine conditions by using the logical operators `&` (AND), `|` (OR), and, less frequently, `xor` (exclusive or, XOR). Here are some examples:

```
a <- 3
b <- 8
```

```
# Is a less than b and greater than 1?
a < b & a > 1

# Is a less than b and equal to 4?
a < b & a == 4

# Is a less than b and/or equal to 4?
a < b | a == 4

# Is a equal to 4 and/or equal to 5?
a == 4 | a == 5

# Is a less than b XOR equal to 4?
# I.e. is one and only one of these satisfied?
xor(a < b, a == 4)
```

~

Exercise 3.6. The following tasks all involve checking conditions for the `airquality` data:

1. Which was the coldest day during the period?
2. How many days was the wind speed greater than 17 mph?
3. How many missing values are there in the `Ozone` vector?
4. How many days are there for which the temperature was below 70 and the wind speed was above 10?

Exercise 3.7. The function `cut` can be used to create a categorical variable from a numerical variable, by dividing it into categories corresponding to different intervals. Reads its documentation and then create a new categorical variable in the `airquality` data, `TempCat`, which divides `Temp` into the three intervals `(50, 70]`, `(70, 90]`, `(90, 110]`[3].

3.3 Importing data

So far, we've looked at examples of data they either came shipped with base R or `ggplot2`, or simple toy examples that we created ourselves, like `bookstore`. While you can do all your data entry work in R, `bookstore` style, it is much more common

[3]In interval notation, `(50, 70]` means that the interval contains all values between 50 and 70, excluding 50 but including 70; the intervals is *open* on the left but *closed* to the right.

to load data from other sources. Two important types of files are *comma-separated value files*, `.csv`, and Excel spreadsheets, `.xlsx`. `.csv` files are spreadsheets stored as text files - basically Excel files stripped down to the bare minimum - no formatting, no formulas, no macros. You can open and edit them in spreadsheet software like LibreOffice Calc, Google Sheets or Microsoft Excel. Many devices and databases can export data in `.csv` format, making it a commonly used file format that you are likely to encounter sooner rather than later.

3.3.1 Importing csv files

In order to load data from a file into R, you need its *path* - that is, you need to tell R where to find the file. Unless you specify otherwise, R will look for files in its current *working directory*. To see what your current working directory is, run the following code in the Console panel:

```
getwd()
```

In RStudio, your working directory will usually be shown in the Files panel. If you have opened RStudio by opening a `.R` file, the working directory will be the directory in which the file is stored. You can change the working directory by using the function `setwd` or selecting *Session > Set Working Directory > Choose Directory* in the RStudio menu.

Before we discuss paths further, let's look at how you can import data from a file that is in your working directory. The data files that we'll use in examples in this book can be downloaded from the book's web page. They are stored in a zip file (`data.zip`) - open it an copy/extract the files to the folder that is your current working directory. Open `philosophers.csv` with a spreadsheet software to have a quick look at it. Then open it in a text editor (for instance Notepad for Windows, TextEdit for Mac or Gedit for Linux). Note how commas are used to separate the columns of the data:

```
"Name","Description","Born","Deceased","Rating"
"Aristotle","Pretty influential, as philosophers go.",-384,"322 BC",
"4.8"
"Basilides","Denied the existence of incorporeal entities.",-175,
"125 BC",4
"Cercops","An Orphic poet",,,"3.2"
"Dexippus","Neoplatonic!",235,"375 AD","2.7"
"Epictetus","A stoic philosopher",50,"135 AD",5
"Favorinus","Sceptic",80,"160 AD","4.7"
```

Then run the following code to import the data using the `read.csv` function and store it in a variable named `imported_data`:

```
imported_data <- read.csv("philosophers.csv")
```

If you get an error message that says:

```
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
  cannot open file 'philosophers.csv': No such file or directory
```

…it means that `philosophers.csv` is not in your working directory. Either move the file to the right directory (remember, you can use run `getwd()` to see what your working directory is) or change your working directory, as described above.

Now, let's have a look at `imported_data`:

```
View(imported_data)
str(imported_data)
```

The columns `Name` and `Description` both contain text, and have been imported as `character` vectors[4]. The `Rating` column contains numbers with decimals and has been imported as a `numeric` vector. The column `Born` only contain integer values, and has been imported as an `integer` vector. The missing value is represented by an `NA`. The `Deceased` column contains years formatted like `125 BC` and `135 AD`. These have been imported into a `character` vector - because numbers and letters are mixed in this column, R treats is as a text string (in Chapter 5 we will see how we can convert it to numbers or proper dates). In this case, the missing value is represented by an empty string, `""`, rather than by `NA`.

So, what can you do in case you need to import data from a file that is not in your working directory? This is a common problem, as many of us store script files and data files in separate folders (or even on separate drives). One option is to use `file.choose`, which opens a pop-up window that lets you choose which file to open using a graphical interface:

```
imported_data2 <- read.csv(file.choose())
```

A third option is not to write any code at all. Instead, you can import the data using RStudio's graphical interface by choosing *File > Import dataset > From Text (base)* and then choosing `philosophers.csv`. This will generate the code needed to import the data (using `read.csv`) and run it in the Console window.

The latter two solutions work just fine if you just want to open a single file once. But if you want to reuse your code or run it multiple times, you probably don't want to

[4]If you are running an older version of R (specifically, a version older than the 4.0.0 version released in April 2020), the `character` vectors will have been imported as `factor` vectors instead. You can change that behaviour by adding a `stringsAsFactors = FALSE` argument to `read.csv`.

have to click and select your file each time. Instead, you can specify the path to your file in the call to `read.csv`.

3.3.2 File paths

File paths look different in different operating systems. If the user `Mans` has a file named `philosophers.csv` stored in a folder called `MyData` on his desktop, its path on an English-language Windows system would be:

```
C:\Users\Mans\Desktop\MyData\philosophers.csv
```

On a Mac it would be:

```
/Users/Mans/Desktop/MyData/philosophers.csv
```

And on Linux:

```
/home/Mans/Desktop/MyData/philosophers.csv
```

You can copy the path of the file from your file browser: Explorer[5] (Windows), Finder[6] (Mac) or Nautilus/similar[7] (Linux). Once you have copied the path, you can store it in R as a `character` string.

Here's how to do this on Mac and Linux:

```
file_path <- "/Users/Mans/Desktop/MyData/philosophers.csv" # Mac
file_path <- "/home/Mans/Desktop/MyData/philosophers.csv"  # Linux
```

If you're working on a Windows system, file paths are written using backslashes, \, like so:

```
C:\Users\Mans\Desktop\MyData\file.csv
```

You have to be careful when using backslashes in `character` strings in R, because they are used to create special characters (see Section 5.5). If we place the above path in a string, R won't recognise it as a path. Instead we have to reformat it into one of the following two formats:

```
# Windows example 1:
file_path <- "C:/Users/Mans/Desktop/MyData/philosophers.csv"
# Windows example 2:
file_path <- "C:\\Users\\Mans\\Desktop\\MyData\\philosophers.csv"
```

[5] To copy the path, navigate to the file in Explorer. Hold down the Shift key and right-click the file, selecting *Copy as path*.

[6] To copy the path, navigate to the file in Finder and right-click/Control+click/two-finger click on the file. Hold down the Option key, and then select *Copy "file name" as Pathname*.

[7] To copy the path from Nautilus, navigate to the file and press Ctrl+L to show the path, then copy it. If you are using some other file browser or the terminal, my guess is that you're tech-savvy enough that you don't need me to tell you how to find the path of a file.

If you've copied the path to your clipboard, you can also get the path in the second of the formats above by using

```
file_path <- readClipboard()   # Windows example 3
```

Once the path is stored in `file_path`, you can then make a call to `read.csv` to import the data:

```
imported_data <- read.csv(file_path)
```

Try this with your `philosophers.csv` file, to make sure that you know how it works.

Finally, you can read a file directly from a URL, by giving the URL as the file path. Here is an example with data from the WHO Global Tuberculosis Report:

```
# Download WHO tuberculosis burden data:
tb_data <- read.csv("https://tinyurl.com/whotbdata")
```

`.csv` files can differ slightly in how they are formatted - for instance, different symbols can be used to delimit the columns. You will learn how to handle this in the exercises below.

A downside to `read.csv` is that it is very slow when reading large (50 MB or more) csv files. Faster functions are available in add-on packages; see Section 5.7.1. In addition, it is also possible to import data from other statistical software packages such as SAS and SPSS, from other file formats like JSON, and from databases. We'll discuss most of these in Section 5.14

3.3.3 Importing Excel files

One common file format we will discuss right away though - `.xlsx` - Excel spreadsheet files. There are several packages that can be used to import Excel files to R. I like the `openxlsx` package, so let's install that:

```
install.packages("openxlsx")
```

Now, download the `philosophers.xlsx` file from the book's web page and save it in a folder of your choice. Then set `file_path` to the path of the file, just as you did for the `.csv` file. To import data from the Excel file, you can then use:

```
library(openxlsx)
imported_from_Excel <- read.xlsx(file_path)

View(imported_from_Excel)
str(imported_from_Excel)
```

As with `read.csv`, you can replace the file path with `file.choose()` in order to select the file manually.

~

Exercise 3.8. The abbreviation CSV stands for *Comma Separated Values*, i.e. that commas `,` are used to separate the data columns. Unfortunately, the `.csv` format is not standardised, and `.csv` files can use different characters to delimit the columns. Examples include semicolons (`;`) and tabs (multiple spaces, denoted `\t` in strings in R). Moreover, decimal points can be given either as points (`.`) or as commas (`,`). Download the `vas.csv` file from the book's web page. In this dataset, a number of patients with chronic pain have recorded how much pain they experience each day during a period, using the Visual Analogue Scale (VAS, ranging from 0 - no pain - to 10 - worst imaginable pain). Inspect the file in a spreadsheet software and a text editor - check which symbol is used to separate the columns and whether a decimal point or a decimal comma is used. Then set `file_path` to its path and import the data from it using the code below:

```
vas <- read.csv(file_path, sep = ";", dec = ",", skip = 4)

View(vas)
str(vas)
```

1. Why are there two variables named `X` and `X.1` in the data frame?
2. What happens if you remove the `sep = ";"` argument?
3. What happens if you instead remove the `dec = ","` argument?
4. What happens if you instead remove the `skip = 4` argument?
5. What happens if you change `skip = 4` to `skip = 5`?

Exercise 3.9. Download the `projects-email.xlsx` file from the book's web page and have a look at it in a spreadsheet software. Note that it has three sheet: *Projects*, *Email*, and *Contact*.

1. Read the documentation for `read.xlsx`. How can you import the data from the second sheet, *Email*?
2. Some email addresses are repeated more than once. Read the documentation for `unique`. How can you use it to obtain a vector containing the email addresses without any duplicates?

Exercise 3.10. Download the `vas-transposed.csv` file from the book's web page and have a look at it in a spreadsheet software. It is a *transposed* version of `vas.csv`, where rows represent variables and columns represent observations (instead of the other way around, as is the case in data frames in R). How can we import this data into R?

1. Import the data using `read.csv`. What does the resulting data frame look like?
2. Read the documentation for `read.csv`. How can you make it read the row names that can be found in the first column of the `.csv` file?
3. The function `t` can be applied to transpose (i.e. rotate) your data frame. Try it out on your imported data. Is the resulting object what you were looking for? What happens if you make a call to `as.data.frame` with your data after transposing it?

3.4 Saving and exporting your data

In many a case, data manipulation is a huge part of statistical work, and of course you want to be able to save a data frame after manipulating it. There are two options for doing this in R - you can either export the data as e.g. a `.csv` or a `.xlsx` file, or save it in R format as an `.RData` file.

3.4.1 Exporting data

Just as we used the functions `read.csv` and `read.xlsx` to import data, we can use `write.csv` and `write.xlsx` to export it. The code below saves the `bookstore` data frame as a `.csv` file and an `.xlsx` file. Both files will be created in the current working directory. If you wish to store them somewhere else, you can replace the `"bookstore.csv"` bit with a full path, e.g. `"/home/mans/my-business/bookstore.csv"`.

```
# Bookstore example
age <- c(28, 48, 47, 71, 22, 80, 48, 30, 31)
purchase <- c(20, 59, 2, 12, 22, 160, 34, 34, 29)
bookstore <- data.frame(age, purchase)

# Export to .csv:
write.csv(bookstore, "bookstore.csv")

# Export to .xlsx (Excel):
library(openxlsx)
write.xlsx(bookstore, "bookstore.xlsx")
```

3.4.2 Saving and loading R data

Being able to export to different spreadsheet formats is very useful, but sometimes you want to save an object that can't be saved in a spreadsheet format. For instance, you may wish to save a machine learning model that you've created. `.RData` files can be used to store one or more R objects.

To save the objects `bookstore` and `age` in a `.Rdata` file, we can use the `save` function:

```
save(bookstore, age, file = "myData.RData")
```

To save all objects in your environment, you can use `save.image`:

```
save.image(file = "allMyData.RData")
```

When we wish to load the stored objects, we use the `load` function:

```
load(file = "myData.RData")
```

3.5 RStudio projects

It is good practice to create a new folder for each new data analysis project that you are working on, where you store code, data and the output from the analysis. In RStudio you can associate a folder with a Project, which lets you start RStudio with that folder as your working directory. Moreover, by opening another Project you can have several RStudio sessions, each with their separate variables and working directories, running simultaneously.

To create a new Project, click *File > New Project* in the RStudio menu. You then get to choose whether to create a Project associated with a folder that already exists, or to create a Project in a new folder. After you've created the Project, it will be saved as an `.Rproj` file. You can launch RStudio with the Project folder as the working directory by double-clicking the `.Rproj` file. If you already have an active RStudio session, this will open another session in a separate window.

When working in a Project, I recommend that you store your data in a subfolder of the Project folder. You can the use *relative paths* to access your data files, i.e. paths that are relative to you working directory. For instance, if the file `bookstore.csv` is in a folder in your working directory called `Data`, it's relative path is:

```
file_path <- "Data/bookstore.csv"
```

Much simpler that having to write the entire path, isn't it?

If instead your working directory is contained inside the folder where `bookstore.csv` is stored, its relative path would be

```
file_path <- "../bookstore.csv"
```

The beauty of using relative paths is that they are simpler to write, and if you transfer the entire project folder to another computer, your code will still run, because the relative paths will stay the same.

3.6 Running a t-test

R has thousands of functions for running different statistical hypothesis tests. We'll delve deeper into that in Chapter 7, but we'll have a look at one of them right away: `t.test`, which (yes, you guessed it!) can be used to run Student's t-test, which can be used to test whether the mean of two populations are equal.

Let's say that we want to compare the mean sleeping times of carnivores and herbivores, using the `msleep` data. `t.test` takes two vectors as input, corresponding to the measurements from the two groups:

```
library(ggplot2)
carnivores <- msleep[msleep$vore == "carni",]
herbivores <- msleep[msleep$vore == "herbi",]
t.test(carnivores$sleep_total, herbivores$sleep_total)
```

The output contains a lot of useful information, including the p-value (0.53) and a 95 % confidence interval. `t.test` contains a number of useful arguments that we can use to tailor the test to our taste. For instance, we can change the confidence level of the confidence interval (to 90 %, say), use a one-sided alternative hypothesis ("carnivores sleep more than herbivores", i.e. the mean of the first group is *greater* than that of the second group) and perform the test under the assumption of equal variances in the two samples:

```
t.test(carnivores$sleep_total, herbivores$sleep_total,
       conf.level = 0.90,
       alternative = "greater",
       var.equal = TRUE)
```

We'll explore `t.test` and related functions further in Section 7.2.

3.7 Fitting a linear regression model

The `mtcars` data from Henderson and Velleman (1981) has become one of the classic datasets in R, and a part of the initiation rite for new R users is to use the `mtcars` data to fit a linear regression model. The data describes fuel consumption, number of cylinders and other information about cars from the 1970's:

```
?mtcars
View(mtcars)
```

Let's have a look at the relationship between gross horsepower (`hp`) and fuel consumption (`mpg`):

```
library(ggplot2)
ggplot(mtcars, aes(hp, mpg)) +
      geom_point()
```

The relationship doesn't appear to be perfectly linear, but nevertheless, we can try fitting a linear regression model to the data. This can be done using `lm`. We fit a model with `mpg` as the response variable and `hp` as the explanatory variable:

```
m <- lm(mpg ~ hp, data = mtcars)
```

The first argument is a formula, saying that `mpg` is a function of `hp`, i.e.

$$mpg = \beta_0 + \beta_1 \cdot hp.$$

A summary of the model is obtained using `summary`. Among other things, it includes the estimated parameters, p-values and the coefficient of determination R^2.

```
summary(m)
```

We can add the fitted line to the scatterplot by using `geom_abline`, which lets us add a straight line with a given intercept and slope - we take these to be the coefficients from the fitted model, given by `coef`:

```
# Check model coefficients:
coef(m)

# Add regression line to plot:
ggplot(mtcars, aes(hp, mpg)) +
      geom_point() +
      geom_abline(aes(intercept = coef(m)[1], slope = coef(m)[2]),
                  colour = "red")
```

Diagnostic plots for the residuals are obtained using `plot`:

```
plot(m)
```

If we wish to add further variables to the model, we simply add them to the right-hand-side of the formula in the function call:

```
m2 <- lm(mpg ~ hp + wt, data = mtcars)
summary(m2)
```

In this case, the model becomes

$$mpg = \beta_0 + \beta_1 \cdot hp + \beta_2 \cdot wt.$$

There is much more to be said about linear models in R. We'll return to them in Section 8.1.

~

Exercise 3.11. Fit a linear regression model to the `mtcars` data, using `mpg` as the response variable and `hp`, `wt`, `cyl`, and `am` as explanatory variables. Are all four explanatory variables significant?

3.8 Grouped summaries

Being able to compute the mean temperature for the `airquality` data during the entire period is great, but it would be even better if we also had a way to compute it for each month. The `aggregate` function can be used to create that kind of *grouped summary*.

To begin with, let's compute the mean temperature for each month. Using `aggregate`, we do this as follows:

```
aggregate(Temp ~ Month, data = airquality, FUN = mean)
```

The first argument is a formula, similar to what we used for `lm`, saying that we want a summary of `Temp` grouped by `Month`. Similar formulas are used also in other R functions, for instance when building regression models. In the second argument, `data`, we specify in which data frame the variables are found, and in the third, `FUN`, we specify which function should be used to compute the summary.

By default, `mean` returns `NA` if there are missing values. In `airquality`, `Ozone` contains missing values, but when we compute the grouped means the results are not `NA`:

```
aggregate(Ozone ~ Month, data = airquality, FUN = mean)
```

By default, `aggregate` removes `NA` values before computing the grouped summaries.

It is also possible to compute summaries for multiple variables at the same time. For instance, we can compute the standard deviations (using `sd`) of `Temp` and `Wind`, grouped by `Month`:

```
aggregate(cbind(Temp, Wind) ~ Month, data = airquality, FUN = sd)
```

`aggregate` can also be used to count the number of observations in the groups. For instance, we can count the number of days in each month. In order to do so, we put a variable with no `NA` values on the left-hand side in the formula, and use `length`, which returns the length of a vector:

```
aggregate(Temp ~ Month, data = airquality, FUN = length)
```

Another function that can be used to compute grouped summaries is `by`. The results are the same, but the output is not as nicely formatted. Here's how to use it to compute the mean temperature grouped by month:

```
by(airquality$Temp, airquality$Month, mean)
```

What makes `by` useful is that unlike `aggregate` it is easy to use with functions that take more than one variable as input. If we want to compute the correlation between `Wind` and `Temp` grouped by month, we can do that as follows:

```
names(airquality)  # Check that Wind and Temp are in columns 3 and 4
by(airquality[, 3:4], airquality$Month, cor)
```

For each month, this outputs a *correlation matrix*, which shows both the correlation between `Wind` and `Temp` and the correlation of the variables with themselves (which always is 1).

~

Exercise 3.12. Load the VAS pain data `vas.csv` from Exercise 3.8. Then do the following:

1. Compute the mean VAS for each patient.
2. Compute the lowest and highest VAS recorded for each patient.
3. Compute the number of high-VAS days, defined as days where the VAS was at least 7, for each patient.

Exercise 3.13. Install the `datasauRus` package using `install.packages("datasauRus")` (note the capital R!). It contains the dataset `datasaurus_dozen`. Check its structure and then do the following:

1. Compute the mean of `x`, mean of `y`, standard deviation of `x`, standard deviation of `y`, and correlation between `x` and `y`, grouped by `dataset`. Are there any differences between the 12 datasets?
2. Make a scatterplot of `x` against `y` for each dataset (use facetting!). Are there any differences between the 12 datasets?

3.9 Using %>% pipes

Consider the code you used to solve part 1 of Exercise 3.5:

```
bookstore$rev_per_minute <- bookstore$purchase / bookstore$visit_length
```

Wouldn't it be more convenient if you didn't have to write the `bookstore$` part each time? To just say once that you are manipulating `bookstore`, and have R implicitly understand that all the variables involved reside in that data frame? Yes. Yes, it would. Fortunately, R has tools that will let you do just that.

3.9.1 *Ceci n'est pas une pipe*

The `magrittr` package[8] adds a set of tools called *pipes* to R. Pipes are operators that let you improve your code's readability and restructure your code so that it is read from the left to the right instead of from the inside out. Let's start by installing the package:

```
install.packages("magrittr")
```

Now, let's say that we are interested in finding out what the mean wind speed (in m/s rather than mph) on hot days (temperature above 80, say) in the `airquality` data is, aggregated by month. We could do something like this:

```
# Extract hot days:
airquality2 <- airquality[airquality$Temp > 80, ]
# Convert wind speed to m/s:
airquality2$Wind <- airquality2$Wind * 0.44704
# Compute mean wind speed for each month:
hot_wind_means <- aggregate(Wind ~ Month, data = airquality2,
                            FUN = mean)
```

There is nothing wrong with this code per se. We create a copy of `airquality` (because we don't want to change the original data), change the units of the wind speed, and then compute the grouped means. A downside is that we end up with a copy of `airquality` that we maybe won't need again. We could avoid that by putting all the operations inside of `aggregate`:

```
# More compact:
hot_wind_means <-  aggregate(Wind*0.44704 ~ Month,
                            data = airquality[airquality$Temp > 80, ],
                            FUN = mean)
```

The problem with this is that it is a little difficult to follow because we have to read the code from the inside out. When we run the code, R will first extract the hot days, then convert the wind speed to m/s, and then compute the grouped means - so the operations happen in an order that is the opposite of the order in which we wrote them.

`magrittr` introduces a new operator, `%>%`, called a *pipe*, which can be used to chain functions together. Calls that you would otherwise write as

```
new_variable <- function_2(function_1(your_data))
```

can be written as

```
your_data %>% function_1 %>% function_2 -> new_variable
```

[8] Arguably the best-named R package.

so that the operations are written in the order they are performed. Some prefer the former style, which is more like mathematics, but many prefer the latter, which is more like natural language (particularly for those of us who are used to reading from left to right).

Three operations are required to solve the `airquality` wind speed problem:

1. Extract the hot days,
2. Convert the wind speed to m/s,
3. Compute the grouped means.

Where before we used function-less operations like `airquality2$Wind <- airquality2$Wind * 0.44704`, we would now require functions that carried out the same operations if we wanted to solve this problem using pipes.

A function that lets us extract the hot days is `subset`:

```
subset(airquality, Temp > 80)
```

The `magrittr` function `inset` lets us convert the wind speed:

```
library(magrittr)
inset(airquality, "Wind", value = airquality$Wind * 0.44704)
```

And finally, `aggregate` can be used to compute the grouped means. We could use these functions step-by-step:

```
# Extract hot days:
airquality2 <- subset(airquality, Temp > 80)
# Convert wind speed to m/s:
airquality2 <- inset(airquality2, "Wind",
                     value = airquality2$Wind * 0.44704)
# Compute mean wind speed for each month:
hot_wind_means <- aggregate(Wind ~ Month, data = airquality2,
                            FUN = mean)
```

But, because we have functions to perform the operations, we can instead use `%>%` pipes to chain them together in a *pipeline*. Pipes automatically send the output from the previous function as the first argument to the next, so that the data flows from left to right, which make the code more concise. They also let us refer to the output from the previous function as `.`, which saves even more space. The resulting code is:

```
airquality %>%
      subset(Temp > 80) %>%
      inset("Wind", value = .$Wind * 0.44704) %>%
      aggregate(Wind ~ Month, data = ., FUN = mean) ->
      hot_wind_means
```

You can read the `%>%` operator as *then*: take the `airquality` data, *then* subset it,

then convert the `Wind` variable, *then* compute the grouped means. Once you wrap your head around the idea of reading the operations from left to right, this code is arguably clearer and easier to read. Note that we used the right-assignment operator `->` to assign the result to `hot_wind_means`, to keep in line with the idea that the data flows from the left to the right.

3.9.2 Aliases and placeholders

In the remainder of the book, we will use pipes in some situations where they make the code easier to write or read. Pipes don't always make code easier to read though, as can be seen if we use them to compute exp(log(2)):

```
# Standard solution:
exp(log(2))
# magrittr solution:
2 %>% log %>% exp
```

If you need to use binary operators like `+`, `^` and `<`, `magrittr` has a number of *aliases* that you can use. For instance, `add` works as an alias for `+`:

```
x <- 2
exp(x + 2)
x %>% add(2) %>% exp
```

Here are a few more examples:

```
x <- 2
# Base solution;            magrittr solution
exp(x - 2);                 x %>% subtract(2) %>% exp
exp(x * 2);                 x %>% multiply_by(2) %>% exp
exp(x / 2);                 x %>% divide_by(2) %>% exp
exp(x^2);                   x %>% raise_to_power(2) %>% exp
head(airquality[,1:4]);     airquality %>% extract(,1:4) %>% head
airquality$Temp[1:5];       airquality %>%
                              use_series(Temp) %>% extract(1:5)
```

In simple cases like these it is usually preferable to use the base R solution - the point here is that if you need to perform this kind of operation inside a pipeline, the aliases make it easy to do so. For a complete list of aliases, see `?extract`.

If the function does not take the output from the previous function as its first argument, you can use `.` as a placeholder, just as we did in the `airquality` problem. Here is another example:

```
cat(paste("The current time is ", Sys.time())))
Sys.time() %>% paste("The current time is", .) %>% cat
```

If the data only appears inside parentheses, you need to wrap the function in curly

brackets {}, or otherwise %>% will try to pass it as the first argument to the function:

```
airquality %>% cat("Number of rows in data:", nrow(.)) # Doesn't work
airquality %>% {cat("Number of rows in data:", nrow(.))} # Works!
```

In addition to the magrittr pipes, from version 4.1 R also offers a native pipe, |>, which can be used in lieu of %>% without loading any packages. Nevertheless, we'll use %>% pipes in the remainder of the book, partially because they are more commonly used (meaning that you are more likely to encounter them when looking at other people's code), and partially because magrittr also offers some other useful pipe operators. You'll see plenty of examples of how pipes can be used in Chapters 5-9, and learn about other pipe operators in Section 6.2.

~

Exercise 3.14. Rewrite the following function calls using pipes, with x <- 1:8:

1. sqrt(mean(x))
2. mean(sqrt(x))
3. sort(x^2-5)[1:2]

Exercise 3.15. Using the bookstore data:

```
age <- c(28, 48, 47, 71, 22, 80, 48, 30, 31)
purchase <- c(20, 59, 2, 12, 22, 160, 34, 34, 29)
visit_length <- c(5, 2, 20, 22, 12, 31, 9, 10, 11)
bookstore <- data.frame(age, purchase, visit_length)
```

Add a new variable rev_per_minute which is the ratio between purchase and the visit length, using a pipe.

3.10 Flavours of R: base and tidyverse

R is a programming *language*, and just like any language, it has different dialects. When you read about R online, you'll frequently see people mentioning the words "base" and "tidyverse". These are the two most common dialects of R. Base R is just that, R in its purest form. The tidyverse is a collection of add-on packages for working with different types of data. The two are fully compatible, and you can mix and match as much as you like. Both ggplot2 and magrittr are part of the tidyverse.

In recent years, the tidyverse has been heavily promoted as being "modern" R which "makes data science faster, easier and more fun". You should believe the hype. The tidyverse is marvellous. But if you only learn tidyverse R, you will miss out on

much of what R has to offer. Base R is just as marvellous, and can definitely make data science as fast, easy and fun as the tidyverse. Besides, nobody uses just base R anyway - there are a ton of non-tidyverse packages that extend and enrich R in exciting new ways. Perhaps "extended R" or "superpowered R" would be better names for the non-tidyverse dialect.

Anyone who tells you to just learn one of these dialects is wrong. Both are great, they work extremely well together, and they are similar enough that you shouldn't limit yourself to just mastering one of them. This book will show you both base R and tidyverse solutions to problems, so that you can decide for yourself which is faster, easier, and more fun.

A defining property of the tidyverse is that there are separate functions for everything, which is perfect for code that relies on pipes. In contrast, base R uses fewer functions, but with more parameters, to perform the same tasks. If you use tidyverse solutions there is a good chance that there exists a function which performs exactly the task you're going to do with its default settings. This is great (once again, especially if you want to use pipes), but it means that there are many more functions to master for tidyverse users, whereas you can make do with much fewer in base R. You will spend more time looking up function arguments when working with base R (which fortunately is fairly straightforward using the `?` documentation), but on the other hand, looking up arguments for a function that you know the name of is easier than finding a function that does something very specific that you don't know the name of. There are advantages and disadvantages to both approaches.

3.11 Ethics and good statistical practice

Throughout this book, there will be sections devoted to ethics. Good statistical practice is intertwined with good ethical practice. Both require transparent assumptions, reproducible results, and valid interpretations.

One of the most commonly cited ethical guidelines for statistical work is The American Statistical Association's *Ethical Guidelines for Statistical Practice* (Committee on Professional Ethics of the American Statistical Association, 2018), a shortened version of which is presented below[9]. The full ethical guidelines are available at https://www.amstat.org/ASA/Your-Career/Ethical-Guidelines-for-Statistical-Practice.aspx

- **Professional Integrity and Accountability**. The ethical statistician uses methodology and data that are relevant and appropriate; without favoritism or prejudice; and in a manner intended to produce valid, interpretable, and reproducible results. The ethical statistician does not knowingly accept work for which he/she is not sufficiently qualified, is honest with the client about

[9]The excerpt is from the version of the guidelines dated April 2018, and presented here with permission from the ASA.

any limitation of expertise, and consults other statisticians when necessary or in doubt. It is essential that statisticians treat others with respect.

- **Integrity of data and methods**. The ethical statistician is candid about any known or suspected limitations, defects, or biases in the data that may affect the integrity or reliability of the statistical analysis. Objective and valid interpretation of the results requires that the underlying analysis recognizes and acknowledges the degree of reliability and integrity of the data.
- **Responsibilities to Science/Public/Funder/Client**. The ethical statistician supports valid inferences, transparency, and good science in general, keeping the interests of the public, funder, client, or customer in mind (as well as professional colleagues, patients, the public, and the scientific community).
- **Responsibilities to Research Subjects**. The ethical statistician protects and respects the rights and interests of human and animal subjects at all stages of their involvement in a project. This includes respondents to the census or to surveys, those whose data are contained in administrative records, and subjects of physically or psychologically invasive research.
- **Responsibilities to Research Team Colleagues**. Science and statistical practice are often conducted in teams made up of professionals with different professional standards. The statistician must know how to work ethically in this environment.
- **Responsibilities to Other Statisticians or Statistics Practitioners**. The practice of statistics requires consideration of the entire range of possible explanations for observed phenomena, and distinct observers drawing on their own unique sets of experiences can arrive at different and potentially diverging judgments about the plausibility of different explanations. Even in adversarial settings, discourse tends to be most successful when statisticians treat one another with mutual respect and focus on scientific principles, methodology, and the substance of data interpretations.
- **Responsibilities Regarding Allegations of Misconduct**. The ethical statistician understands the differences between questionable statistical, scientific, or professional practices and practices that constitute misconduct. The ethical statistician avoids all of the above and knows how each should be handled.
- **Responsibilities of Employers, Including Organizations, Individuals, Attorneys, or Other Clients Employing Statistical Practitioners**. Those employing any person to analyze data are implicitly relying on the profession's reputation for objectivity. However, this creates an obligation on the part of the employer to understand and respect statisticians' obligation of objectivity.

Similar ethical guidelines for statisticians have been put forward by the International Statistical Institute (https://www.isi-web.org/about-isi/policies/professional-ethics), the United Nations Statistics Division (https://unstats.un.org/unsd/dnss/gp/fundprinciples.aspx), and the Data Science Association (http://www.datascie

nceassn.org/code-of-conduct.html). For further reading on ethics in statistics, see Franks (2020) and Fleming & Bruce (2021).

~

Exercise 3.16. *Discuss the following.* In the introduction to American Statistical Association's *Ethical Guidelines for Statistical Practice*, it is stated that "using statistics in pursuit of unethical ends is inherently unethical". What is considered unethical depends on social, moral, political, and religious values, and ultimately you must decide for yourself what you consider to be unethical ends. Which (if any) of the following do you consider to be unethical?

1. Using statistical analysis to help a company that harm the environment through their production processes. Does it matter to you what the purpose of the analysis is?
2. Using statistical analysis to help a tobacco or liquor manufacturing company. Does it matter to you what the purpose of the analysis is?
3. Using statistical analysis to help a bank identify which loan applicants that are likely to default on their loans.
4. Using statistical analysis of social media profiles to identify terrorists.
5. Using statistical analysis of social media profiles to identify people likely to protest against the government.
6. Using statistical analysis of social media profiles to identify people to target with political adverts.
7. Using statistical analysis of social media profiles to target ads at people likely to buy a bicycle.
8. Using statistical analysis of social media profiles to target ads at people likely to gamble at a new online casino. Does it matter to you if it's an ad for the casino or for help for gambling addiction?

Chapter 4

Exploratory data analysis and unsupervised learning

Exploratory data analysis (EDA) is a process in which we summarise and visually explore a dataset. An important part of EDA is unsupervised learning, which is a collection of methods for finding interesting subgroups and patterns in our data. Unlike statistical hypothesis testing, which is used to reject hypotheses, EDA can be used to *generate* hypotheses (which can then be confirmed or rejected by new studies). Another purpose of EDA is to find outliers and incorrect observations, which can lead to a cleaner and more useful dataset. In EDA we ask questions about our data and then try to answer them using summary statistics and graphics. Some questions will prove to be important, and some will not. The key to finding important questions is to ask a lot of questions. This chapter will provide you with a wide range of tools for question-asking.

After working with the material in this chapter, you will be able to use R to:

- Create reports using R Markdown,
- Customise the look of your plots,
- Visualise the distribution of a variable,
- Create interactive plots,
- Detect and label outliers,
- Investigate patterns in missing data,
- Visualise trends,
- Plot time series data,
- Visualise multiple variables at once using scatterplot matrices, correlograms and bubble plots,
- Visualise multivariate data using principal component analysis,
- Use unsupervised learning techniques for clustering,

- Use factor analysis to find latent variables in your data.

4.1 Reports with R Markdown

R Markdown files can be used to create nicely formatted documents using R, that are easy to export to other formats, like HTML, Word or pdf. They allow you to mix R code with results and text. They can be used to create reproducible reports that are easy to update with new data, because they include the code for making tables and figures. Additionally, they can be used as notebooks for keeping track of your work and your thoughts as you carry out an analysis. You can even use them for writing books - in fact, this entire book was written using R Markdown.

It is often a good idea to use R Markdown for exploratory analyses, as it allows you to write down your thoughts and comments as the analysis progresses, as well as to save the results of the exploratory journey. For that reason, we'll start this chapter by looking at some examples of what you can do using R Markdown. According to your preference, you can use either R Markdown or ordinary R scripts for the analyses in the remainder of the chapter. The R code used is the same and the results are identical, but if you use R Markdown, you can also save the output of the analysis in a nicely formatted document.

4.1.1 A first example

When you create a new R Markdown document in RStudio by clicking *File > New File > R Markdown* in the menu, a document similar to that below is created :

```
---
title: "Untitled"
author: "Måns Thulin"
date: "10/20/2020"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax
for authoring HTML, PDF, and MS Word documents. For more details on
using R Markdown see
<http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that
```

```
includes both content as well as the output of any embedded R code
chunks within the document. You can embed an R code chunk like this:

```{r cars}
summary(cars)
```

Including Plots

You can also embed plots, for example:

```{r pressure, echo=FALSE}
plot(pressure)
```

Note that the `echo = FALSE` parameter was added to the code chunk to
prevent printing of the R code that generated the plot.
```

Press the *Knit* button at the top of the Script panel to create an HTML document using this Markdown file. It will be saved in the same folder as your Markdown file. Once the HTML document has been created, it will open so that you can see the results. You may have to install additional packages for this to work, in which case RStudio will prompt you.

Now, let's have a look at what the different parts of the Markdown document do. The first part is called the *document header* or *YAML header*. It contains information about the document, including its title, the name of its author, and the date on which it was first created:

```
---
title: "Untitled"
author: "Måns Thulin"
date: "10/20/2020"
output: html_document
---
```

The part that says `output: html_document` specifies what type of document should be created when you press *Knit*. In this case, it's set to `html_document`, meaning that an HTML document will be created. By changing this to `output: word_document` you can create a `.docx` Word document instead. By changing it to `output: pdf_document`, you can create a `.pdf` document using LaTeX (you'll have to install LaTeX if you haven't already - RStudio will notify you if that is the case).

The second part sets the default behaviour of *code chunks* included in the document, specifying that the output from running the chunks should be printed unless otherwise specified:

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
```

The third part contains the first proper section of the document. First, a header is created using `##`. Then there is some text with formatting: `< >` is used to create a link and `**` is used to get **bold text**. Finally, there is a code chunk, delimited by ```:

```
## R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax
for authoring HTML, PDF, and MS Word documents. For more details on
using R Markdown see
<http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that
includes both content as well as the output of any embedded R code
chunks within the document. You can embed an R code chunk like this:

```{r cars}
summary(cars)
```
```

The fourth and final part contains another section, this time with a figure created using R. A setting is added to the code chunk used to create the figure, which prevents the underlying code from being printed in the document:

```
## Including Plots

You can also embed plots, for example:

```{r pressure, echo=FALSE}
plot(pressure)
```

Note that the `echo = FALSE` parameter was added to the code chunk to
prevent printing of the R code that generated the plot.
```

In the next few sections, we will look at how formatting and code chunks work in R Markdown.

4.1.2 Formatting text

To create plain text in a Markdown file, you simply have to write plain text. If you wish to add some formatting to your text, you can use the following:

- `_italics_` or `*italics*`: to create text in *italics*.
- `__bold__` or `**bold**`: to create **bold** text.
- `[linked text](http://www.modernstatisticswithr.com)`: to create linked text.
- `` `code` ``: to include inline `code` in your document.
- `$a^2 + b^2 = c^2$`: to create inline equations like $a^2 + b^2 = c^2$ using LaTeX syntax.
- `$$a^2 + b^2 = c^2$$`: to create a centred equation on a new line, like

$$a^2 + b^2 = c^2.$$

To add headers and subheaders, and to divide your document into section, start a new line with `#`'s as follows:

```
# Header text
## Sub-header text
### Sub-sub-header text
...and so on.
```

4.1.3 Lists, tables, and images

To create a bullet list, you can use `*` as follows. Note that you need a blank line between your list and the previous paragraph to begin a list.

```
* Item 1
* Item 2
   + Sub-item 1
   + Sub-item 2
* Item 3
```

yielding:

- Item 1
- Item 2
  - Sub-item 1
  - Sub-item 2
- Item 3

To create an ordered list, use:

```
1. First item
2. Second item
  i) Sub-item 1
  ii) Sub-item 2
3. Item 3
```

yielding:

1. First item

2. Second item

i) Sub-item 1
ii) Sub-item 2

3. Item 3

To create a table, use `|` and `---------` as follows:

```
Column 1  | Column 2
--------- | ---------
Content   | More content
Even more  | And some here
Even more?  | Yes!
```

which yields the following output:

Column 1	Column 2
Content	More content
Even more	And some here
Even more?	Yes!

To include an image, use the same syntax as when creating linked text with a link to the image path (either local or on the web), but with a `!` in front:

```
![](https://www.r-project.org/Rlogo.png)

![Put some text here if you want a caption](https://www.r-project.org/Rlogo.png)
```

which yields the following:

Figure 4.1: Put some text here if you want a caption

4.1.4 Code chunks

The simplest way to define a code chunk is to write:

```
```{r}
plot(pressure)
```
```

In RStudio, Ctrl+Alt+I is a keyboard shortcut for inserting this kind of code chunk.

We can add a name and a caption to the chunk, which lets us reference objects created by the chunk:

```
```{r pressureplot, fig.cap = "Plot of the pressure data."}
plot(pressure)
```
```

```
As we can see in Figure \\ref{fig:cars-plot}, the relationship between
temperature and pressure resembles a banana.
```

This yields the following output:

```
plot(pressure)
```

As we can see in Figure 4.2, the relationship between temperature and pressure resembles a banana.

In addition, you can add settings to the chunk header to control its behaviour. For

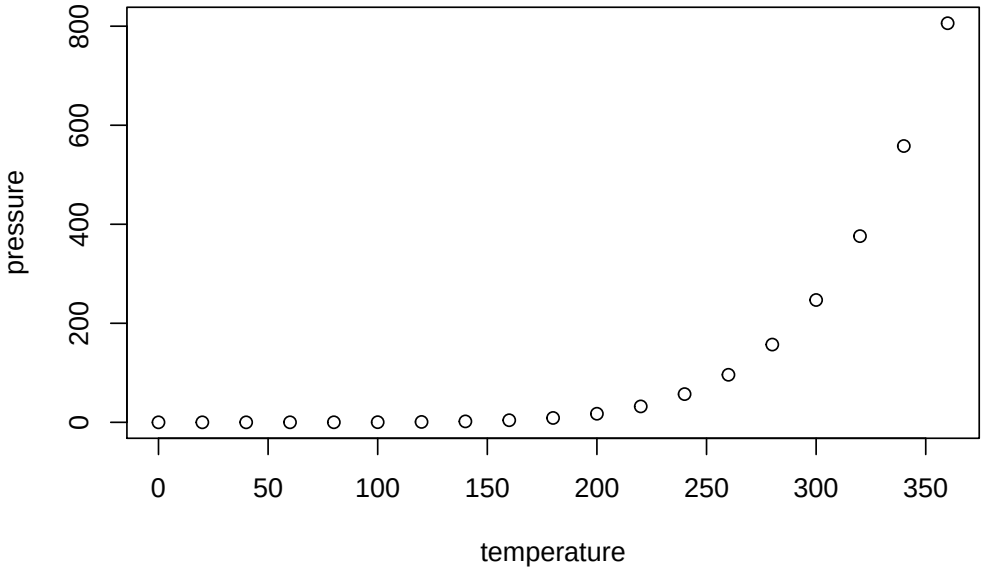

Figure 4.2: Plot of the pressure data.

instance, you can include a code chunk without running it by adding `echo = FALSE`:

```
```{r, eval = FALSE}
plot(pressure)
```
```

You can add the following settings to your chunks:

- `echo = FALSE` to run the code without printing it,
- `eval = FALSE` to print the code without running it,
- `results = "hide"` to hide printed output,
- `fig.show = "hide"` to hide plots,
- `warning = FALSE` to suppress warning messages from being printed in your document,
- `message = FALSE` to suppress other messages from being printed in your document,
- `include = FALSE` to run a chunk without showing the code or results in the document,
- `error = TRUE` to continue running your R Markdown document even if there is an error in the chunk (by default, the documentation creation stops if there is an error).

Data frames can be printed either as in the Console or as a nicely formatted table. For example,

Table 4.2: Some data I found.

Ozone	Solar.R	Wind	Temp	Month	Day
41	190	7.4	67	5	1
36	118	8.0	72	5	2
12	149	12.6	74	5	3
18	313	11.5	62	5	4
NA	NA	14.3	56	5	5
28	NA	14.9	66	5	6

```{r, echo = FALSE}
head(airquality)
```

yields:

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

whereas

```{r, echo = FALSE}
knitr::kable(
  head(airquality),
  caption = "Some data I found."
)
```

yields the table below.

Further help and documentation for R Markdown can be found through the RStudio menus, by clicking *Help > Cheatsheets > R Markdown Cheat Sheet* or *Help > Cheatsheets > R Markdown Reference Guide*.

4.2 Customising ggplot2 plots

We'll be using `ggplot2` a lot in this chapter, so before we get started with exploratory analyses, we'll take some time to learn how we can customise the look of `ggplot2`-plots.

Consider the following facetted plot from Section 2.7.4:

```
library(ggplot2)

ggplot(msleep, aes(brainwt, sleep_total)) +
      geom_point() +
      xlab("Brain weight (logarithmic scale)") +
      ylab("Total sleep time") +
      scale_x_log10() +
      facet_wrap(~ vore)
```

It looks nice, sure, but there may be things that you'd like to change. Maybe you'd like the plot's background to be white instead of grey, or perhaps you'd like to use a different font. These, and many other things, can be modified using *themes*.

4.2.1 Using themes

`ggplot2` comes with a number of basic themes. All are fairly similar, but differ in things like background colour, grids and borders. You can add them to your plot using `theme_themeName`, where `themeName` is the name of the theme[1]. Here are some examples:

```
p <- ggplot(msleep, aes(brainwt, sleep_total, colour = vore)) +
      geom_point() +
      xlab("Brain weight (logarithmic scale)") +
      ylab("Total sleep time") +
      scale_x_log10() +
      facet_wrap(~ vore)

# Create plot with different themes:
p + theme_grey() # The default theme
p + theme_bw()
p + theme_linedraw()
p + theme_light()
p + theme_dark()
p + theme_minimal()
p + theme_classic()
```

There are several packages available that contain additional themes. Let's download a few:

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

# Create plot with different themes from ggthemes:
p + theme_tufte() # Minimalist Tufte theme
```

[1]See `?theme_grey` for a list of available themes.

```
p + theme_wsj() # Wall Street Journal
p + theme_solarized() + scale_colour_solarized() # Solarized colours

##############################

install.packages("hrbrthemes")
library(hrbrthemes)

# Create plot with different themes from hrbrthemes:
p + theme_ipsum() # Ipsum theme
p + theme_ft_rc() # Suitable for use with dark RStudio themes
p + theme_modern_rc() # Suitable for use with dark RStudio themes
```

4.2.2 Colour palettes

Unlike e.g. background colours, the *colour palette*, i.e. the list of colours used for plotting, is not part of the theme that you're using. Next, we'll have a look at how to change the colour palette used for your plot.

Let's start by creating a `ggplot` object:

```
p <- ggplot(msleep, aes(brainwt, sleep_total, colour = vore)) +
      geom_point() +
      xlab("Brain weight (logarithmic scale)") +
      ylab("Total sleep time") +
      scale_x_log10()
```

You can change the colour palette using `scale_colour_brewer`. Three types of colour palettes are available:

- **Sequential palettes**: that range from a colour to white. These are useful for representing ordinal (i.e. ordered) categorical variables and numerical variables.
- **Diverging palettes**: these range from one colour to another, with white in between. Diverging palettes are useful when there is a meaningful middle or 0 value (e.g. when your variables represent temperatures or profit/loss), which can be mapped to white.
- **Qualitative palettes**: which contain multiple distinct colours. These are useful for nominal (i.e. with no natural ordering) categorical variables.

See `?scale_colour_brewer` or http://www.colorbrewer2.org for a list of the available palettes. Here are some examples:

```
# Sequential palette:
p + scale_colour_brewer(palette = "OrRd")
# Diverging palette:
p + scale_colour_brewer(palette = "RdBu")
```

```
# Qualitative palette:
p + scale_colour_brewer(palette = "Set1")
```

In this case, because `vore` is a nominal categorical variable, a qualitative palette is arguably the best choice.

4.2.3 Theme settings

The point of using a theme is that you get a combination of colours, fonts and other choices that are supposed to go well together, meaning that you don't have to spend too much time picking combinations. But if you like, you can override the default options and customise any and all parts of a theme.

The theme controls all visual aspects of the plot not related to the aesthetics. You can change the theme settings using the `theme` function. For instance, you can use `theme` to remove the legend or change its position:

```
# No legend:
p + theme(legend.position = "none")

# Legend below figure:
p + theme(legend.position = "bottom")

# Legend inside plot:
p + theme(legend.position = c(0.9, 0.7))
```

In the last example, the vector `c(0.9, 0.7)` gives the relative coordinates of the legend, with `c(0 0)` representing the bottom left corner of the plot and `c(1, 1)` the upper right corner. Try to change the coordinates to different values between `0` and `1` and see what happens.

`theme` has a lot of other settings, including for the colours of the background, the grid and the text in the plot. Here are a few examples that you can use as starting point for experimenting with the settings:

```
p + theme(panel.grid.major = element_line(colour = "black"),
          panel.grid.minor = element_line(colour = "purple",
                                          linetype = "dotted"),
          panel.background = element_rect(colour = "red", size = 2),
          plot.background = element_rect(fill = "yellow"),
          axis.text = element_text(family = "mono", colour = "blue"),
          axis.title = element_text(family = "serif", size = 14))
```

To find a complete list of settings, see `?theme`, `?element_line` (lines), `?element_rect` (borders and backgrounds), `?element_text` (text), and `element_blank` (for suppressing plotting of elements). As before, you can use `colors()` to get a list of

built-in colours, or use colour hex codes.

~

Exercise 4.1. Use the documentation for `theme` and the `element_...` functions to change the plot object `p` created above as follows:

1. Change the background colour of the entire plot to `lightblue`.
2. Change the font of the legend to `serif`.
3. Remove the grid.
4. Change the colour of the axis ticks to `orange` and make them thicker.

4.3 Exploring distributions

It is often useful to visualise the distribution of a numerical variable. Comparing the distributions of different groups can lead to important insights. Visualising distributions is also essential when checking assumptions used for various statistical tests (sometimes called *initial data analysis*). In this section we will illustrate how this can be done using the `diamonds` data from the `ggplot2` package, which you started to explore in Chapter 2.

4.3.1 Density plots and frequency polygons

We already know how to visualise the distribution of the data by dividing it into bins and plotting a histogram:

```
library(ggplot2)
ggplot(diamonds, aes(carat)) +
      geom_histogram(colour = "black")
```

A similar plot is created using frequency polygons, which uses lines instead of bars to display the counts in the bins:

```
ggplot(diamonds, aes(carat)) +
      geom_freqpoly()
```

An advantage with frequency polygons is that they can be used to compare groups, e.g. diamonds with different cuts, without facetting:

```
ggplot(diamonds, aes(carat, colour = cut)) +
      geom_freqpoly()
```

It is clear from this figure that there are more diamonds with ideal cuts than diamonds with fair cuts in the data. The polygons have roughly the same shape, except perhaps for the polygon for diamonds with fair cuts.

In some cases, we are more interested in the shape of the distribution than in the actual counts in the different bins. Density plots are similar to frequency polygons but show an estimate of the density function of the underlying random variable. These estimates are smooth curves that are scaled so that the area below them is 1 (i.e. scaled to be proper density functions):

```
ggplot(diamonds, aes(carat, colour = cut)) +
      geom_density()
```

From this figure, it becomes clear that low-carat diamonds tend to have better cuts, which wasn't obvious from the frequency polygons. However, the plot does not provide any information about *how* common different cuts are. Use density plots if you're more interested in the shape of a variable's distribution, and frequency polygons if you're more interested in counts.

~

Exercise 4.2. Using the density plot created above and the documentation for `geom_density`, do the following:

1. Increase the smoothness of the density curves.
2. Fill the area under the density curves with the same colour as the curves themselves.
3. Make the colours that fill the areas under the curves transparent.
4. The figure still isn't that easy to interpret. Install and load the `ggridges` package, an extension of `ggplot2` that allows you to make so-called ridge plots (density plots that are separated along the y-axis, similar to facetting). Read the documentation for `geom_density_ridges` and use it to make a ridge plot of diamond prices for different cuts.

Exercise 4.3. Return to the histogram created by `ggplot(diamonds, aes(carat)) + geom_histogram()` above. As there are very few diamonds with carat greater than 3, cut the x-axis at 3. Then decrease the bin width to 0.01. Do any interesting patterns emerge?

4.3.2 Asking questions

Exercise 4.3 causes us to ask why diamonds with carat values that are multiples of 0.25 are more common than others. Perhaps the price is involved? Unfortunately, a plot of `carat` versus `price` is not that informative:

```
ggplot(diamonds, aes(carat, price)) +
      geom_point(size = 0.05)
```

Maybe we could compute the average price in each bin of the histogram? In that case, we need to extract the bin breaks from the histogram somehow. We could then create a new categorical variable using the breaks with `cut` (as we did in Exercise 3.7). It turns out that extracting the bins is much easier using base graphics than `ggplot2`, so let's do that:

```
# Extract information from a histogram with bin width 0.01,
# which corresponds to 481 breaks:
carat_br <- hist(diamonds$carat, breaks = 481, right = FALSE,
                 plot = FALSE)
# Of interest to us are:
# carat_br$breaks, which contains the breaks for the bins
# carat_br$mid, which contains the midpoints of the bins
#               (useful for plotting!)

# Create categories for each bin:
diamonds$carat_cat <- cut(diamonds$carat, 481, right = FALSE)
```

We now have a variable, `carat_cat`, that shows to which bin each observation belongs. Next, we'd like to compute the mean for each bin. This is a grouped summary - mean by category. After we've computed the bin means, we could then plot them against the bin midpoints. Let's try it:

```
means <- aggregate(price ~ carat_cat, data = diamonds, FUN = mean)

plot(carat_br$mid, means$price)
```

That didn't work as intended. We get an error message when attempting to plot the results:

```
Error in xy.coords(x, y, xlabel, ylabel, log) :
  'x' and 'y' lengths differ
```

The error message implies that the number of bins and the number of mean values that have been computed differ. But we've just computed the mean for each bin, haven't we? So what's going on?

By default, `aggregate` ignores groups for which there are no values when computing grouped summaries. In this case, there are a lot of empty bins - there is for instance no observation in the `[4.99,5)` bin. In fact, only 272 out of the 481 bins are non-empty.

We can solve this in different ways. One way is to remove the empty bins. We can do this using the `match` function, which returns the positions of *matching values* in

two vectors. If we use it with the bins from the grouped summary and the vector containing all bins, we can find the indices of the non-empty bins. This requires the use of the `levels` function, that you'll learn more about in Section 5.4:

```
means <- aggregate(price ~ carat_cat, data = diamonds, FUN = mean)

id <- match(means$carat_cat, levels(diamonds$carat_cat))
```

Finally, we'll also add some vertical lines to our plot, to call attention to multiples of 0.25.

Using base graphics is faster here:

```
plot(carat_br$mid[id], means$price,
     cex = 0.5)

# Add vertical lines at multiples
# of 0.25:
abline(v = c(0.5, 0.75, 1,
             1.25, 1.5))
```

But we can of course stick to `ggplot2` if we like:

```
library(ggplot2)

d2 <- data.frame(
         bin = carat_br$mid[id],
         mean = means$price)

ggplot(d2, aes(bin, mean)) +
       geom_point() +
       geom_vline(xintercept =
                  c(0.5, 0.75, 1,
                    1.25, 1.5))
# geom_vline add vertical lines at
# multiples of 0.25
```

It appears that there are small jumps in the prices at some of the 0.25-marks. This explains why there are more diamonds just above these marks than just below.

The above example illustrates three crucial things regarding exploratory data analysis:

- Plots (in our case, the histogram) often lead to new questions.
- Often, we must transform, summarise or otherwise manipulate our data to answer a question. Sometimes this is straightforward and sometimes it means diving deep into R code.
- Sometimes the thing that we're trying to do doesn't work straight away. There is almost always a solution though (and oftentimes more than one!). The more you work with R, the more problem-solving tricks you will learn.

4.3.3 Violin plots

Density curves can also be used as alternatives to boxplots. In Exercise 2.16, you created boxplots to visualise price differences between diamonds of different cuts:

```
ggplot(diamonds, aes(cut, price)) +
      geom_boxplot()
```

Instead of using a boxplot, we can use a violin plot. Each group is represented by a "violin", given by a rotated and duplicated density plot:

```
ggplot(diamonds, aes(cut, price)) +
      geom_violin()
```

Compared to boxplots, violin plots capture the entire distribution of the data rather than just a few numerical summaries. If you like numerical summaries (and you should!) you can add the median and the quartiles (corresponding to the borders of the box in the boxplot) using the `draw_quantiles` argument:

```
ggplot(diamonds, aes(cut, price)) +
      geom_violin(draw_quantiles =  c(0.25, 0.5, 0.75))
```

~

Exercise 4.4. Using the first boxplot created above, i.e. `ggplot(diamonds, aes(cut, price)) + geom_violin()`, do the following:

1. Add some colour to the plot by giving different colours to each violin.
2. Because the categories are shown along the x-axis, we don't really need the legend. Remove it.
3. Both boxplots and violin plots are useful. Maybe we can have the best of both worlds? Add the corresponding boxplot inside each violin. Hint: the `width` and `alpha` arguments in `geom_boxplot` are useful for creating a nice-looking figure here.
4. Flip the coordinate system to create horizontal violins and boxes instead.

4.3.4 Combine multiple plots into a single graphic

When exploring data with many variables, you'll often want to make the same kind of plot (e.g. a violin plot) for several variables. It will frequently make sense to place these side-by-side in the same plot window. The `patchwork` package extends `ggplot2` by letting you do just that. Let's install it:

```
install.packages("patchwork")
```

To use it, save each plot as a plot object and then add them together:

```
plot1 <- ggplot(diamonds, aes(cut, carat, fill = cut)) +
            geom_violin() +
            theme(legend.position = "none")
plot2 <- ggplot(diamonds, aes(cut, price, fill = cut)) +
            geom_violin() +
            theme(legend.position = "none")

library(patchwork)
plot1 + plot2
```

You can also arrange the plots on multiple lines, with different numbers of plots on each line. This is particularly useful if you are combining different types of plots in a single plot window. In this case, you separate plots that are same line by `|` and mark the beginning of a new line with `/`:

```
# Create two more plot objects:
plot3 <- ggplot(diamonds, aes(cut, depth, fill = cut)) +
            geom_violin() +
            theme(legend.position = "none")
plot4 <- ggplot(diamonds, aes(carat, fill = cut)) +
            geom_density(alpha = 0.5) +
            theme(legend.position = c(0.9, 0.6))

# One row with three plots and one row with a single plot:
(plot1 | plot2 | plot3) / plot4

# One column with three plots and one column with a single plot:
(plot1 / plot2 / plot3) | plot4
```

(You may need to enlarge your plot window for this to look good!)

4.4 Outliers and missing data

4.4.1 Detecting outliers

Both boxplots and scatterplots are helpful in detecting deviating observations - often called outliers. Outliers can be caused by measurement errors or errors in the data input but can also be interesting rare cases that can provide valuable insights about the process that generated the data. Either way, it is often of interest to detect outliers, for instance because that may influence the choice of what statistical tests to use.

Let's return to the scatterplot of diamond carats versus prices:

```
ggplot(diamonds, aes(carat, price)) +
      geom_point()
```

There are some outliers which we may want to study further. For instance, there is a surprisingly cheap 5 carat diamond, and some cheap 3 carat diamonds[2]. But how can we identify those points?

One option is to use the `plotly` package to make an interactive version of the plot, where we can hover interesting points to see more information about them. Start by installing it:

```
install.packages("plotly")
```

To use `plotly` with a `ggplot` graphic, we store the graphic in a variable and then use it as input to the `ggplotly` function. The resulting (interactive!) plot takes a little longer than usual to load. Try hovering the points:

```
myPlot <- ggplot(diamonds, aes(carat, price)) +
      geom_point()

library(plotly)
ggplotly(myPlot)
```

By default, `plotly` only shows the carat and price of each diamond. But we can add more information to the box by adding a `text` aesthetic:

```
myPlot <- ggplot(diamonds, aes(carat, price, text = paste("Row:",
                                                 rownames(diamonds)))) +
                                                 geom_point()

ggplotly(myPlot)
```

We can now find the row numbers of the outliers visually, which is very useful when exploring data.

~

Exercise 4.5. The variables `x` and `y` in the `diamonds` data describe the length and width of the diamonds (in mm). Use an interactive scatterplot to identify outliers in these variables. Check prices, carat and other information and think about if any of the outliers can be due to data errors.

[2]Note that it is not just the prices nor just the carats of these diamonds that make them outliers, but the unusual combinations of prices and carats!

4.4.2 Labelling outliers

Interactive plots are great when exploring a dataset but are not always possible to use in other contexts, e.g. for printed reports and some presentations. In these other cases, we can instead annotate the plot with notes about outliers. One way to do this is to use a geom called `geom_text`.

For instance, we may want to add the row numbers of outliers to a plot. To do so, we use `geom_text` along with a condition that specifies for which points we should add annotations. Like in Section 3.2.3, if we e.g. wish to add row numbers for diamonds with carats greater than four, our condition would be `carat > 4`. The `ifelse` function, which we'll look closer at in Section 6.3, is perfect to use here. The syntax will be `ifelse(condition, what text to write if the condition is satisfied, what text to write else)`. To add row names for observations that fulfil the condition but not for other observations, we use `ifelse(condition, rownames(diamonds), "")`. If instead, we wanted to print the price of the diamonds, we'd use `ifelse(condition, price, "")`.

Here are some different examples of conditions used to plot text:

```
## Using the row number (the 5 carat diamond is on row 27,416)
ggplot(diamonds, aes(carat, price)) +
      geom_point() +
      geom_text(aes(label = ifelse(rownames(diamonds) == 27416,
                                   rownames(diamonds), "")),
                hjust = 1.1)
## (hjust=1.1 shifts the text to the left of the point)

## Plot text next to all diamonds with carat>4
ggplot(diamonds, aes(carat, price)) +
     geom_point() +
     geom_text(aes(label = ifelse(carat > 4, rownames(diamonds), "")),
               hjust = 1.1)

## Plot text next to 3 carat diamonds with a price below 7500
ggplot(diamonds, aes(carat, price)) +
      geom_point() +
      geom_text(aes(label = ifelse(carat == 3 & price < 7500,
                                   rownames(diamonds), "")),
                hjust = 1.1)
```

~

Exercise 4.6. Create a static (i.e. non-interactive) scatterplot of `x` versus `y` from the `diamonds` data. Label the diamonds with suspiciously high y-values.

4.4.3 Missing data

Like many datasets, the mammal sleep data `msleep` contains a lot of missing values, represented by `NA` (Not Available) in R. This becomes evident when we have a look at the data:

```
library(ggplot2)
View(msleep)
```

We can check if a particular observation is missing using the `is.na` function:

```
is.na(msleep$sleep_rem[4])
is.na(msleep$sleep_rem)
```

We can count the number of missing values for each variable using:

```
colSums(is.na(msleep))
```

Here, `colSums` computes the sum of `is.na(msleep)` for each column of `msleep` (remember that in summation, `TRUE` counts as `1` and `FALSE` as `0`), yielding the number of missing values for each variable. In total, there are 136 missing values in the dataset:

```
sum(is.na(msleep))
```

You'll notice that `ggplot2` prints a warning in the Console when you create a plot with missing data:

```
ggplot(msleep, aes(brainwt, sleep_total)) +
      geom_point() +
      scale_x_log10()
```

Sometimes data are missing simply because the information is not yet available (for instance, the brain weight of the mountain beaver could be missing because no one has ever weighed the brain of a mountain beaver). In other cases, data can be missing because something about them is different (for instance, values for a male patient in a medical trial can be missing because the patient died, or because some values only were collected for female patients). Therefore, it is of interest to see if there are any differences in non-missing variables between subjects that have missing data and subjects that don't.

In `msleep`, all animals have recorded values for `sleep_total` and `bodywt`. To check if the animals that have missing `brainwt` values differ from the others, we can plot them in a different colour in a scatterplot:

```
ggplot(msleep, aes(bodywt, sleep_total, colour = is.na(brainwt))) +
      geom_point() +
      scale_x_log10()
```

(If `is.na(brainwt)` is `TRUE` then the brain weight is missing in the dataset.) In this case, there are no apparent differences between the animals with missing data and those without.

~

Exercise 4.7. Create a version of the diamonds dataset where the `x` value is missing for all diamonds with $x > 9$. Make a scatterplot of carat versus price, with points where the `x` value is missing plotted in a different colour. How would you interpret this plot?

4.4.4 Exploring data

The `nycflights13` package contains data about flights to and from three airports in New York, USA, in 2013. As a summary exercise, we will study a subset of these, namely all flights departing from New York on 1 January that year:

```
install.packages("nycflights13")
library(nycflights13)
flights2 <- flights[flights$month == 1 & flights$day == 1,]
```

~

Exercise 4.8. Explore the `flights2` dataset, focusing on delays and the amount of time spent in the air. Are there any differences between the different carriers? Are there missing data? Are there any outliers?

4.5 Trends in scatterplots

Let's return to a familiar example - the relationship between animal brain size and sleep times:

```
ggplot(msleep, aes(brainwt, sleep_total)) +
      geom_point() +
      xlab("Brain weight (logarithmic scale)") +
      ylab("Total sleep time") +
      scale_x_log10()
```

There appears to be a decreasing trend in the plot. To aid the eye, we can add a smoothed line by adding a new geom, `geom_smooth`, to the figure:

```
ggplot(msleep, aes(brainwt, sleep_total)) +
      geom_point() +
      geom_smooth() +
```

```
      xlab("Brain weight (logarithmic scale)") +
      ylab("Total sleep time") +
      scale_x_log10()
```

This technique is useful for bivariate data as well as for time series, which we'll delve into next.

By default, `geom_smooth` adds a line fitted using either LOESS[3] or GAM[4], as well as the corresponding 95 % confidence interval describing the uncertainty in the estimate. There are several useful arguments that can be used with `geom_smooth`. You will explore some of these in the exercise below.

~

Exercise 4.9. Check the documentation for `geom_smooth`. Starting with the plot of brain weight vs. sleep time created above, do the following:

1. Decrease the degree of smoothing for the LOESS line that was fitted to the data. What is better in this case, more or less smoothing?
2. Fit a straight line to the data instead of a non-linear LOESS line.
3. Remove the confidence interval from the plot.
4. Change the colour of the fitted line to red.

4.6 Exploring time series

Before we have a look at time series, you should install four useful packages: `forecast`, `nlme`, `fma` and `fpp2`. The first contains useful functions for plotting time series data, and the latter three contain datasets that we'll use.

```
install.packages(c("nlme", "forecast", "fma", "fpp2"),
                 dependencies = TRUE)
```

The `a10` dataset contains information about the monthly anti-diabetic drug sales in Australia during the period July 1991 to June 2008. By checking its structure, we see that it is saved as a time series object[5]:

```
library(fpp2)
str(a10)
```

[3]LOESS, LOcally Estimated Scatterplot Smoothing, is a non-parametric regression method that fits a polynomial to local areas of the data.

[4]GAM, Generalised Additive Model, is a generalised linear model where the response variable is a linear function of smooth functions of the predictors.

[5]Time series objects are a special class of vectors, with data sampled at equispaced points in time. Each observation can have a year/date/time associated with it.

`ggplot2` requires that data is saved as a data frame in order for it to be plotted. In order to plot the time series, we could first convert it to a data frame and then plot each point using `geom_points`:

```
a10_df <- data.frame(time = time(a10), sales = a10)
ggplot(a10_df, aes(time, sales)) +
      geom_point()
```

It is however usually preferable to plot time series using lines instead of points. This is done using a different geom: `geom_line`:

```
ggplot(a10_df, aes(time, sales)) +
      geom_line()
```

Having to convert the time series object to a data frame is a little awkward. Luckily, there is a way around this. `ggplot2` offers a function called `autoplot`, that automatically draws an appropriate plot for certain types of data. `forecast` extends this function to time series objects:

```
library(forecast)
autoplot(a10)
```

We can still add other geoms, axis labels and other things just as before. `autoplot` has simply replaced the `ggplot(data, aes()) + geom` part that would be the first two rows of the `ggplot2` figure, and has implicitly converted the data to a data frame.

~

Exercise 4.10. Using the `autoplot(a10)` figure, do the following:

1. Add a smoothed line describing the trend in the data. Make sure that it is smooth enough *not* to capture the seasonal variation in the data.
2. Change the label of the x-axis to "Year" and the label of the y-axis to "Sales ($ million)".
3. Check the documentation for the `ggtitle` function. What does it do? Use it with the figure.
4. Change the colour of the time series line to red.

4.6.1 Annotations and reference lines

We sometimes wish to add text or symbols to plots, for instance to highlight interesting observations. Consider the following time series plot of daily morning gold prices, based on the `gold` data from the `forecast` package:

```
library(forecast)
autoplot(gold)
```

There is a sharp spike a few weeks before day 800, which is due to an incorrect value in the data series. We'd like to add a note about that to the plot. First, we wish to find out on which day the spike appears. This can be done by checking the data manually or using some code:

```
spike_date <- which.max(gold)
```

To add a circle around that point, we add a call to `annotate` to the plot:

```
autoplot(gold) +
      annotate(geom = "point", x = spike_date, y = gold[spike_date],
               size = 5, shape = 21,
               colour = "red",
               fill = "transparent")
```

`annotate` can be used to annotate the plot with both geometrical objects and text (and can therefore be used as an alternative to `geom_text`).

~

Exercise 4.11. Using the figure created above and the documentation for `annotate`, do the following:

1. Add the text "Incorrect value" next to the circle.
2. Create a second plot where the incorrect value has been removed.
3. Read the documentation for the geom `geom_hline`. Use it to add a red reference line to the plot, at $y = 400$.

4.6.2 Longitudinal data

Multiple time series with identical time points, known as longitudinal data or panel data, are common in many fields. One example of this is given by the `elecdaily` time series from the `fpp2` package, which contains information about electricity demand in Victoria, Australia during 2014. As with a single time series, we can plot these data using `autoplot`:

```
library(fpp2)
autoplot(elecdaily)
```

In this case, it is probably a good idea to facet the data, i.e. to plot each series in a different figure:

```
autoplot(elecdaily, facets = TRUE)
```

~

Exercise 4.12. Make the following changes to the `autoplot(elecdaily, facets = TRUE)`:

1. Remove the `WorkDay` variable from the plot (it describes whether or not a given date is a workday, and while it is useful for modelling purposes, we do not wish to include it in our figure).
2. Add smoothed trend lines to the time series plots.

4.6.3 Path plots

Another option for plotting multiple time series is path plots. A path plot is a scatterplot where the points are connected with lines in the order they appear in the data (which, for time series data, should correspond to time). The lines and points can be coloured to represent time.

To make a path plot of Temperature versus Demand for the `elecdaily` data, we first convert the time series object to a data frame and create a scatterplot:

```
library(fpp2)
ggplot(as.data.frame(elecdaily), aes(Temperature, Demand)) +
      geom_point()
```

Next, we connect the points by lines using the `geom_path` geom:

```
ggplot(as.data.frame(elecdaily), aes(Temperature, Demand)) +
      geom_point() +
      geom_path()
```

The resulting figure is quite messy. Using colour to indicate the passing of time helps a little. For this, we need to add the day of the year to the data frame. To get the values right, we use `nrow`, which gives us the number of rows in the data frame.

```
elecdaily2 <- as.data.frame(elecdaily)
elecdaily2$day <- 1:nrow(elecdaily2)

ggplot(elecdaily2, aes(Temperature, Demand, colour = day)) +
      geom_point() +
      geom_path()
```

It becomes clear from the plot that temperatures were the highest at the beginning of the year and lower in the winter months (July-August).

~

Exercise 4.13. Make the following changes to the plot you created above:

1. Decrease the size of the points.
2. Add text annotations showing the dates of the highest and lowest temperatures, next to the corresponding points in the figure.

4.6.4 Spaghetti plots

In cases where we've observed multiple subjects over time, we often wish to visualise their individual time series together using so-called spaghetti plots. With `ggplot2` this is done using the `geom_line` geom. To illustrate this, we use the `Oxboys` data from the `nlme` package, showing the heights of 26 boys over time.

```
library(nlme)
ggplot(Oxboys, aes(age, height, group = Subject)) +
      geom_point() +
      geom_line()
```

The first two `aes` arguments specify the x and y-axes, and the third specifies that there should be one line per subject (i.e. per boy) rather than a single line interpolating all points. The latter would be a rather useless figure that looks like this:

```
ggplot(Oxboys, aes(age, height)) +
      geom_point() +
      geom_line() +
      ggtitle("A terrible plot")
```

Returning to the original plot, if we wish to be able to identify which time series corresponds to which boy, we can add a colour aesthetic:

```
ggplot(Oxboys, aes(age, height, group = Subject, colour = Subject)) +
      geom_point() +
      geom_line()
```

Note that the boys are ordered by height, rather than subject number, in the legend.

Now, imagine that we wish to add a trend line describing the general growth trend for all boys. The growth appears approximately linear, so it seems sensible to use `geom_smooth(method = "lm")` to add the trend:

```
ggplot(Oxboys, aes(age, height, group = Subject, colour = Subject)) +
      geom_point() +
      geom_line() +
      geom_smooth(method = "lm", colour = "red", se = FALSE)
```

Unfortunately, because we have specified in the aesthetics that the data should be grouped by `Subject`, `geom_smooth` produces one trend line for each boy. The "problem" is that when we specify an aesthetic in the `ggplot` call, it is used for all geoms.

~

Exercise 4.14. Figure out how to produce a spaghetti plot of the `Oxboys` data with a single red trend line based on the data from all 26 boys.

4.6.5 Seasonal plots and decompositions

The `forecast` package includes a number of useful functions when working with time series. One of them is `ggseasonplot`, which allows us to easily create a spaghetti plot of different periods of a time series with seasonality, i.e. with patterns that repeat seasonally over time. It works similar to the `autoplot` function, in that it replaces the `ggplot(data, aes) + geom` part of the code.

```
library(forecast)
library(fpp2)
ggseasonplot(a10)
```

This function is very useful when visually inspecting seasonal patterns.

The `year.labels` and `year.labels.left` arguments removes the legend in favour of putting the years at the end and beginning of the lines:

```
ggseasonplot(a10, year.labels = TRUE, year.labels.left = TRUE)
```

As always, we can add more things to our plot if we like:

```
ggseasonplot(a10, year.labels = TRUE, year.labels.left = TRUE) +
      ylab("Sales ($ million)") +
      ggtitle("Seasonal plot of anti-diabetic drug sales")
```

When working with seasonal time series, it is common to decompose the series into a seasonal component, a trend component and a remainder. In R, this is typically done using the `stl` function, which uses repeated LOESS smoothing to decompose the series. There is an `autoplot` function for `stl` objects:

```
autoplot(stl(a10, s.window = 365))
```

This plot can too be manipulated in the same way as other `ggplot` objects. You can access the different parts of the decomposition as follows:

```
stl(a10, s.window = 365)$time.series[,"seasonal"]
stl(a10, s.window = 365)$time.series[,"trend"]
stl(a10, s.window = 365)$time.series[,"remainder"]
```

~

Exercise 4.15. Investigate the `writing` dataset from the `fma` package graphically. Make a time series plot with a smoothed trend line, a seasonal plot and an `stl`-decomposition plot. Add appropriate plot titles and labels to the axes. Can you see any interesting patterns?

4.6.6 Detecting changepoints

The `changepoint` package contains a number of methods for detecting *changepoints* in time series, i.e. time points at which either the mean or the variance of the series changes. Finding changepoints can be important for detecting changes in the process underlying the time series. The `ggfortify` package extends `ggplot2` by adding `autoplot` functions for a variety of tools, including those in `changepoint`. Let's install the packages:

```
install.packages(c("changepoint", "ggfortify"))
```

We can now look at some examples with the anti-diabetic drug sales data:

```
library(forecast)
library(fpp2)
library(changepoint)
library(ggfortify)

# Plot the time series:
autoplot(a10)

# Remove the seasonal part and plot the series again:
a10_ns <- a10 - stl(a10, s.window = 365)$time.series[,"seasonal"]
autoplot(a10_ns)

# Plot points where there are changes in the mean:
autoplot(cpt.mean(a10_ns))

# Choosing a different method for finding changepoints
# changes the result:
autoplot(cpt.mean(a10_ns, method = "BinSeg"))

# Plot points where there are changes in the variance:
autoplot(cpt.var(a10_ns))

# Plot points where there are changes in either the mean or
# the variance:
```

```
autoplot(cpt.meanvar(a10_ns))
```

As you can see, the different methods from `changepoint` all yield different results. The results for changes in the mean are a bit dubious - which isn't all that strange as we are using a method that looks for jumps in the mean on a time series where the increase actually is more or less continuous. The changepoint for the variance looks more reliable - there is a clear change towards the end of the series where the sales become more volatile. We won't go into details about the different methods here, but mention that the documentation at `?cpt.mean`, `?cpt.var`, and `?cpt.meanvar` contains descriptions of and references for the available methods.

~

Exercise 4.16. Are there any changepoints for variance in the `Demand` time series in `elecdaily`? Can you explain why the behaviour of the series changes?

4.6.7 Interactive time series plots

The `plotly` packages can be used to create interactive time series plots. As before, you create a `ggplot2` object as usual, assigning it to a variable and then call the `ggplotly` function. Here is an example with the `elecdaily` data:

```
library(plotly)
library(fpp2)
myPlot <- autoplot(elecdaily[,"Demand"])

ggplotly(myPlot)
```

When you hover the mouse pointer over a point, a box appears, displaying information about that data point. Unfortunately, the date formatting isn't great in this example - dates are shown as weeks with decimal points. We'll see how to fix this in Section 5.6.

4.7 Using polar coordinates

Most plots are made using *Cartesian coordinates systems*, in which the axes are orthogonal to each other and values are placed in an even spacing along each axis. In some cases, nonlinear axes (e.g. log-transformed) are used instead, as we have already seen.

Another option is to use a *polar* coordinate system, in which positions are specified using an angle and a (radial) distance from the origin. Here is an example of a polar scatterplot:

```
ggplot(msleep, aes(sleep_rem, sleep_total, colour = vore)) +
      geom_point() +
      xlab("REM sleep (circular axis)") +
      ylab("Total sleep time (radial axis)") +
      coord_polar()
```

4.7.1 Visualising periodic data

Polar coordinates are particularly useful when the data is periodic. Consider for instance the following dataset, describing monthly weather averages for Cape Town, South Africa:

```
Cape_Town_weather <- data.frame(
Month = 1:12,
Temp_C = c(22, 23, 21, 18, 16, 13, 13, 13, 14, 16, 18, 20),
Rain_mm = c(20, 20, 30, 50, 70, 90, 100, 70, 50, 40, 20, 20),
Sun_h = c(11, 10, 9, 7, 6, 6, 5, 6, 7, 9, 10, 11))
```

We can visualise the monthly average temperature using lines in a Cartesian coordinate system:

```
ggplot(Cape_Town_weather, aes(Month, Temp_C)) +
      geom_line()
```

What this plot doesn't show is that the 12th month and the 1st month actually are consecutive months. If we instead use polar coordinates, this becomes clearer:

```
ggplot(Cape_Town_weather, aes(Month, Temp_C)) +
      geom_line() +
      coord_polar()
```

To improve the presentation, we can change the scale of the x-axis (i.e. the circular axis) so that January and December aren't plotted at the same angle:

```
ggplot(Cape_Town_weather, aes(Month, Temp_C)) +
      geom_line() +
      coord_polar() +
      xlim(0, 12)
```

~

Exercise 4.17. In the plot that we just created, the last and first month of the year aren't connected. You can fix manually this by adding a cleverly designed faux data point to `Cape_Town_weather`. How?

4.7.2 Pie charts

Consider the stacked bar chart that we plotted in Section 2.8:

```
ggplot(msleep, aes(factor(1), fill = vore)) +
      geom_bar()
```

What would happen if we plotted this figure in a polar coordinate system instead? If we map the height of the bars (the y-axis of the Cartesian coordinate system) to both the angle and the radial distance, we end up with a pie chart:

```
ggplot(msleep, aes(factor(1), fill = vore)) +
      geom_bar() +
      coord_polar(theta = "y")
```

There are many arguments against using pie charts for visualisations. Most boil down to the fact that the same information is easier to interpret when conveyed as a bar chart. This is at least partially due to the fact that most people are more used to reading plots in Cartesian coordinates than in polar coordinates.

If we make a similar transformation of a grouped bar chart, we get a different type of pie chart, in which the height of the bars are mapped to both the angle and the radial distance[6]:

```
# Cartestian bar chart:
ggplot(msleep, aes(vore, fill = vore)) +
      geom_bar()

# Polar bar chart:
ggplot(msleep, aes(vore, fill = vore)) +
      geom_bar() +
      coord_polar()
```

4.8 Visualising multiple variables

4.8.1 Scatterplot matrices

When we have a large enough number of `numeric` variables in our data, plotting scatterplots of all pairs of variables becomes tedious. Luckily there are some R functions that speed up this process.

The `GGally` package is an extension to `ggplot2` which contains several functions for plotting multivariate data. They work similarly to the `autoplot` functions that we have used in previous sections. One of these is `ggpairs`, which creates a scatterplot matrix, a grid with scatterplots of all pairs of variables in `data`. In addition, it also

[6]Florence Nightingale, who famously pioneered the use of the pie chart, drew her pie charts this way.

plots density estimates (along the diagonal) and shows the (Pearson) correlation for each pair. Let's start by installing `GGally`:

```
install.packages("GGally")
```

To create a scatterplot matrix for the `airquality` dataset, simply write:

```
library(GGally)
ggpairs(airquality)
```

(Enlarging your Plot window can make the figure look better.)

If we want to create a scatterplot matrix but only want to include some of the variables in a dataset, we can do so by providing a vector with variable names. Here is an example for the animal sleep data `msleep`:

```
ggpairs(msleep[, c("sleep_total", "sleep_rem", "sleep_cycle", "awake",
                   "brainwt", "bodywt")])
```

Optionally, if we wish to create a scatterplot involving all `numeric` variables, we can replace the vector with variable names with some R code that extracts the columns containing `numeric` variables:

```
ggpairs(msleep[, which(sapply(msleep, class) == "numeric")])
```

(You'll learn more about the `sapply` function in Section 6.5.)

The resulting plot is identical to the previous one, because the list of names contained all `numeric` variables. The grab-all-numeric-variables approach is often convenient, because we don't have to write all the variable names. On the other hand, it's not very helpful in case we only want to include some of the `numeric` variables.

If we include a categorical variable in the list of variables (such as the feeding behaviour `vore`), the matrix will include a bar plot of the categorical variable as well as boxplots and facetted histograms to show differences between different categories in the continuous variables:

```
ggpairs(msleep[, c("vore", "sleep_total", "sleep_rem", "sleep_cycle",
                   "awake", "brainwt", "bodywt")])
```

Alternatively, we can use a categorical variable to colour points and density estimates using `aes(colour = ...)`. The syntax for this is follows the same pattern as that for a standard `ggplot` call - `ggpairs(data, aes)`. The only exception is that if the categorical variable is not included in the `data` argument, we much specify which data frame it belongs to:

```
ggpairs(msleep[, c("sleep_total", "sleep_rem", "sleep_cycle", "awake",
                   "brainwt", "bodywt")],
        aes(colour = msleep$vore, alpha = 0.5))
```

As a side note, if all variables in your data frame are `numeric`, and if you only are looking for a quick-and-dirty scatterplot matrix without density estimates and correlations, you can also use the base R `plot`:

```
plot(airquality)
```

~

Exercise 4.18. Create a scatterplot matrix for all `numeric` variables in `diamonds`. Differentiate different cuts by colour. Add a suitable title to the plot. (`diamonds` is a fairly large dataset, and it may take a minute or so for R to create the plot.)

4.8.2 3D scatterplots

The `plotly` package lets us make three-dimensional scatterplots with the `plot_ly` function, which can be a useful alternative to scatterplot matrices in some cases. Here is an example using the `airquality` data:

```
library(plotly)
plot_ly(airquality, x = ~Ozone, y = ~Wind, z = ~Temp,
        color = ~factor(Month))
```

Note that you can drag and rotate the plot, to see it from different angles.

4.8.3 Correlograms

Scatterplot matrices are not a good choice when we have too many variables, partially because the plot window needs to be very large to fit all variables and partially because it becomes difficult to get a good overview of the data. In such cases, a correlogram, where the strength of the correlation between each pair of variables is plotted instead of scatterplots, can be used instead. It is effectively a visualisation of the correlation matrix of the data, where the strengths and signs of the correlations are represented by different colours.

The `GGally` package contains the function `ggcorr`, which can be used to create a correlogram:

```
ggcorr(msleep[, c("sleep_total", "sleep_rem", "sleep_cycle", "awake",
                  "brainwt", "bodywt")])
```

~

Exercise 4.19. Using the `diamonds` dataset and the documentation for `ggcorr`, do the following:

1. Create a correlogram for all `numeric` variables in the dataset.

2. The Pearson correlation that `ggcorr` uses by default isn't always the best choice. A commonly used alternative is the Spearman correlation. Change the type of correlation used to create the plot to the Spearman correlation.

3. Change the colour scale from a categorical scale with 5 categories.

4. Change the colours on the scale to go from yellow (low correlation) to black (high correlation).

4.8.4 Adding more variables to scatterplots

We have already seen how scatterplots can be used to visualise two continuous and one categorical variable by plotting the two continuous variables against each other and using the categorical variable to set the colours of the points. There are however more ways we can incorporate information about additional variables into a scatterplot.

So far, we have set three aesthetics in our scatterplots: `x`, `y` and `colour`. Two other important aesthetics are `shape` and `size`, which, as you'd expect, allow us to control the shape and size of the points. As a first example using the `msleep` data, we use feeding behaviour (`vore`) to set the shapes used for the points:

```
ggplot(msleep, aes(brainwt, sleep_total, shape = vore)) +
      geom_point() +
      scale_x_log10()
```

The plot looks a little nicer if we increase the `size` of the points:

```
ggplot(msleep, aes(brainwt, sleep_total, shape = vore, size = 2)) +
      geom_point() +
      scale_x_log10()
```

Another option is to let `size` represent a continuous variable, in what is known as a bubble plot:

```
ggplot(msleep, aes(brainwt, sleep_total, colour = vore,
                   size = bodywt)) +
      geom_point() +
      scale_x_log10()
```

The size of each "bubble" now represents the weight of the animal. Because some animals are much heavier (i.e. have higher `bodywt` values) than most others, almost all points are quite small. There are a couple of things we can do to remedy this. First, we can transform `bodywt`, e.g. using the square root transformation `sqrt(bodywt)`, to decrease the differences between large and small animals. This can be done by adding `scale_size(trans = "sqrt")` to the plot. Second, we can also use `scale_size` to control the range of point sizes (e.g. from size 1 to size 20). This will cause some points to overlap, so we add `alpha = 0.5` to the geom, to make the points transparent:

```
ggplot(msleep, aes(brainwt, sleep_total, colour = vore,
                   size = bodywt)) +
      geom_point(alpha = 0.5) +
      scale_x_log10() +
      scale_size(range = c(1, 20), trans = "sqrt")
```

This produces a fairly nice-looking plot, but it'd look even better if we changed the axes labels and legend texts. We can change the legend text for the size scale by adding the argument `name` to `scale_size`. Including a `\n` in the text lets us create a line break - you'll learn more tricks like that in Section 5.5. Similarly, we can use `scale_colour_discrete` to change the legend text for the colours:

```
ggplot(msleep, aes(brainwt, sleep_total, colour = vore,
                   size = bodywt)) +
      geom_point(alpha = 0.5) +
      xlab("log(Brain weight)") +
      ylab("Sleep total (h)") +
      scale_x_log10() +
      scale_size(range = c(1, 20), trans = "sqrt",
                 name = "Square root of\nbody weight") +
      scale_colour_discrete(name = "Feeding behaviour")
```

~

Exercise 4.20. Using the bubble plot created above, do the following:

1. Replace `colour = vore` in the `aes` by `fill = vore` and add `colour = "black", shape = 21` to `geom_point`. What happens?
2. Use `ggplotly` to create an interactive version of the bubble plot above, where variable information and the animal name are displayed when you hover a point.

4.8.5 Overplotting

Let's make a scatterplot of `table` versus `depth` based on the `diamonds` dataset:

```
ggplot(diamonds, aes(table, depth)) +
      geom_point()
```

This plot is cluttered. There are too many points, which makes it difficult to see if, for instance, high `table` values are more common than low `table` values. In this section, we'll look at some ways to deal with this problem, known as overplotting.

The first thing we can try is to decrease the point size:

```
ggplot(diamonds, aes(table, depth)) +
      geom_point(size = 0.1)
```

This helps a little, but now the outliers become a bit difficult to spot. We can try changing the opacity using `alpha` instead:

```
ggplot(diamonds, aes(table, depth)) +
      geom_point(alpha = 0.2)
```

This is also better than the original plot, but neither plot is great. Instead of plotting each individual point, maybe we can try plotting the counts or densities in different regions of the plot instead? Effectively, this would be a 2D version of a histogram. There are several ways of doing this in `ggplot2`.

First, we bin the points and count the numbers in each bin, using `geom_bin2d`:

```
ggplot(diamonds, aes(table, depth)) +
      geom_bin2d()
```

By default, `geom_bin2d` uses 30 bins. Increasing that number can sometimes give us a better idea about the distribution of the data:

```
ggplot(diamonds, aes(table, depth)) +
      geom_bin2d(bins = 50)
```

If you prefer, you can get a similar plot with hexagonal bins by using `geom_hex` instead:

```
ggplot(diamonds, aes(table, depth)) +
      geom_hex(bins = 50)
```

As an alternative to bin counts, we could create a 2-dimensional density estimate and create a contour plot showing the levels of the density:

```
ggplot(diamonds, aes(table, depth)) +
      stat_density_2d(aes(fill = ..level..), geom = "polygon",
                      colour = "white")
```

The `fill = ..level..` bit above probably looks a little strange to you. It means that an internal function (the level of the contours) is used to choose the fill colours. It also means that we've reached a point where we're reaching deep into the depths of `ggplot2`!

We can use a similar approach to show a summary statistic for a third variable in a plot. For instance, we may want to plot the average price as a function of `table` and `depth`. This is called a tile plot:

```
ggplot(diamonds, aes(table, depth, z = price)) +
     geom_tile(binwidth = 1, stat = "summary_2d", fun = mean) +
```

```
    ggtitle("Mean prices for diamonds with different depths and
            tables")
```

~

Exercise 4.21. The following tasks involve the `diamonds` dataset:

1. Create a tile plot of `table` versus `depth`, showing the highest price for a diamond in each bin.
2. Create a bin plot of `carat` versus `price`. What type of diamonds have the highest bin counts?

4.8.6 Categorical data

When visualising a pair of categorical variables, plots similar to those in the previous section prove to be useful. One way of doing this is to use the `geom_count` geom. We illustrate this with an example using `diamonds`, showing how common different combinations of colours and cuts are:

```
ggplot(diamonds, aes(color, cut)) +
      geom_count()
```

However, it is often better to use colour rather than point size to visualise counts, which we can do using a tile plot. First, we have to compute the counts though, using `aggregate`. We now wish to have two grouping variables, `color` and `cut`, which we can put on the right-hand side of the formula as follows:

```
diamonds2 <- aggregate(carat ~ cut + color, data = diamonds,
                       FUN = length)
diamonds2
```

`diamonds2` is now a data frame containing the different combinations of `color` and `cut` along with counts of how many diamonds belong to each combination (labelled `carat`, because we put `carat` in our formula). Let's change the name of the last column from `carat` to `Count`:

```
names(diamonds2)[3] <- "Count"
```

Next, we can plot the counts using `geom_tile`:

```
ggplot(diamonds2, aes(color, cut, fill = Count)) +
      geom_tile()
```

It is also possible to combine point size and colours:

```
ggplot(diamonds2, aes(color, cut, colour = Count, size = Count)) +
      geom_count()
```

~

Exercise 4.22. Using the `diamonds` dataset, do the following:

1. Use a plot to find out what the most common combination of cut and clarity is.
2. Use a plot to find out which combination of cut and clarity has the highest average price.

4.8.7 Putting it all together

In the next two exercises, you will repeat what you have learned so far by investigating the `gapminder` and `planes` datasets. First, load the corresponding libraries and have a look at the documentation for each dataset:

```
install.packages("gapminder")
library(gapminder)
?gapminder

library(nycflights13)
?planes
```

~

Exercise 4.23. Do the following using the `gapminder` dataset:

1. Create a scatterplot matrix showing life expectancy, population and GDP per capita for all countries, using the data from the year 2007. Use colours to differentiate countries from different continents. Note: you'll probably need to add the argument `upper = list(continuous = "na")` when creating the scatterplot matrix. By default, correlations are shown above the diagonal, but the fact that there only are two countries from Oceania will cause a problem there - at least 3 points are needed for a correlation test.
2. Create an interactive bubble plot, showing information about each country when you hover the points. Use data from the year 2007. Put log(GDP per capita) on the x-axis and life expectancy on the y-axis. Let population determine point size. Plot each country in a different colour and facet by continent. Tip: the `gapminder` package provides a pretty colour scheme for different countries, called `country_colors`. You can use that scheme by adding `scale_colour_manual(values = country_colors)` to your plot.

Exercise 4.24. Use graphics to answer the following questions regarding the `planes` dataset:

1. What is the most common combination of manufacturer and plane type in the dataset?
2. Which combination of manufacturer and plane type has the highest average number of seats?
3. Do the numbers of seats on planes change over time? Which plane had the highest number of seats?
4. Does the type of engine used change over time?

4.9 Principal component analysis

If there are many variables in your data, it can often be difficult to detect differences between groups or create a perspicuous visualisation. A useful tool in this context is *principal component analysis* (PCA, for short), which can reduce high-dimensional data to a lower number of variables that can be visualised in one or two scatterplots. The idea is to compute new variables, called *principal components*, that are linear combinations of the original variables[7]. These are constructed with two goals in mind: the principal components should capture as much of the variance in the data as possible, and each principal component should be uncorrelated to the other components. You can then plot the principal components to get a low-dimensional representation of your data, which hopefully captures most of its variation.

By design, the number of principal components computed are as many as the original number of variables, with the first having the largest variance, the second having the second largest variance, and so on. We hope that it will suffice to use just the first few of these to represent most of the variation in the data, but this is not guaranteed. Principal component analysis is more likely to yield a useful result if several variables are correlated.

To illustrate the principles of PCA we will use a dataset from Charytanowicz et al. (2010), containing measurements on wheat kernels for three varieties of wheat. A description of the variables is available at

http://archive.ics.uci.edu/ml/datasets/seeds

We are interested to find out if these measurements can be used to distinguish between the varieties. The data is stored in a `.txt` file, which we import using `read.table` (which works just like `read.csv`, but is tailored to text files) and convert the `Variety`

[7] A linear combination is a weighted sum of the form $a_1x_1 + a_2x_2 + \cdots + a_px_p$. If you like, you can think of principal components as weighted averages of variables, computed for each row in your data.

column to a categorical `factor` variable (which you'll learn more about in Section 5.4):

```
# The data is downloaded from the UCI Machine Learning Repository:
# http://archive.ics.uci.edu/ml/datasets/seeds
seeds <- read.table("https://tinyurl.com/seedsdata",
        col.names = c("Area", "Perimeter", "Compactness",
         "Kernel_length", "Kernel_width", "Asymmetry",
         "Groove_length", "Variety"))
seeds$Variety <- factor(seeds$Variety)
```

If we make a scatterplot matrix of all variables, it becomes evident that there are differences between the varieties, but that no single pair of variables is enough to separate them:

```
library(ggplot2)
library(GGally)
ggpairs(seeds[, -8], aes(colour = seeds$Variety, alpha = 0.2))
```

Moreover, for presentation purposes, the amount of information in the scatterplot matrix is a bit overwhelming. It would be nice to be able to present the data in a single scatterplot, without losing too much information. We'll therefore compute the principal components using the `prcomp` function. It is usually recommended that PCA is performed using standardised data, i.e. using data that has been scaled to have mean 0 and standard deviation 1. The reason for this is that it puts all variables on the same scale. If we don't standardise our data then variables with a high variance will completely dominate the principal components. This isn't desirable, as variance is affected by the scale of the measurements, meaning that the choice of measurement scale would influence the results (as an example, the variance of kernel length will be a million times greater if lengths are measured in millimetres instead of in metres).

We don't have to standardise the data ourselves, but can let `prcomp` do that for us using the arguments `center = TRUE` (to get mean 0) and `scale. = TRUE` (to get standard deviation 1):

```
# Compute principal components:
pca <- prcomp(seeds[,-8], center = TRUE, scale. = TRUE)
```

To see the *loadings* of the components, i.e. how much each variable contributes to the components, simply type the name of the object `prcomp` created:

```
pca
```

The first principal component is more or less a weighted average of all variables, but has stronger weights on `Area`, `Perimeter`, `Kernel_length`, `Kernel_width`, and `Groove_length`, all of which are measures of size. We can therefore interpret it as a size variable. The second component has higher loadings for `Compactness` and `Asymmetry`, meaning that it mainly measures those shape features. In Exercise 4.26

you'll see how the loadings can be visualised in a *biplot*.

To see how much of the variance each component represents, use `summary`:

```
summary(pca)
```

The first principal component accounts for 71.87 % of the variance, and the first three combined account for 98.67 %.

To visualise this, we can draw a *scree plot*, which shows the variance of each principal component - the total variance of the data is the sum of the variances of the principal components:

```
screeplot(pca, type = "lines")
```

We can use this to choose how many principal components to use when visualising or summarising our data. In that case, we look for an "elbow", i.e. a bend after which increases the number of components doesn't increase the amount of variance explained much.

We can access the values of the principal components using `pca$x`. Let's check that the first two components really are uncorrelated:

```
cor(pca$x[,1], pca$x[,2])
```

In this case, almost all of the variance is summarised by the first two or three principal components. It appears that we have successfully reduced the data from 7 variables to 2-3, which should make visualisation much easier. The `ggfortify` package contains an `autoplot` function for PCA objects, that creates a scatterplot of the first two principal components:

```
library(ggfortify)
autoplot(pca, data = seeds, colour = "Variety")
```

That is much better! The groups are almost completely separated, which shows that the variables can be used to discriminate between the three varieties. The first principal component accounts for 71.87 % of the total variance in the data, and the second for 17.11 %.

If you like, you can plot other pairs of principal components than just components 1 and 2. In this case, component 3 may be of interest, as its variance is almost as high as that of component 2. You can specify which components to plot with the `x` and `y` arguments:

```
# Plot 2nd and 3rd PC:
autoplot(pca, data = seeds, colour = "Variety",
         x = 2, y = 3)
```

Here, the separation is nowhere near as clear as in the previous figure. In this particular example, plotting the first two principal components is the better choice.

Judging from these plots, it appears that the kernel measurements can be used to discriminate between the three varieties of wheat. In Chapters 7 and 9 you'll learn how to use R to build models that can be used to do just that, e.g. by predicting which variety of wheat a kernel comes from given its measurements. If we wanted to build a statistical model that could be used for this purpose, we could use the original measurements. But we could also try using the first two principal components as the only input to the model. Principal component analysis is very useful as a preprocessing tool, used to create simpler models based on fewer variables (or ostensibly simpler, because the new variables are typically more difficult to interpret than the original ones).

~

Exercise 4.25. Use principal components on the `carat`, `x`, `y`, `z`, `depth`, and `table` variables in the `diamonds` data, and answer the following questions:

1. How much of the total variance does the first principal component account for? How many components are needed to account for at least 90 % of the total variance?

2. Judging by the loadings, what do the first two principal components measure?

3. What is the correlation between the first principal component and `price`?

4. Can the first two principal components be used to distinguish between diamonds with different cuts?

Exercise 4.26. Return to the scatterplot of the first two principal components for the `seeds` data, created above. Adding the arguments `loadings = TRUE` and `loadings.label = TRUE` to the `autoplot` call creates a *biplot*, which shows the loadings for the principal components on top of the scatterplot. Create a biplot and compare the result to those obtained by looking at the loadings numerically. Do the conclusions from the two approaches agree?

4.10 Cluster analysis

Cluster analysis is concerned with grouping observations into groups, *clusters*, that in some sense are similar. Numerous methods are available for this task, approaching the problem from different angles. Many of these are available in the `cluster` package, which ships with R. In this section, we'll look at a smorgasbord of clustering techniques.

4.10.1 Hierarchical clustering

As a first example where clustering can be of interest, we'll consider the `votes.repub` data from `cluster`. It describes the proportion of votes for the Republican candidate in US presidential elections from 1856 to 1976 in 50 different states:

```
library(cluster)
?votes.repub
View(votes.repub)
```

We are interested in finding subgroups - clusters - of states with similar voting patterns.

To find clusters of similar observations (states, in this case), we could start by assigning each observation to its own cluster. We'd then start with 50 clusters, one for each observation. Next, we could merge the two clusters that are the most similar, yielding 49 clusters, one of which consisted of two observations and 48 consisting of a single observation. We could repeat this process, merging the two most similar clusters in each iteration until only a single cluster was left. This would give us a *hierarchy* of clusters, which could be plotted in a tree-like structure, where observations from the same cluster would be one the same branch. Like this:

```
clusters_agnes <- agnes(votes.repub)
plot(clusters_agnes, which = 2)
```

This type of plot is known as a *dendrogram*.

We've just used `agnes`, a function from `cluster` that can be used to carry out *hierarchical clustering* in the manner described above. There are a couple of things that need to be clarified, though.

First, how do we measure how similar two p-dimensional observations x and y are? `agnes` provides two measures of distance between points:

- `metric = "euclidean"` (the default), uses the Euclidean L_2 distance $||x-y|| = \sqrt{\sum_{i=1}^{p}(x_i - y_i)^2}$,
- `metric = "manhattan"`, uses the Manhattan L_1 distance $||x - y|| = \sum_{i=1}^{p}|x_i - y_i|$.

Note that neither of these will work create if you have categorical variables in your data. If all your variables are binary, i.e. categorical with two values, you can use `mona` instead of `agnes` for hierarchical clustering.

Second, how do we measure how similar two clusters of observations are? `agnes` offers a number of options here. Among them are:

- `method = "average` (the default), unweighted average linkage, uses the average distance between points from the two clusters,

- `method = "single`, single linkage, uses the smallest distance between points from the two clusters,
- `method = "complete`, complete linkage, uses the largest distance between points from the two clusters,
- `method = "ward"`, Ward's method, uses the within-cluster variance to compare different possible clusterings, with the clustering with the lowest within-cluster variance being chosen.

Regardless of which of these that you use, it is often a good idea to standardise the numeric variables in your dataset so that they all have the same variance. If you don't, your distance measure is likely to be dominated by variables with larger variance, while variables with low variances will have little or no impact on the clustering. To standardise your data, you can use `scale`:

```
# Perform clustering on standardised data:
clusters_agnes <- agnes(scale(votes.repub))
# Plot dendrogram:
plot(clusters_agnes, which = 2)
```

At this point, we're starting to use several functions after another, and so this looks like a perfect job for a pipeline. To carry out the same analysis uses `%>%` pipes, we write:

```
library(magrittr)
votes.repub %>% scale() %>%
                agnes() %>%
                plot(which = 2)
```

We can now try changing the metric and clustering method used as described above. Let's use the Manhattan distance and complete linkage:

```
votes.repub %>% scale() %>%
                agnes(metric = "manhattan", method = "complete") %>%
                plot(which = 2)
```

We can change the look of the dendrogram by adding `hang = -1`, which causes all observations to be placed at the same level:

```
votes.repub %>% scale() %>%
                agnes(metric = "manhattan", method = "complete") %>%
                plot(which = 2, hang = -1)
```

As an alternative to `agnes`, we can consider `diana`. `agnes` is an *agglomerative* method, which starts with a lot of clusters and then merge them step-by-step. `diana`, in contrast, is a *divisive* method, which starts with one large cluster and then step-by-step splits it into several smaller clusters.

```
votes.repub %>% scale() %>%
                diana() %>%
                plot(which = 2)
```

You can change the distance measure used by setting `metric` in the `diana` call. Euclidean distance is the default.

To wrap this section up, we'll look at two packages that are useful for plotting the results of hierarchical clustering: `dendextend` and `factoextra`. We installed `factoextra` in the previous section, but still need to install `dendextend`:

```
install.packages("dendextend")
```

To compare the dendrograms from produced by different methods (or the same method with different settings), in a *tanglegram*, where the dendrograms are plotted against each other, we can use `tanglegram` from `dendextend`:

```
library(dendextend)
# Create clusters using agnes:
votes.repub %>% scale() %>%
                agnes() -> clusters_agnes
# Create clusters using diana:
votes.repub %>% scale() %>%
                diana() -> clusters_diana

# Compare the results:
tanglegram(as.dendrogram(clusters_agnes),
           as.dendrogram(clusters_diana))
```

Some clusters are quite similar here, whereas others are very different.

Often, we are interested in finding a comparatively small number of clusters, k. In hierarchical clustering, we can reduce the number of clusters by "cutting" the dendrogram tree. To do so using the `factoextra` package, we first use `hcut` to cut the tree into k parts, and then `fviz_dend` to plot the dendrogram, with each cluster plotted in a different colour. If, for instance, we want $k = 5$ clusters[8] and want to use `agnes` with average linkage and Euclidean distance for the clustering, we'd do the following:

```
library(factoextra)
votes.repub %>% scale() %>%
                hcut(k = 5, hc_func = "agnes",
                     hc_method = "average",
                     hc_metric = "euclidean") %>%
```

[8] Just to be clear, 5 is just an arbitrary number here. We could of course want 4, 14, or any other number of clusters.

```
                    fviz_dend()
```

There is no inherent meaning to the colours - they are simply a way to visually distinguish between clusters.

Hierarchical clustering is especially suitable for data with named observations. For other types of data, other methods may be better. We will consider some alternatives next.

~

Exercise 4.27. Continue the last example above by changing the clustering method to complete linkage with the Manhattan distance.

1. Do any of the 5 coloured clusters remain the same?
2. How can you produce a tanglegram with 5 coloured clusters, to better compare the results from the two clusterings?

Exercise 4.28. The `USArrests` data contains statistics on violent crime rates in 50 US states. Perform a hierarchical cluster analysis of the data. With which states are Maryland clustered?

4.10.2 Heatmaps and clustering variables

When looking at a dendrogram, you may ask why and how different observations are similar. Similarities between observations can be visualised using a *heatmap*, which displays the levels of different variables using colour hues or intensities. The `heatmap` function creates a heatmap from a `matrix` object. Let's try it with the `votes.repub` voting data. Because `votes.repub` is a `data.frame` object, we have to convert it to a matrix with `as.matrix` first (see Section 3.1.2):

```
library(cluster)
library(magrittr)
votes.repub %>% as.matrix() %>% heatmap()
```

You may want to increase the height of your Plot window so that the names of all states are displayed properly. Using the default colours, low values are represented by a light yellow and high values by a dark red. White represents missing values.

You'll notice that dendrograms are plotted along the margins. `heatmap` performs hierarchical clustering (by default, agglomerative with complete linkage) of the observations as well as of the variables. In the latter case, variables are grouped together based on *similarities between observations*, creating *clusters of variables*. In essence, this is just a hierarchical clustering of the transposed data matrix, but it does offer

a different view of the data, which at times can be very revealing. The rows and columns are sorted according to the two hierarchical clusterings.

As per usual, it is a good idea to standardise the data before clustering, which can be done using the `scale` argument in `heatmap`. There are two options for scaling, either in the row direction (preferable if you wish to cluster variables) or the column direction (preferable if you wish to cluster observations):

```
# Standardisation suitable for clustering variables:
votes.repub %>% as.matrix() %>% heatmap(scale = "row")

# Standardisation suitable for clustering observations:
votes.repub %>% as.matrix() %>% heatmap(scale = "col")
```

Looking at the first of these plots, we can see which elections (i.e. which variables) had similar outcomes in terms of Republican votes. For instance, we can see that the elections in 1960, 1976, 1888, 1884, 1880, and 1876 all had similar outcomes, with the large number of orange rows indicating that the Republicans neither did great nor did poorly.

If you like, you can change the colour palette used. As in Section 4.2.2, you can choose between palettes from http://www.colorbrewer2.org. `heatmap` is not a `ggplot2` function, so this is done in a slightly different way to what you're used to from other examples. Here are two examples, with the white-blue-purple sequential palette `"BuPu"` and the red-white-blue diverging palette `"RdBu"`:

```
library(RColorBrewer)
col_palette <- colorRampPalette(brewer.pal(8, "BuPu"))(25)
votes.repub %>% as.matrix() %>%
    heatmap(scale = "row", col = col_palette)

col_palette <- colorRampPalette(brewer.pal(8, "RdBu"))(25)
votes.repub %>% as.matrix() %>%
    heatmap(scale = "row", col = col_palette)
```

~

Exercise 4.29. Draw a heatmap for the `USArrests` data. Have a look at Maryland and the states with which it is clustered. Do they have high or low crime rates?

4.10.3 Centroid-based clustering

Let's return to the `seeds` data that we explored in Section 4.9:

```
# Download the data:
seeds <- read.table("https://tinyurl.com/seedsdata",
```

```
        col.names = c("Area", "Perimeter", "Compactness",
         "Kernel_length", "Kernel_width", "Asymmetry",
         "Groove_length", "Variety"))
seeds$Variety <- factor(seeds$Variety)
```

We know that there are three varieties of seeds in this dataset, but what if we didn't? Or what if we'd lost the labels and didn't know what seeds are of what type? There are no rows names for this data, and plotting a dendrogram may therefore not be that useful. Instead, we can use k-means clustering, where the points are clustered into k clusters based on their distances to the cluster means, or *centroids*.

When performing k-means clustering (using the algorithm of Hartigan & Wong (1979) that is the default in the function that we'll use), the data is split into k clusters based on their distance to the mean of all points. Points are then moved between clusters, one at a time, based on how close they are (as measured by Euclidean distance) to the mean of each cluster. The algorithm finishes when no point can be moved between clusters without increasing the average distance between points and the means of their clusters.

To run a k-means clustering in R, we can use `kmeans`. Let's start by using $k = 3$ clusters:

```
# First, we standardise the data, and then we do a k-means
# clustering.
# We ignore variable 8, Variety, which is the group label.
library(magrittr)
seeds[, -8] %>% scale() %>%
                kmeans(centers = 3) -> seeds_cluster

seeds_cluster
```

To visualise the results, we'll plot the first two principal components. We'll use colour to show the clusters. Moreover, we'll plot the different varieties in different shapes, to see if the clusters found correspond to different varieties:

```
# Compute principal components:
pca <- prcomp(seeds[,-8])
library(ggfortify)
autoplot(pca, data = seeds, colour = seeds_cluster$cluster,
         shape = "Variety", size = 2, alpha = 0.75)
```

In this case, the clusters more or less overlap with the varieties! Of course, in a lot of cases, we don't know the number of clusters beforehand. What happens if we change k?

First, we try $k = 2$:

```
seeds[, -8] %>% scale() %>%
                kmeans(centers = 2) -> seeds_cluster
autoplot(pca, data = seeds, colour = seeds_cluster$cluster,
         shape = "Variety", size = 2, alpha = 0.75)
```

Next, $k = 4$:

```
seeds[, -8] %>% scale() %>%
                kmeans(centers = 4) -> seeds_cluster
autoplot(pca, data = seeds, colour = seeds_cluster$cluster,
         shape = "Variety", size = 2, alpha = 0.75)
```

And finally, a larger number of clusters, say $k = 12$:

```
seeds[, -8] %>% scale() %>%
                kmeans(centers = 12) -> seeds_cluster
autoplot(pca, data = seeds, colour = seeds_cluster$cluster,
         shape = "Variety", size = 2, alpha = 0.75)
```

If it weren't for the fact that the different varieties were shown as different shapes, we'd have no way to say, based on this plot alone, which choice of k that is preferable here. Before we go into methods for choosing k though, we'll mention `pam`. `pam` is an alternative to k-means that works in the same way, but uses median-like points, *medoids* instead of cluster means. This makes it more robust to outliers. Let's try it with $k = 3$ clusters:

```
seeds[, -8] %>% scale() %>%
                pam(k = 3) -> seeds_cluster
autoplot(pca, data = seeds, colour = seeds_cluster$clustering,
         shape = "Variety", size = 2, alpha = 0.75)
```

For both `kmeans` and `pam`, there are visual tools that can help us choose the value of k in the `factoextra` package. Let's install it:

```
install.packages("factoextra")
```

The `fviz_nbclust` function in `factoextra` can be used to obtain plots that can guide the choice of k. It takes three arguments as input: the data, the clustering function (e.g. `kmeans` or `pam`) and the method used for evaluating different choices of k. There are three options for the latter: `"wss"`, `"silhouette"` and `"gap_stat"`.

`method = "wss"` yields a plot that relies on the within-cluster sum of squares, WSS, which is a measure of the within-cluster variation. The smaller this is, the more compact are the clusters. The WSS is plotted for several choices of k, and we look for an "elbow", just as we did when using a scree plot for PCA. That is, we look for the value of k such that increasing k further doesn't improve the WSS much. Let's have a look at an example, using `pam` for clustering:

```
library(factoextra)
fviz_nbclust(scale(seeds[, -8]), pam, method = "wss")

# Or, using a pipeline instead:
library(magrittr)
seeds[, -8] %>% scale() %>%
                fviz_nbclust(pam, method = "wss")
```

$k = 3$ seems like a good choice here.

`method = "silhouette"` produces a silhouette plot. The silhouette value measures how similar a point is compared to other points in its cluster. The closer to 1 this value is, the better. In a silhouette plot, the average silhouette value for points in the data are plotted against k:

```
fviz_nbclust(scale(seeds[, -8]), pam, method = "silhouette")
```

Judging by this plot, $k = 2$ appears to be the best choice.

Finally, `method = "gap_stat"` yields a plot of the gap statistic (Tibshirani et al., 2001), which is based on comparing the WSS to its expected value under a null distribution obtained using the bootstrap (Section 7.7). Higher values of the gap statistic are preferable:

```
fviz_nbclust(scale(seeds[, -8]), pam, method = "gap_stat")
```

In this case, $k = 3$ gives the best value.

In addition to plots for choosing k, `factoextra` provides the function `fviz_cluster` for creating PCA-based plots, with an option to add convex hulls or ellipses around the clusters:

```
# First, find the clusters:
seeds[, -8] %>% scale() %>%
                kmeans(centers = 3) -> seeds_cluster

# Plot clusters and their convex hulls:
library(factoextra)
fviz_cluster(seeds_cluster, data = seeds[, -8])

# Without row numbers:
fviz_cluster(seeds_cluster, data = seeds[, -8], geom = "point")

# With ellipses based on the multivariate normal distribution:
fviz_cluster(seeds_cluster, data = seeds[, -8],
             geom = "point", ellipse.type = "norm")
```

Note that in this plot, the shapes correspond to the clusters and not the varieties of seeds.

~

Exercise 4.30. The `chorSub` data from `cluster` contains measurements of 10 chemicals in 61 geological samples from the Kola Peninsula. Cluster this data using using either `kmeans` or `pam` (does either seem to be a better choice here?). What is a good choice of k here? Visualise the results.

4.10.4 Fuzzy clustering

An alternative to k-means clustering is *fuzzy clustering*, in which each point is "spread out" over the k clusters instead of being placed in a single cluster. The more similar it is to other observations in a cluster, the higher is its membership in that cluster. Points can have a high degree of membership to several clusters, which is useful in applications where points should be allowed to belong to more than one cluster. An important example is genetics, where genes can encode proteins with more than one function. If each point corresponds to a gene, it then makes sense to allow the points to belong to several clusters, potentially associated with different functions. The opposite of fuzzy clustering is *hard clustering*, in which each point only belongs to one cluster.

`fanny` from `cluster` can be used to perform fuzzy clustering:

```
library(cluster)
library(magrittr)
seeds[, -8] %>% scale() %>%
                fanny(k = 3) -> seeds_cluster

# Check membership of each cluster for the different points:
seeds_cluster$membership

# Plot the closest hard clustering:
library(factoextra)
fviz_cluster(seeds_cluster, geom = "point")
```

As for `kmeans` and `pam`, we can use `fviz_nbclust` to determine how many clusters to use:

```
seeds[, -8] %>% scale() %>%
                fviz_nbclust(fanny, method = "wss")
seeds[, -8] %>% scale() %>%
                fviz_nbclust(fanny, method = "silhouette")
# Producing the gap statistic plot takes a while here, so
```

```
# you may want to skip it in this case:
seeds[, -8] %>% scale() %>%
                fviz_nbclust(fanny, method = "gap")
```

~

Exercise 4.31. Do a fuzzy clustering of the `USArrests` data. Is Maryland strongly associated with a single cluster, or with several clusters? What about New Jersey?

4.10.5 Model-based clustering

As a last option, we'll consider model-based clustering, in which each cluster is assumed to come from a multivariate normal distribution. This will yield ellipsoidal clusters. `Mclust` from the `mclust` package fits such a model, called a Gaussian finite mixture model, using the EM-algorithm (Scrucca et al., 2016). First, let's install the package:

```
install.packages("mclust")
```

Now, let's cluster the `seeds` data. The number of clusters is chosen as part of the clustering procedure. We'll use a function from the `factoextra` for plotting the clusters with ellipsoids, and so start by installing that:

```
install.packages("factoextra")

library(mclust)
seeds_cluster <- Mclust(scale(seeds[, -8]))
summary(seeds_cluster)

# Plot results with ellipsoids:
library(factoextra)
fviz_cluster(seeds_cluster, geom = "point", ellipse.type = "norm")
```

Gaussian finite mixture models are based on the assumption that the data is numerical. For categorical data, we can use latent class analysis, which we'll discuss in Section 4.11.2, instead.

~

Exercise 4.32. Return to the `chorSub` data from Exercise 4.30. Cluster it using a Gaussian finite mixture model. How many clusters do you find?

4.10.6 Comparing clusters

Having found some interesting clusters, we are often interested in exploring differences between the clusters. To do so, we much first extract the cluster labels from our clustering (which are contained in the variables `clustering` for methods with Western female names, `cluster` for `kmeans`, and `classification` for `Mclust`). We can then add those labels to our data frame and use them when plotting.

For instance, using the `seeds` data, we can compare the area of seeds from different clusters:

```
# Cluster the seeds using k-means with k=3:
library(cluster)
seeds[, -8] %>% scale() %>%
                kmeans(centers = 3) -> seeds_cluster

# Add the results to the data frame:
seeds$clusters <- factor(seeds_cluster$cluster)
# Instead of $cluster, we'd use $clustering for agnes, pam, and fanny
# objects, and $classification for an Mclust object.

# Compare the areas of the 3 clusters using boxplots:
library(ggplot2)
ggplot(seeds, aes(x = Area, group = clusters, fill = clusters)) +
      geom_boxplot()

# Or using density estimates:
ggplot(seeds, aes(x = Area, group = clusters, fill = clusters)) +
      geom_density(alpha = 0.7)
```

We can also create a scatterplot matrix to look at all variables simultaneously:

```
library(GGally)
ggpairs(seeds[, -8], aes(colour = seeds$clusters, alpha = 0.2))
```

It may be tempting to run some statistical tests (e.g. a t-test) to see if there are differences between the clusters. Note, however, that in statistical hypothesis testing, it is typically assumed that the hypotheses that are being tested have been generated independently from the data. *Double-dipping*, where the data first is used to generate a hypothesis ("judging from this boxplot, there seems to be a difference in means between these two groups!" or "I found these clusters, and now I'll run a test to see if they are different") and then test that hypothesis, is generally frowned upon, as that substantially inflates the risk of a type I error. Recently, there have however been some advances in valid techniques for testing differences in means between clusters found using hierarchical clustering; see Gao et al. (2020).

4.11 Exploratory factor analysis

The purpose of *factor analysis* is to describe and understand the correlation structure for a set of observable variables through a smaller number of unobservable underlying variables, called *factors* or *latent variables*. These are thought to explain the values of the observed variables in a causal manner. Factor analysis is a popular tool in psychometrics, where it for instance is used to identify latent variables that explain people's results on different tests, e.g. related to personality, intelligence, or attitude.

4.11.1 Factor analysis

We'll use the `psych` package, along with the associated package `GPArotation`, for our analyses. Let's install them:

```
install.packages(c("psych", "GPArotation"))
```

For our first example of factor analysis, we'll be using the `attitude` data that comes with R. It describes the outcome of a survey of employees at a financial organisation. Have a look at its documentation to read about the variables in the dataset:

```
?attitude
attitude
```

To fit a factor analysis model to these data, we can use `fa` from `psych`. `fa` requires us to specify the number of factors used in the model. We'll get back to how to choose the number of factors, but for now, let's go with 2:

```
library(psych)
# Fit factor model:
attitude_fa <- fa(attitude, nfactors = 2,
                  rotate = "oblimin", fm = "ml")
```

`fa` does two things for us. First, it fits a factor model to the data, which yields a table of *factor loadings*, i.e. the correlation between the two unobserved factors and the observed variables. However, there is an infinite number of mathematically valid factor models for any given dataset. Therefore, the factors are *rotated* according to some rule to obtain a factor model that hopefully allows for easy and useful interpretation. Several methods can be used to fit the factor model (set using the `fm` argument in `fa`) and for rotation the solution (set using `rotate`). We'll look at some of the options shortly.

First, we'll print the result, showing the factor loadings (after rotation). We'll also plot the resulting model using `fa.diagram`, showing the correlation between the factors and the observed variables:

```
# Print results:
attitude_fa
```

```
# Plot results:
fa.diagram(attitude_fa, simple = FALSE)
```

The first factor is correlated to the variables `advance`, `learning` and `raises`. We can perhaps interpret this factor as measuring the employees' career opportunity at the organisation. The second factor is strongly correlated to `complaints` and (overall) `rating`, but also to a lesser degree correlated to `raises`, `learning` and `privileges`. This can maybe be interpreted as measuring how the employees' feel that they are treated at the organisation.

We can also see that the two factors are correlated. In some cases, it makes sense to expect the factors to be uncorrelated. In that case, we can change the rotation method used, from `oblimin` (which yields *oblique rotations*, allowing for correlations - usually a good default) to `varimax`, which yields uncorrelated factors:

```
attitude_fa <- fa(attitude, nfactors = 2,
                  rotate = "varimax", fm = "ml")
fa.diagram(attitude_fa, simple = FALSE)
```

In this case, the results are fairly similar.

The `fm = "ml"` setting means that maximum likelihood estimation of the factor model is performed, under the assumption of a normal distribution for the data. Maximum likelihood estimation is widely recommended for estimation of factor models, and can often work well even for non-normal data (Costello & Osborne, 2005). However, there are cases where it fails to find useful factors. `fa` offers several different estimation methods. A good alternative is `minres`, which often works well when maximum likelihood fails:

```
attitude_fa <- fa(attitude, nfactors = 2,
                  rotate = "oblimin", fm = "minres")
fa.diagram(attitude_fa, simple = FALSE)
```

Once again, the results are similar to what we saw before. In other examples, the results differ more. When choosing which estimation method and rotation to use, bear in mind that in an exploratory study, there is no harm in playing around with a few different methods. After all, your purpose is to generate hypotheses rather than confirm them, and looking at the data in a few different ways will help you do that.

To determine the number of factors that are appropriate for a particular dataset, we can draw a scree plot with `scree`. This is interpreted in the same way as for principal components analysis (Section 4.9) and centroid-based clustering (Section 4.10.3) - we look for an "elbow" in the plot, which tells us at which point adding more factors no longer contributes much to the model:

```
scree(attitude, pc = FALSE)
```

A useful alternative version of this is provided by `fa.parallel`, which adds lines showing what the scree plot would look like for randomly generated uncorrelated data of the same size as the original dataset. As long as the blue line, representing the actual data, is higher than the red line, representing randomly generated data, adding more factors improves the model:

```
fa.parallel(attitude, fm = "ml", fa = "fa")
```

Some older texts recommend that only factors with an eigenvalue (the y-axis in the scree plot) greater than 1 be kept in the model. It is widely agreed that this so-called Kaiser rule is inappropriate (Costello & Osborne, 2005), as it runs the risk of leaving out important factors.

Similarly, some older texts also recommend using principal components analysis to fit factor models. While the two are mathematically similar in that both in some sense reduce the dimensionality of the data, PCA and factor analysis are designed to target different problems. Factor analysis is concerned with an underlying causal structure where the unobserved factors affect the observed variables. In contrast, PCA simply seeks to create a small number of variables that summarise the variation in the data, which can work well even if there are no unobserved factors affecting the variables.

~

Exercise 4.33. Factor analysis only relies on the covariance or correlation matrix of your data. When using `fa` and other functions for factor analysis, you can input either a data frame or a covariance/correlation matrix. Read about the `ability.cov` data that comes shipped with R, and perform a factor analysis of it.

4.11.2 Latent class analysis

When there is a single categorical latent variable, factor analysis overlaps with clustering, which we studied in Section 4.10. Whether we think of the values of the latent variable as clusters, classes, factor levels, or something else is mainly a philosophical question - from a mathematical perspective, it doesn't matter what name we use for them.

When observations from the same cluster are assumed to be uncorrelated, the resulting model is called *latent profile analysis*, which typically is handled using model-based clustering (Section 4.10.5). The special case where the observed variables are categorical is instead known as *latent class analysis*. This is common e.g. in analyses of survey data, and we'll have a look at such an example in this section. The package that we'll use for our analyses is called `poLCA` - let's install it:

```
install.packages("poLCA")
```

The National Mental Health Services Survey is an annual survey collecting information about mental health treatment facilities in the US. We'll analyse data from the 2019 survey, courtesy of the Substance Abuse and Mental Health Data Archive, and try to find latent classes. Download `nmhss-puf-2019.csv` from the book's web page, and set `file_path` to its path. We can then load and look at a summary of the data using:

```
nmhss <- read.csv(file_path)
summary(nmhss)
```

All variables are categorical (except perhaps for the first one, which is an identifier). According to the survey's documentation, negative values are used to represent missing values. For binary variables, `0` means no/non-presence and `1` means yes/presence.

Next, we'll load the `poLCA` package and read the documentation for the function that we'll use for the analysis.

```
library(poLCA)
?poLCA
```

As you can see in the description of the `data` argument, the observed variables (called *manifest variables* here) are only allowed to contain consecutive integer values, starting from 1. Moreover, missing values should be represented by `NA`, and not by negative numbers (just as elsewhere in R!). We therefore need to make two changes to our data:

- Change negative values to `NA`,
- Change the levels of binary variables so that `1` means no/non-presence and `2` means yes/presence.

In our example, we'll look at variables describing what treatments are available at the different facilities. Let's create a new data frame for those variables:

```
treatments <- nmhss[, names(nmhss)[17:30]]
summary(treatments)
```

To make the changes to the data that we need, we can do the following:

```
# Change negative values to NA:
treatments[treatments < 0] <- NA

# Change binary variables from 0 and 1 to
# 1 and 2:
treatments <- treatments + 1

# Check the results:
summary(treatments)
```

We are now ready to get started with our analysis. To begin with, we will try to find classes based on whether or not the facilities offer the following five treatments:

- `TREATPSYCHOTHRPY`: The facility offers individual psychotherapy,
- `TREATFAMTHRPY`: The facility offers couples/family therapy,
- `TREATGRPTHRPY`: The facility offers group therapy,
- `TREATCOGTHRPY`: The facility offers cognitive behavioural therapy,
- `TREATPSYCHOMED`: The facility offers psychotropic medication. The `poLCA` function needs three inputs: a formula describing what observed variables to use, a data frame containing the observations, and `nclass`, the number of latent classes to find. To begin with, let's try two classes:

```
m <- poLCA(cbind(TREATPSYCHOTHRPY, TREATFAMTHRPY,
                 TREATGRPTHRPY, TREATCOGTHRPY,
                 TREATPSYCHOMED) ~ 1,
           data = treatments, nclass = 2)
```

The output shows the probabilities of 1's (no/non-presence) and 2's (yes/presence) for the two classes. So, for instance, from the output

```
$TREATPSYCHOTHRPY
           Pr(1)  Pr(2)
class 1:  0.6628 0.3372
class 2:  0.0073 0.9927
```

we gather that 34 % of facilities belonging to the first class offer individual psychotherapy, whereas 99 % of facilities from the second class offer individual psychotherapy. Looking at the other variables, we see that the second class always has high probabilities of offering therapies, while the first class doesn't. Interpreting this, we'd say that the second class contains facilities that offer a wide variety of treatments, and the first facilities that only offer some therapies. Finally, we see from the output that 88 % of the facilities belong to the second class:

```
Estimated class population shares
 0.1167 0.8833
```

We can visualise the class differences in a plot:

```
plot(m)
```

To see which classes different observations belong to, we can use:

```
m$predclass
```

Just as in a cluster analysis, it is often a good idea to run the analysis with different numbers of classes. Next, let's try 3 classes:

```
m <- poLCA(cbind(TREATPSYCHOTHRPY, TREATFAMTHRPY,
                 TREATGRPTHRPY, TREATCOGTHRPY,
```

```
                  TREATPSYCHOMED) ~ 1,
            data = treatments, nclass = 3)
```

This time, we run into numerical problems - the model estimation has failed, as indicated by the following warning message:

```
ALERT: iterations finished, MAXIMUM LIKELIHOOD NOT FOUND
```

`poLCA` fits the model using a method known as the *EM algorithm*, which finds maximum likelihood estimates numerically. First, the observations are randomly assigned to the classes. Step by step, the observations are then moved between classes, until the optimal split has been found. It can however happen that more steps are needed to find the optimum (by default 1,000 steps are used), or that we end up with unfortunate initial class assignments that prevent the algorithm from finding the optimum. To attenuate this problem, we can increase the number of steps used, or run the algorithm multiple times, each with new initial class assignments. The `poLCA` arguments for this are `maxiter`, which controls the number of steps (or iterations) used, and `nrep`, which controls the number of repetitions with different initial assignments. We'll increase both, and see if that helps. Note that this means that the algorithm will take longer to run:

```
m <- poLCA(cbind(TREATPSYCHOTHRPY, TREATFAMTHRPY,
                  TREATGRPTHRPY, TREATCOGTHRPY,
                  TREATPSYCHOMED) ~ 1,
            data = treatments, nclass = 3,
            maxiter = 2500, nrep = 5)
```

These setting should do the trick for this dataset, and you probably won't see a warning message this time. If you do, try increasing either number and run the code again.

The output that you get can differ between runs - in particular, the order of the classes can differ depending on initial assignments. Here is part of the output from my run:

```
$TREATPSYCHOTHRPY
           Pr(1)  Pr(2)
class 1:  0.0076 0.9924
class 2:  0.0068 0.9932
class 3:  0.6450 0.3550

$TREATFAMTHRPY
           Pr(1)  Pr(2)
class 1:  0.1990 0.8010
class 2:  0.0223 0.9777
class 3:  0.9435 0.0565
```

```
$TREATGRPTHRPY
           Pr(1)  Pr(2)
class 1:  0.0712 0.9288
class 2:  0.3753 0.6247
class 3:  0.4935 0.5065

$TREATCOGTHRPY
           Pr(1)  Pr(2)
class 1:  0.0291 0.9709
class 2:  0.0515 0.9485
class 3:  0.5885 0.4115

$TREATPSYCHOMED
           Pr(1)  Pr(2)
class 1:  0.0825 0.9175
class 2:  1.0000 0.0000
class 3:  0.3406 0.6594

Estimated class population shares
 0.8059 0.0746 0.1196
```

We can interpret this as follows:

- Class 1 (81 % of facilities): Offer all treatments, including psychotropic medication.
- Class 2 (7 % of facilities): Offer all treatments, except for psychotropic medication.
- Class 3 (12 % of facilities): Only offer some treatments, which may include psychotropic medication.

You can either let interpretability guide your choice of how many classes to include in your analysis, our use model fit measures like AIC and BIC, which are printed in the output and can be obtained from the model using:

```
m$aic
m$bic
```

The lower these are, the better is the model fit.

If you like, you can add a covariate to your latent class analysis, which allows you to simultaneously find classes and study their relationship with the covariate. Let's add the variable `PAYASST` (which says whether a facility offers treatment at no charge or minimal payment to clients who cannot afford to pay) to our data, and then use that as a covariate.

```
# Add PAYASST variable to data, then change negative values
# to NA's:
treatments$PAYASST <- nmhss$PAYASST
treatments$PAYASST[treatments$PAYASST < 0] <- NA

# Run LCA with covariate:
m <- poLCA(cbind(TREATPSYCHOTHRPY, TREATFAMTHRPY,
                 TREATGRPTHRPY, TREATCOGTHRPY,
                 TREATPSYCHOMED) ~ PAYASST,
           data = treatments, nclass = 3,
           maxiter = 2500, nrep = 5)
```

My output from this model includes the following tables:

```
==========================================================
Fit for 3 latent classes:
==========================================================
2 / 1
            Coefficient  Std. error  t value  Pr(>|t|)
(Intercept)     0.10616     0.18197    0.583     0.570
PAYASST         0.43302     0.11864    3.650     0.003
==========================================================
3 / 1
            Coefficient  Std. error  t value  Pr(>|t|)
(Intercept)     1.88482     0.20605    9.147         0
PAYASST         0.59124     0.10925    5.412         0
==========================================================
```

The interpretation is that both class 2 and class 3 differ significantly from class 1 (the p-values in the `Pr(>|t|)` column are low), with the positive coefficients for `PAYASST` telling us that class 2 and 3 facilities are more likely to offer pay assistance than class 1 facilities.

~

Exercise 4.34. The `cheating` dataset from `poLCA` contains students' answers to four questions about cheating, along with their grade point averages (GPA). Perform a latent class analysis using GPA as a covariate. What classes do you find? Does having a high GPA increase the probability of belonging to either class?

Chapter 5

Dealing with messy data

…or, put differently, *welcome to the real world*. Real datasets are seldom as tidy and clean as those you have seen in the previous examples in this book. On the contrary, real data is messy. Things will be out of place, and formatted in the wrong way. You'll need to filter the rows to remove those that aren't supposed to be used in the analysis. You'll need to remove some columns and merge others. You will need to wrestle, clean, coerce, and coax your data until it finally has the right format. Only then will you be able to actually analyse it.

This chapter contains a number of examples that serve as cookbook recipes for common data wrangling tasks. And as with any cookbook, you'll find yourself returning to some recipes more or less every day, until you know them by heart, while you never find the right time to use other recipes. You do definitely not have to know all of them by heart, and can always go back and look up a recipe that you need.

After working with the material in this chapter, you will be able to use R to:

- Handle numeric and categorical data,
- Manipulate and find patterns in text strings,
- Work with dates and times,
- Filter, subset, sort, and reshape your data using `data.table`, `dplyr`, and `tidyr`,
- Split and merge datasets,
- Scrape data from the web,
- Import data from different file formats.

5.1 Changing data types

In Exercise 3.1 you discovered that R implicitly coerces variables into other data types when needed. For instance, if you add a `numeric` to a `logical`, the result is

a `numeric`. And if you place them together in a vector, the vector will contain two `numeric` values:

```
TRUE + 5
v1 <- c(TRUE, 5)
v1
```

However, if you add a `numeric` to a `character`, the operation fails. If you put them together in a vector, both become `character` strings:

```
"One" + 5
v2 <- c("One", 5)
v2
```

There is a hierarchy for data types in R: `logical` < `integer` < `numeric` < `character`. When variables of different types are somehow combined (with addition, put in the same vector, and so on), R will coerce both to the higher ranking type. That is why `v1` contained `numeric` variables (`numeric` is higher ranked than `logical`) and `v2` contained `character` values (`character` is higher ranked than `numeric`).

Automatic coercion is often useful, but will sometimes cause problems. As an example, a vector of numbers may accidentally be converted to a `character` vector, which will confuse plotting functions. Luckily it is possible to convert objects to other data types. The functions most commonly used for this are `as.logical`, `as.numeric` and `as.character`. Here are some examples of how they can be used:

```
as.logical(1)            # Should be TRUE
as.logical("FALSE")      # Should be FALSE
as.numeric(TRUE)         # Should be 1
as.numeric("2.718282")   # Should be numeric 2.718282
as.character(2.718282)   # Should be the string "2.718282"
as.character(TRUE)       # Should be the string "TRUE"
```

A word of warning though - conversion only works if R can find a natural conversion between the types. Here are some examples where conversion fails. Note that only some of them cause warning messages:

```
as.numeric("two")                    # Should be 2
as.numeric("1+1")                    # Should be 2
as.numeric("2,718282")               # Should be numeric 2.718282
as.logical("Vaccines cause autism") # Should be FALSE
```

~

Exercise 5.1. The following tasks are concerned with converting and checking data types:

1. What happens if you apply `as.logical` to the `numeric` values 0 and 1? What happens if you apply it to other numbers?

2. What happens if you apply `as.character` to a vector containing `numeric` values?

3. The functions `is.logical`, `is.numeric` and `is.character` can be used to check if a variable is a `logical`, `numeric` or `character`, respectively. What type of object do they return?

4. Is `NA` a `logical`, `numeric` or `character`?

5.2 Working with lists

A data structure that is very convenient for storing data of different types is `list`. You can think of a `list` as a data frame where you can put different types of objects in each column: like a `numeric` vector of length 5 in the first, a data frame in the second and a single `character` in the third[1]. Here is an example of how to create a `list` using the function of the same name:

```
my_list <- list(my_numbers = c(86, 42, 57, 61, 22),
                my_data = data.frame(a = 1:3, b = 4:6),
                my_text = "Lists are the best.")
```

To access the elements in the list, we can use the same `$` notation as for data frames:

```
my_list$my_numbers
my_list$my_data
my_list$my_text
```

In addition, we can access them using indices, but using *double* brackets:

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

To access elements within the elements of lists, additional brackets can be added. For instance, if you wish to access the second element of the `my_numbers` vector, you can use either of these:

```
my_list[[1]][2]
my_list$my_numbers[2]
```

[1] In fact, the opposite is true: under the hood, a data frame is a list of vectors of equal length.

5.2.1 Splitting vectors into lists

Consider the `airquality` dataset, which among other things describe the temperature on each day during a five-month period. Suppose that we wish to split the `airquality$Temp` vector into five separate vectors: one for each month. We could do this by repeated filtering, e.g.

```
temp_may <- airquality$Temp[airquality$Month == 5]
temp_june <- airquality$Temp[airquality$Month == 6]
# ...and so on.
```

Apart from the fact that this isn't a very good-looking solution, this would be infeasible if we needed to split our vector into a larger number of new vectors. Fortunately, there is a function that allows us to split the vector by month, storing the result as a list - `split`:

```
temps <- split(airquality$Temp, airquality$Month)
temps

# To access the temperatures for June:
temps$`6`
temps[[2]]

# To give more informative names to the elements in the list:
names(temps) <- c("May", "June", "July", "August", "September")
temps$June
```

Note that, in breach of the rules for variable names in R, the original variable names here were numbers (actually `character` variables that happened to contain numeric characters). When accessing them using `$` notation, you need to put them between backticks (`` ` ``), e.g. `temps$`6``, to make it clear that `6` is a variable name and not a number.

5.2.2 Collapsing lists into vectors

Conversely, there are times where you want to collapse a list into a vector. This can be done using `unlist`:

```
unlist(temps)
```

~

Exercise 5.2. Load the `vas.csv` data from Exercise 3.8. Split the `VAS` vector so that you get a list containing one vector for each patient. How can you then access the VAS values for patient 212?

5.3 Working with numbers

A lot of data analyses involve numbers, which typically are represented as `numeric` values in R. We've already seen in Section 2.4.5 that there are numerous mathematical operators that can be applied to numbers in R. But there are also other functions that come in handy when working with numbers.

5.3.1 Rounding numbers

At times you may want to round numbers, either for presentation purposes or for some other reason. There are several functions that can be used for this:

```
a <- c(2.1241, 3.86234, 4.5, -4.5, 10000.1001)
round(a, 3)           # Rounds to 3 decimal places
signif(a, 3)          # Rounds to 3 significant digits
ceiling(a)            # Rounds up to the nearest integer
floor(a)              # Rounds down to the nearest integer
trunc(a)              # Rounds to the nearest integer, toward 0
                      # (note the difference in how 4.5
                      #  and -4.5 are treated!)
```

5.3.2 Sums and means in data frames

When working with numerical data, you'll frequently find yourself wanting to compute sums or means of either columns or rows of data frames. The `colSums`, `rowSums`, `colMeans` and `rowMeans` functions can be used to do this. Here is an example with an expanded version of the `bookstore` data, where three purchases have been recorded for each customer:

```
bookstore2 <- data.frame(purchase1 = c(20, 59, 2, 12, 22, 160,
                                       34, 34, 29),
                         purchase2 = c(14, 67, 9, 20, 20, 81,
                                       19, 55, 8),
                         purchase3 = c(4, 62, 11, 18, 33, 57,
                                       24, 49, 29))

colSums(bookstore2)   # The total amount for customers' 1st, 2nd and
                      # 3rd purchases
rowSums(bookstore2)   # The total amount for each customer
colMeans(bookstore2)  # Mean purchase for 1st, 2nd and 3rd purchases
rowMeans(bookstore2)  # Mean purchase for each customer
```

Moving beyond sums and means, in Section 6.5 you'll learn how to apply any function to the rows or columns of a data frame.

5.3.3 Summaries of series of numbers

When a `numeric` vector contains a series of consecutive measurements, as is the case e.g. in a time series, it is often of interest to compute various cumulative summaries. For instance, if the vector contains the daily revenue of a business during a month, it may be of value to know the total revenue up to each day - that is, the *cumulative sum* for each day.

Let's return to the `a10` data from Section 4.6, which described the monthly anti-diabetic drug sales in Australia during 1991-2008.

```
library(fpp2)
a10
```

Elements 7 to 18 contain the sales for 1992. We can compute the total, highest and smallest monthly sales up to and including each month using `cumsum`, `cummax` and `cummin`:

```
a10[7:18]
cumsum(a10[7:18])  # Total sales
cummax(a10[7:18])  # Highest monthly sales
cummin(a10[7:18])  # Lowest monthly sales

# Plot total sales up to and including each month:
plot(1:12, cumsum(a10[7:18]),
     xlab = "Month",
     ylab = "Total sales",
     type = "b")
```

In addition, the `cumprod` function can be used to compute cumulative products.

At other times, we are interested in studying *run lengths* in series, that is, the lengths of runs of equal values in a vector. Consider the `upp_temp` vector defined in the code chunk below, which contains the daily temperatures in Uppsala, Sweden, in February 2020[2].

```
upp_temp <- c(5.3, 3.2, -1.4, -3.4, -0.6, -0.6, -0.8, 2.7, 4.2, 5.7,
              3.1, 2.3, -0.6, -1.3, 2.9, 6.9, 6.2, 6.3, 3.2, 0.6, 5.5,
              6.1, 4.4, 1.0, -0.4, -0.5, -1.5, -1.2, 0.6)
```

It could be interesting to look at runs of sub-zero days, i.e. consecutive days with sub-zero temperatures. The `rle` function counts the lengths of runs of equal values in a vector. To find the length of runs of temperatures below or above zero we can use the vector defined by the condition `upp_temp < 0`, the values of which are `TRUE` on sub-zero days and `FALSE` when the temperature is 0 or higher. When we apply `rle` to this vector, it returns the length and value of the runs:

[2]Courtesy of the Department of Earth Sciences at Uppsala University.

```
rle(upp_temp < 0)
```

We first have a 2-day run of above zero temperatures (`FALSE`), then a 5-day run of sub-zero temperatures (`TRUE`), then a 5-day run of above zero temperatures, and so on.

5.3.4 Scientific notation `1e-03`

When printing very large or very small numbers, R uses *scientific notation*, meaning that $7,000,000$ (7 followed by 6 zeroes) is displayed as (the mathematically equivalent) $7 \cdot 10^6$ and 0.0000007 is displayed as $7 \cdot 10^{-7}$. Well, almost, the *ten raised to the power of x* bit isn't really displayed as 10^x, but as `e+x`, a notation used in many programming languages and calculators. Here are some examples:

```
7000000
0.0000007
7e+07
exp(30)
```

Scientific notation is a convenient way to display large numbers, but it's not always desirable. If you just want to print the number, the `format` function can be used to convert it to a character, suppressing scientific notation:

```
format(7000000, scientific = FALSE)
```

If you still want your number to be a `numeric` (as you often do), a better choice is to change the option for when R uses scientific notation. This can be done using the `scipen` argument in the `options` function:

```
options(scipen = 1000)
7000000
0.0000007
7e+07
exp(30)
```

To revert this option back to the default, you can use:

```
options(scipen = 0)
7000000
0.0000007
7e+07
exp(30)
```

Note that this option only affects how R *prints* numbers, and not how they are treated in computations.

5.3.5 Floating point arithmetics

Some numbers cannot be written in finite decimal forms. Take 1/3 for example, the decimal form of which is

$$0.3333333333333333333333333333333333\ldots.$$

Clearly, the computer cannot store this number exactly, as that would require an infinite memory[3]. Because of this, numbers in computers are stored as *floating point numbers*, which aim to strike a balance between *range* (being able to store both very small and very large numbers) and *precision* (being able to represent numbers accurately). Most of the time, calculations with floating points yield exactly the results that we'd expect, but sometimes these non-exact representations of numbers will cause unexpected problems. If we wish to compute $1.5 - 0.2$ and $1.1 - 0.2$, say, we could of course use R for that. Let's see if it gets the answers right:

```
1.5 - 0.2
1.5 - 0.2 == 1.3  # Check if 1.5-0.2=1.3
1.1 - 0.2
1.1 - 0.2 == 0.9  # Check if 1.1-0.2=0.9
```

The limitations of floating point arithmetics causes the second calculation to fail. To see what has happened, we can use `sprintf` to print numbers with 30 decimals (by default, R prints a rounded version with fewer decimals):

```
sprintf("%.30f", 1.1 - 0.2)
sprintf("%.30f", 0.9)
```

The first 12 decimals are identical, but after that the two numbers `1.1 - 0.2` and `0.9` diverge. In our other example, $1.5 - 0.2$, we don't encounter this problem - both `1.5 - 0.2` and `0.3` have the same floating point representation:

```
sprintf("%.30f", 1.5 - 0.2)
sprintf("%.30f", 1.3)
```

The order of the operations also matters in this case. The following three calculations would all yield identical results if performed with real numbers, but in floating point arithmetics the results differ:

```
1.1 - 0.2 - 0.9
1.1 - 0.9 - 0.2
1.1 - (0.9 + 0.2)
```

In most cases, it won't make a difference whether a variable is represented as 0.90000000000000013... or 0.90000000000000002..., but in some cases tiny differences like that can propagate and cause massive problems. A famous example of this

[3]This is not strictly speaking true; if we use base 3, 1/3 is written as 0.1 which can be stored in a finite memory. But then other numbers become problematic instead.

involves the US Patriot surface-to-air defence system, which at the end of the first Gulf war missed an incoming missile due to an error in floating point arithmetics[4]. It is important to be aware of the fact that floating point arithmetics occasionally will yield incorrect results. This can happen for numbers of any size, but is more likely to occur when very large and very small numbers appear in the same computation.

So, `1.1 - 0.2` and `0.9` may not be the same thing in floating point arithmetics, but at least they are *nearly* the same thing. The `==` operator checks if two numbers are exactly equal, but there is an alternative that can be used to check if two numbers are nearly equal: `all.equal`. If the two numbers are (nearly) equal, it returns `TRUE`, and if they are not, it returns a description of how they differ. In order to avoid the latter, we can use the `isTRUE` function to return `FALSE` instead:

```
1.1 - 0.2 == 0.9
all.equal(1.1 - 0.2, 0.9)
all.equal(1, 2)
isTRUE(all.equal(1, 2))
```

~

Exercise 5.3. These tasks showcase some problems that are commonly faced when working with numeric data:

1. The vector `props <- c(0.1010, 0.2546, 0.6009, 0.0400, 0.0035)` contains proportions (which, by definition, are between 0 and 1). Convert the proportions to percentages with one decimal place.

2. Compute the highest and lowest temperatures up to and including each day in the `airquality` dataset.

3. What is the longest run of days with temperatures above 80 in the `airquality` dataset?

Exercise 5.4. These tasks are concerned with floating point arithmetics:

1. Very large numbers, like `10e500`, are represented by `Inf` (infinity) in R. Try to find out what the largest number that can be represented as a floating point number in R is.

2. Due to an error in floating point arithmetics, `sqrt(2)^2 - 2` is not equal to `0`. Change the order of the operations so that the results is `0`.

[4]Not in R though.

5.4 Working with factors

In Sections 2.6.2 and 2.8 we looked at how to analyse and visualise categorical data, i.e data where the variables can take a fixed number of possible values that somehow correspond to groups or categories. But so far we haven't really gone into how to handle categorical variables in R.

Categorical data is stored in R as `factor` variables. You may ask why a special data structure is needed for categorical data, when we could just use `character` variables to represent the categories. Indeed, the latter is what R does by default, e.g. when creating a `data.frame` object or reading data from `.csv` and `.xlsx` files.

Let's say that you've conducted a survey on students' smoking habits. The possible responses are *Never*, *Occasionally*, *Regularly* and *Heavy*. From 10 students, you get the following responses:

```
smoke <- c("Never", "Never", "Heavy", "Never", "Occasionally",
           "Never", "Never", "Regularly", "Regularly", "No")
```

Note that the last answer is invalid - `No` was not one of the four answers that were allowed for the question.

You could use `table` to get a summary of how many answers of each type that you got:

```
table(smoke)
```

But the categories are not presented in the correct order! There is a clear order between the different categories, *Never* < *Occasionally* < *Regularly* < *Heavy*, but `table` doesn't present the results in that way. Moreover, R didn't recognise that `No` was an invalid answer, and treats it just the same as the other categories.

This is where `factor` variables come in. They allow you to specify which values your variable can take, and the ordering between them (if any).

5.4.1 Creating factors

When creating a `factor` variable, you typically start with a `character`, `numeric` or `logical` variable, the values of which are turned into categories. To turn the `smoke` vector that you created in the previous section into a `factor`, you can use the `factor` function:

```
smoke2 <- factor(smoke)
```

You can inspect the elements, and *levels*, i.e. the values that the categorical variable takes, as follows:

```
smoke2
levels(smoke2)
```

So far, we haven't solved neither the problem of the categories being in the wrong order nor that invalid `No` value. To fix both these problems, we can use the `levels` argument in `factor`:

```
smoke2 <- factor(smoke, levels = c("Never", "Occasionally",
                                   "Regularly", "Heavy"),
                        ordered = TRUE)

# Check the results:
smoke2
levels(smoke2)
table(smoke2)
```

You can control the order in which the levels are presented by choosing which order we write them in in the `levels` argument. The `ordered = TRUE` argument specifies that the order of the variables is *meaningful*. It can be excluded in cases where you wish to specify the order in which the categories should be presented purely for presentation purposes (e.g. when specifying whether to use the order `Male/Female/Other` or `Female/Male/Other`). Also note that the `No` answer now became an `NA`, which in the case of `factor` variables represents both missing observations and invalid observations. To find the values of `smoke` that became `NA` in `smoke2` you can use `which` and `is.na`:

```
smoke[which(is.na(smoke2))]
```

By checking the original values of the `NA` elements, you can see if they should be excluded from the analysis or recoded into a proper category (`No` could for instance be recoded into `Never`). In Section 5.5.3 you'll learn how to replace values in larger datasets automatically using regular expressions.

5.4.2 Changing factor levels

When we created `smoke2`, one of the elements became an `NA`. `NA` was however not included as a level of the `factor`. Sometimes it is desirable to include `NA` as a level, for instance when you want to analyse rows with missing data. This is easily done using the `addNA` function:

```
smoke2 <- addNA(smoke2)
```

If you wish to change the name of one or more of the `factor` levels, you can do it directly via the `levels` function. For instance, we can change the name of the `NA` category, which is the 5th level of `smoke2`, as follows:

```
levels(smoke2)[5] <- "Invalid answer"
```

The above solution is a little brittle in that it relies on specifying the index of the level name, which can change if we're not careful. More robust solutions using the `data.table` and `dplyr` packages are presented in Section 5.7.6.

Finally, if you've added more levels than what are actually used, these can be dropped using the `droplevels` function:

```
smoke2 <- factor(smoke, levels = c("Never", "Occasionally",
                                   "Regularly", "Heavy",
                                   "Constantly"),
                        ordered = TRUE)
levels(smoke2)
smoke2 <- droplevels(smoke2)
levels(smoke2)
```

5.4.3 Changing the order of levels

Now suppose that we'd like the levels of the `smoke2` variable to be presented in the reverse order: *Heavy*, *Regularly*, *Occasionally*, and *Never*. This can be done by a new call to `factor`, where the new level order is specified in the `levels` argument:

```
smoke2 <- factor(smoke2, levels = c("Heavy", "Regularly",
                                    "Occasionally", "Never"))

# Check the results:
levels(smoke2)
```

5.4.4 Combining levels

Finally, `levels` can be used to merge categories by replacing their separate names with a single name. For instance, we can combine the smoking categories *Occasionally*, *Regularly*, and *Heavy* to a single category named *Yes*. Assuming that these are first, second and third in the list of names (as will be the case if you've run the last code chunk above), here's how to do it:

```
levels(smoke2)[1:3] <- "Yes"

# Check the results:
levels(smoke2)
```

Alternative ways to do this are presented in Section 5.7.6.

~

Exercise 5.5. In Exercise 3.7 you learned how to create a `factor` variable from a `numeric` variable using `cut`. Return to your solution (or the solution at the back of the book) and do the following:

1. Change the category names to `Mild`, `Moderate` and `Hot`.

2. Combine `Moderate` and `Hot` into a single level named `Hot`.

Exercise 5.6. Load the `msleep` data from the `ggplot2` package. Note that categorical variable `vore` is stored as a `character`. Convert it to a `factor` by running `msleep$vore <- factor(msleep$vore)`.

1. How are the resulting factor levels ordered? Why are they ordered in that way?
2. Compute the mean value of `sleep_total` for each `vore` group.
3. Sort the factor levels according to their `sleep_total` means. Hint: this can be done manually, or more elegantly using e.g. a combination of the functions `rank` and `match` in an intermediate step.

5.5 Working with strings

Text in R is represented by `character` strings. These are created using double or single quotes. I recommend double quotes for three reasons. First, it is the default in R, and is the recommended style (see e.g. `?Quotes`). Second, it improves readability - code with double quotes is easier to read because double quotes are easier to spot than single quotes. Third, it will allow you to easily use apostrophes in your strings, which single quotes don't (because apostrophes will be interpreted as the end of the string). Single quotes can however be used if you need to include double quotes inside your string:

```
# This works:
text1 <- "An example of a string. Isn't this great?"
text2 <- 'Another example of a so-called "string".'

# This doesn't work:
text1_fail <- 'An example of a string. Isn't this great?'
text2_fail <- "Another example of a so-called "string"."
```

If you check what these two strings look like, you'll notice something funny about `text2`:

```
text1
text2
```

R has put backslash characters, \, before the double quotes. The backslash is called an *escape character*, which invokes a different interpretation of the character that follows it. In fact, you can use this to put double quotes inside a string that you define using double quotes:

```
text2_success <- "Another example of a so-called \"string\"."
```

There are a number of other special characters that can be included using a backslash:

`\n` for a line break (a new line) and `\t` for a tab (a long whitespace) being the most important[5]:

```
text3 <- "Text...\n\tWith indented text on a new line!"
```

To print your string in the Console in a way that shows special characters instead of their escape character-versions, use the function `cat`:

```
cat(text3)
```

You can also use `cat` to print the string to a text file…

```
cat(text3, file = "new_findings.txt")
```

…and to append text at the end of a text file:

```
cat("Let's add even more text!", file = "new_findings.txt",
    append = TRUE)
```

(Check the output by opening `new_findings.txt`!)

5.5.1 Concatenating strings

If you wish to concatenate multiple strings, `cat` will do that for you:

```
first <- "This is the beginning of a sentence"
second <- "and this is the end."
cat(first, second)
```

By default, `cat` places a single white space between the two strings, so that `"This is the beginning of a sentence"` and `"and this is the end."` are concatenated to `"This is the beginning of a sentence and this is the end."`. You can change that using the `sep` argument in `cat`. You can also add as many strings as you like as input:

```
cat(first, second, sep = "; ")
cat(first, second, sep = "\n")
cat(first, second, sep = "")
cat(first, second, "\n", "And this is another sentence.")
```

At other times, you want to concatenate two or more strings without printing them. You can then use `paste` in exactly the same way as you'd use `cat`, the exception being that `paste` returns a string instead of printing it.

```
my_sentence <- paste(first, second, sep = "; ")
my_novel <- paste(first, second, "\n",
                  "And this is another sentence.")
```

[5] See `?Quotes` for a complete list.

```
# View results:
my_sentence
my_novel
cat(my_novel)
```

Finally, if you wish to create a number of similar strings based on information from other variables, you can use `sprintf`, which allows you to write a string using `%s` as a placeholder for the values that should be pulled from other variables:

```
names <- c("Irma", "Bea", "Lisa")
ages <- c(5, 59, 36)

sprintf("%s is %s years old.", names, ages)
```

There are many more uses of `sprintf` (we've already seen some in Section 5.3.5), but this enough for us for now.

5.5.2 Changing case

If you need to translate characters from lowercase to uppercase or vice versa, that can be done using `toupper` and `tolower`:

```
my_string <- "SOMETIMES I SCREAM (and sometimes I whisper)."
toupper(my_string)
tolower(my_string)
```

If you only wish to change the case of some particular element in your string, you can use `substr`, which allows you to access substrings:

```
months <- c("january", "february", "march", "aripl")

# Replacing characters 2-4 of months[4] with "pri":
substr(months[4], 2, 4) <- "pri"
months

# Replacing characters 1-1 (i.e. character 1) of each element of month
# with its uppercase version:
substr(months, 1, 1) <- toupper(substr(months, 1, 1))
months
```

5.5.3 Finding patterns using regular expressions

Regular expressions, or regexps for short, are special strings that describe patterns. They are extremely useful if you need to find, replace or otherwise manipulate a number of strings depending on whether or not a certain pattern exists in each one of them. For instance, you may want to find all strings containing only numbers and

convert them to `numeric`, or find all strings that contain an email address and remove said addresses (for censoring purposes, say). Regular expressions are incredibly useful, but can be daunting. Not everyone will need them, and if this all seems a bit too much to you can safely skip this section, or just skim through it, and return to it at a later point.

To illustrate the use of regular expressions we will use a sheet from the `projects-email.xlsx` file from the books' web page. In Exercise 3.9, you explored the second sheet in this file, but here we'll use the third instead. Set `file_path` to the path to the file, and then run the following code to import the data:

```
library(openxlsx)
contacts <- read.xlsx(file_path, sheet = 3)
str(contacts)
```

There are now three variables in `contacts`. We'll primarily be concerned with the third one: `Address`. Some people have email addresses attached to them, others have postal addresses and some have no address at all:

```
contacts$Address
```

You can find loads of guides on regular expressions online, but few of them are easy to use with R, the reason being that regular expressions in R sometimes require escape characters that aren't needed in some other programming languages. In this section we'll take a look at regular expressions, *as they are written in R*.

The basic building blocks of regular expressions are patterns consisting of one or more characters. If, for instance, we wish to find all occurrences of the letter `y` in a vector of strings, the regular expression describing that "pattern" is simply `"y"`. The functions used to find occurrences of patterns are called `grep` and `grepl`. They differ only in the output they return: `grep` returns the indices of the strings containing the pattern, and `grepl` returns a `logical` vector with `TRUE` at indices matching the patterns and `FALSE` at other indices.

To find all addresses containing a lowercase `y`, we use `grep` and `grepl` as follows:

```
grep("y", contacts$Address)
grepl("y", contacts$Address)
```

Note how both outputs contain the same information presented in different ways.

In the same way, we can look for word or substrings. For instance, we can find all addresses containing the string `"Edin"`:

```
grep("Edin", contacts$Address)
grepl("Edin", contacts$Address)
```

Similarly, we can also look for special characters. Perhaps we can find all email

addresses by looking for strings containing the `@` symbol:

```
grep("@", contacts$Address)
grepl("@", contacts$Address)

# To display the addresses matching the pattern:
contacts$Address[grep("@", contacts$Address)]
```

Interestingly, this includes two rows that aren't email addresses. To separate the email addresses from the other rows, we'll need a more complicated regular expression, describing the pattern of an email address in more general terms. Here are four examples or regular expressions that'll do the trick:

```
grep(".+@.+[.].+", contacts$Address)
grep(".+@.+\\..+", contacts$Address)
grep("[[:graph:]]+@[[:graph:]]+[.][[:alpha:]]+", contacts$Address)
grep("[[:alnum:].-]+@[[:alnum:].-]+[.][[:alpha:]]+",
     contacts$Address)
```

To try to wrap our head around what these mean we'll have a look at the building blocks of regular expressions. These are:

- Patterns describing a single character.
- Patterns describing a class of characters, e.g. letters or numbers.
- Repetition quantifiers describing how many repetitions of a pattern to look for.
- Other operators.

We've already looked at single character expressions, as well as the multi-character expression `"Edin"` which simply is a combination of four single-character expressions. Patterns describing classes of characters, e.g. characters with certain properties, are denoted by brackets `[]` (for manually defined classes) or double brackets `[[]]` (for predefined classes). One example of the latter is `"[[:digit:]]` which is a pattern that matches all digits: `0 1 2 3 4 5 6 7 8 9`. Let's use it to find all addresses containing a number:

```
grep("[[:digit:]]", contacts$Address)
contacts$Address[grep("[[:digit:]]", contacts$Address)]
```

Some important predefined classes are:

- `[[:lower:]]` matches lowercase letters,
- `[[:upper:]]` matches UPPERCASE letters,
- `[[:alpha:]]` matches both lowercase and UPPERCASE letters,
- `[[:digit:]]` matches digits: `0 1 2 3 4 5 6 7 8 9`,
- `[[:alnum:]]` matches alphanumeric characters (alphabetic characters and digits),
- `[[:punct:]]` matches punctuation characters: `! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ { | } ~`‘,

- `[[:space:]]` matches space characters: space, tab, newline, and so on,
- `[[:graph:]]` matches letters, digits, and punctuation characters,
- `[[:print:]]` matches letters, digits, punctuation characters, and space characters,
- `.` matches *any* character.

Examples of manually defined classes are:

- `[abcd]` matches `a`, `b`, `c`, and `d`,
- `[a-d]` matches `a`, `b`, `c`, and `d`,
- `[aA12]` matches `a`, `A`, `1` and `2`,
- `[.]` matches `.`,
- `[.,]` matches `.` and `,`,
- `[^abcd]` matches anything except `a`, `b`, `c`, or `d`.

So for instance, we can find all addresses that don't contain at least one of the letters `y` and `z` using:

```
grep("[^yz]", contacts$Address)
contacts$Address[grep("[^yz]", contacts$Address)]
```

All of these patterns can be combined with patterns describing a single character:

- `gr[ea]y` matches `grey` and `gray` (but not `greay`!),
- `b[^o]g` matches `bag`, `beg`, and similar strings, but not `bog`,
- `[.]com` matches `.com`.

When using the patterns above, you only look for a single occurrence of the pattern. Sometimes you may want a pattern like *a word of 2-4 letters* or *any number of digits in a row*. To create these, you add repetition patterns to your regular expression:

- `?` means that the preceding patterns is matched *at most once*, i.e. 0 or 1 time,
- `*` means that the preceding pattern is matched *0 or more* times,
- `+` means that the preceding pattern is matched *at least once*, i.e. 1 time or more,
- `{n}` means that the preceding pattern is matched *exactly* `n` times,
- `{n,}` means that the preceding pattern is matched *at least* `n` times, i.e. `n` times or more,
- `{n,m}` means that the preceding pattern is matched *at least* `n` times *but not more than* `m` times.

Here are some examples of how repetition patterns can be used:

```
# There are multiple ways of finding strings containing two n's
# in a row:
contacts$Address[grep("nn", contacts$Address)]
contacts$Address[grep("n{2}", contacts$Address)]

# Find strings with words beginning with an uppercase letter, followed
# by at least one lowercase letter:
```

```
contacts$Address[grep("[[:upper:]][[:lower:]]+", contacts$Address)]

# Find strings with words beginning with an uppercase letter, followed
# by at least six lowercase letters:
contacts$Address[grep("[[:upper:]][[:lower:]]{6,}", contacts$Address)]

# Find strings containing any number of letters, followed by any
# number of digits, followed by a space:
contacts$Address[grep("[[:alpha:]]+[[:digit:]]+[[:space:]]",
                      contacts$Address)]
```

Finally, there are some other operators that you can use to create even more complex patterns:

- `|` alteration, picks one of multiple possible patterns. For example, `ab|bc` matches `ab` or `bc`.
- `()` parentheses are used to denote a subset of an expression that should be evaluated separately. For example, `colo|our` matches `colo` or `our` while `col(o|ou)r` matches `color` or `colour`.
- `^`, when used outside of brackets `[]`, means that the match should be found at the start of the string. For example, `^a` matches strings beginning with `a`, but not `"dad"`.
- `$` means that the match should be found at the end of the string. For example, `a$` matches strings ending with `a`, but not `"dad"`.
- `\\` escape character that can be used to match special characters like `.`, `^` and `$` (`\\.`, `\\^`, `\\$`).

This may seem like a lot (and it is!), but there are in fact many more possibilities when working with regular expression. For the sake of some sorts of brevity, we'll leave it at this for now though.

Let's return to those email addresses. We saw three regular expressions that could be used to find them:

```
grep(".+@.+[.].+", contacts$Address)
grep(".+@.+\\..+", contacts$Address)
grep("[[:graph:]]+@[[:graph:]]+[.][[:alpha:]]+", contacts$Address)
grep("[[:alnum:]._-]+@[[:alnum:]._-]+[.][[:alpha:]]+",
     contacts$Address)
```

The first two of these both specify the same pattern: *any number of any characters, followed by an `@`, followed by any number of any characters, followed by a period `.`, followed by any number of characters.* This will match email addresses, but would also match strings like `"?=)(/x@!.a??"`, which isn't a valid email address. In this case, that's not a big issue, as our goal was to find addresses that looked like email addresses, and not to verify that the addresses were valid.

The third alternative has a slightly different pattern: *any number of letters, digits, and punctuation characters, followed by an* `@`*, followed by any number of letters, digits, and punctuation characters, followed by a period* `.`*, followed by any number of letters.* This too would match `"?=)(/x@!.a??"` as it allows punctuation characters that don't usually occur in email addresses. The fourth alternative, however, won't match `"?=)(/x@!.a??"` as it only allows letters, digits and the symbols `.`, `_` and `-` in the name and domain name of the address.

5.5.4 Substitution

An important use of regular expressions is in substitutions, where the parts of strings that match the pattern in the expression are replaced by another string. There are two email addresses in our data that contain `(a)` instead of `@`:

```
contacts$Address[grep("[(]a[])]", contacts$Address)]
```

If we wish to replace the `(a)` by `@`, we can do so using `sub` and `gsub`. The former replaces only the *first* occurrence of the pattern in the input vector, whereas the latter replaces *all* occurrences.

```
contacts$Address[grep("[(]a[])]", contacts$Address)]
sub("[(]a[])]", "@", contacts$Address)  # Replace first occurrence
gsub("[(]a[])]", "@", contacts$Address) # Replace all occurrences
```

5.5.5 Splitting strings

At times you want to extract only a part of a string, for example if measurements recorded in a column contains units, e.g. `66.8 kg` instead of `66.8`. To split a string into different parts, we can use `strsplit`.

As an example, consider the email addresses in our `contacts` data. Suppose that we want to extract the user names from all email addresses, i.e. remove the `@domain.topdomain` part. First, we store all email addresses from the data in a new vector, and then we split them at the `@` sign:

```
emails <- contacts$Address[grepl(
             "[[:alnum:]._-]+@[[:alnum:]._-]+[.][[:alpha:]]+",
             contacts$Address)]
emails_split <- strsplit(emails, "@")
emails_split
```

`emails_split` is a *list*. In this case, it seems convenient to convert the split strings into a matrix using `unlist` and `matrix` (you may want to have a quick look at Exercise 3.3 to re-familiarise yourself with `matrix`):

```
emails_split <- unlist(emails_split)
```

```
# Store in a matrix with length(emails_split)/2 rows and 2 columns:
emails_matrix <- matrix(emails_split,
                        nrow = length(emails_split)/2,
                        ncol = 2,
                        byrow = TRUE)

# Extract usernames:
emails_matrix[,1]
```

Similarly, when working with data stored in data frames, it is sometimes desirable to split a column containing strings into two columns. Some convenience functions for this are discussed in Section 5.11.3.

5.5.6 Variable names

Variable names can be very messy, particularly when they are imported from files. You can access and manipulate the variable names of a data frame using `names`:

```
names(contacts)
names(contacts)[1] <- "ID number"
grep("[aA]", names(contacts))
```

~

Exercise 5.7. Download the file `handkerchief.csv` from the book's web page. It contains a short list of prices of Italian handkerchiefs from the 1769 publication *Prices in those branches of the weaving manufactory, called, the black branch, and, the fancy branch*. Load the data in a data frame in R and then do the following:

1. Read the documentation for the function `nchar`. What does it do? Apply it to the `Italian.handkerchief` column of your data frame.
2. Use `grep` to find out how many rows of the `Italian.handkerchief` column that contain numbers.
3. Find a way to extract the prices in shillings (S) and pence (D) from the `Price` column, storing these in two new `numeric` variables in your data frame.

Exercise 5.8. Download the `oslo-biomarkers.xlsx` data from the book's web page. It contains data from a medical study about patients with disc herniations, performed at the Oslo University Hospital, Ullevål (this is a modified[6] version of the data analysed by Moen et al. (2016)). Blood samples were collected from a number of patients with disc herniations at three time points: 0 weeks (first visit at the hospital),

[6] For patient confidentiality purposes.

6 weeks and 12 months. The levels of some biomarkers related to inflammation were measured in each blood sample. The first column in the spreadsheet contains information about the patient ID and the time point of sampling. Load the data and check its structure. Each patient is uniquely identified by their ID number. How many patients were included in the study?

Exercise 5.9. What patterns do the following regular expressions describe? Apply them to the `Address` vector of the `contacts` data to check that you interpreted them correctly.

1. `"$g"`
2. `"^[^[[:digit:]]"`
3. `"a(s|l)"`
4. `"[[:lower:]]+[.][[:lower:]]+"`

Exercise 5.10. Write code that, given a string, creates a vector containing all words from the string, with one word in each element and no punctuation marks. Apply it to the following string to check that it works:

```
x <- "This is an example of a sentence, with 10 words. Here are 4 more!"
```

5.6 Working with dates and times

Data describing dates and times can be complex, not least because they can be written is so many different formats. 1 April 2020 can be written as `2020-04-01`, `20/04/01`, `200401`, `1/4 2020`, `4/1/20`, `1 Apr 20`, and a myriad of other ways. 5 past 6 in the evening can be written as `18:05` or `6.05 pm`. In addition to this ambiguity, time zones, daylight saving time, leap years and even leap seconds make working with dates and times even more complicated.

The default in R is to use the ISO8601 standards, meaning that dates are written as YYYY-MM-DD and that times are written using the 24-hour hh:mm:ss format. In order to avoid confusion, you should always use these, unless you have *very* strong reasons not to.

Dates in R are represented as `Date` objects, and dates with times as `POSIXct` objects. The examples below are concerned with `Date` objects, but you will explore `POSIXct` too, in Exercise 5.12.

5.6.1 Date formats

The `as.Date` function tries to coerce a `character` string to a date. For some formats, it will automatically succeed, whereas for others, you have to provide the format of

the date manually. To complicate things further, what formats work automatically will depend on your system settings. Consequently, the safest option is always to specify the format of your dates, to make sure that the code still will run if you at some point have to execute it on a different machine. To help describe date formats, R has a number of tokens to describe days, months and years:

- `%d` - day of the month as a number (01-31).
- `%m` - month of the year as a number (01-12).
- `%y` - year without century (00-99).
- `%Y` - year with century (e.g. 2020).

Here are some examples of date formats, all describing 1 April 2020 - try them both with and without specifying the format to see what happens:

```
as.Date("2020-04-01")
as.Date("2020-04-01", format = "%Y-%m-%d")
as.Date("4/1/20")
as.Date("4/1/20", format = "%m/%d/%y")

# Sometimes dates are expressed as the number of days since a
# certain date. For instance, 1 April 2020 is 43,920 days after
# 1 January 1900:
as.Date(43920, origin = "1900-01-01")
```

If the date includes month or weekday names, you can use tokens to describe that as well:

- `%b` - abbreviated month name, e.g. `Jan`, `Feb`.
- `%B` - full month name, e.g. `January`, `February`.
- `%a` - abbreviated weekday, e.g. `Mon`, `Tue`.
- `%A` - full weekday, e.g. `Monday`, `Tuesday`.

Things become a little more complicated now though, because R will interpret the names as if they were written in the language set in your *locale*, which contains a number of settings related your language and region. To find out what language is in your locale, you can use:

```
Sys.getlocale("LC_TIME")
```

I'm writing this on a machine with Swedish locale settings (my output from the above code chunk is `"sv_SE.UTF-8"`). The Swedish word for *Wednesday* is *onsdag*[7], and therefore the following code doesn't work on my machine:

```
as.Date("Wednesday 1 April 2020", format = "%A %d %B %Y")
```

However, if I translate it to Swedish, it runs just fine:

[7] The Swedish *onsdag* and English *Wednesday* both derive from the proto-Germanic *Wodensdag*, Odin's day, in honour of the old Germanic god of that name.

```
as.Date("Onsdag 1 april 2020", format = "%A %d %B %Y")
```

You may at times need to make similar translations of dates. One option is to use `gsub` to translate the names of months and weekdays into the correct language (see Section 5.5.4). Alternatively, you can change the locale settings. On most systems, the following setting will allow you to read English months and days properly:

```
Sys.setlocale("LC_TIME", "C")
```

The locale settings will revert to the defaults the next time you start R.

Conversely, you may want to extract a substring from a `Date` object, for instance the day of the month. This can be done using `strftime`, using the same tokens as above. Here are some examples, including one with the token `%j`, which can be used to extract the day of the year:

```
dates <- as.Date(c("2020-04-01", "2021-01-29", "2021-02-22"),
                 format = "%Y-%m-%d")

# Extract the day of the month:
strftime(dates, format = "%d")

# Extract the month:
strftime(dates, format = "%m")

# Extract the year:
strftime(dates, format = "%Y")

# Extract the day of the year:
strftime(dates, format = "%j")
```

Should you need to, you can of course convert these objects from `character` to `numeric` using `as.numeric`.

For a complete list of tokens that can be used to describe date patterns, see `?strftime`.

~

Exercise 5.11. Consider the following `Date` vector:

```
dates <- as.Date(c("2015-01-01", "1984-03-12", "2012-09-08"),
                 format = "%Y-%m-%d")
```

1. Apply the functions `weekdays`, `months` and `quarters` to the vector. What do they do?

2. Use the `julian` function to find out how many days passed between 1970-01-01 and the dates in `dates`.

Exercise 5.12. Consider the three `character` objects created below:

```
time1 <- "2020-04-01 13:20"
time2 <- "2020-04-01 14:30"
time3 <- "2020-04-03 18:58"
```

1. What happens if you convert the three variables to `Date` objects using `as.Date` without specifying the date format?
2. Convert `time1` to a `Date` object and add `1` to it. What is the result?
3. Convert `time3` and `time1` to `Date` objects and subtract them. What is the result?
4. Convert `time2` and `time1` to `Date` objects and subtract them. What is the result?
5. What happens if you convert the three variables to `POSIXct` date and time objects using `as.POSIXct` without specifying the date format?
6. Convert `time3` and `time1` to `POSIXct` objects and subtract them. What is the result?
7. Convert `time2` and `time1` to `POSIXct` objects and subtract them. What is the result?
8. Use the `difftime` to repeat the calculation in task 6, but with the result presented in hours.

Exercise 5.13. In some fields, e.g. economics, data is often aggregated on a quarter-year level, as in these examples:

```
qvec1 <- c("2020 Q4", "2021 Q1", "2021 Q2")
qvec2 <- c("Q4/20", "Q1/21", "Q2/21")
qvec3 <- c("Q4-2020", "Q1-2021", "Q2-2021")
```

To convert `qvec1` to a `Date` object, we can use `as.yearqtr` from the `zoo` package in two ways:

```
library(zoo)
as.Date(as.yearqtr(qvec1, format = "%Y Q%q"))
as.Date(as.yearqtr(qvec1, format = "%Y Q%q"), frac = 1)
```

1. Describe the results. What is the difference? Which do you think is preferable?
2. Convert `qvec2` and `qvec3` to `Date` objects in the same way. Make sure that you get the `format` argument, which describes the date format, right.

5.6.2 Plotting with dates

`ggplot2` automatically recognises `Date` objects and will usually plot them in a nice way. That only works if it actually has the dates though. Consider the following plot, which we created in Section 4.6.7 - it shows the daily electricity demand in Victoria, Australia in 2014:

```
library(plotly)
library(fpp2)

## Create the plot object
myPlot <- autoplot(elecdaily[,"Demand"])

## Create the interactive plot
ggplotly(myPlot)
```

When you hover the points, the formatting of the dates looks odd. We'd like to have proper dates instead. In order to do so, we'll use `seq.Date` to create a sequence of dates, ranging from 2014-01-01 to 2014-12-31:

```
## Create a data frame with better formatted dates
elecdaily2 <- as.data.frame(elecdaily)
elecdaily2$Date <- seq.Date(as.Date("2014-01-01"),
                            as.Date("2014-12-31"),
                            by = "day")

## Create the plot object
myPlot <- ggplot(elecdaily2, aes(Date, Demand)) +
      geom_line()

## Create the interactive plot
ggplotly(myPlot)
```

`seq.Date` can be used analogously to create sequences where there is a week, month, quarter or year between each element of the sequence, by changing the `by` argument.

~

Exercise 5.14. Return to the plot from Exercise 4.12, which was created using

```
library(fpp2)
autoplot(elecdaily, facets = TRUE)
```

You'll notice that the x-axis shows week numbers rather than dates (the dates in the `elecdaily` time series object are formatted as weeks with decimal numbers). Make a time series plot of the `Demand` variable with dates (2014-01-01 to 2014-12-31)

along the x-axis (your solution is likely to rely on standard R techniques rather than `autoplot`).

Exercise 5.15. Create an interactive version time series plot of the `a10` anti-diabetic drug sales data, as in Section 4.6.7. Make sure that the dates are correctly displayed.

5.7 Data manipulation with `data.table`, `dplyr`, and `tidyr`

In the remainder of this chapter, we will use three packages that contain functions for fast and efficient data manipulation: `data.table` and the tidyverse packages `dplyr` and `tidyr`. To begin with, it is therefore a good idea to install them. And while you wait for the installation to finish, read on.

```
install.packages(c("dplyr", "tidyr", "data.table"))
```

There is almost always more than one way to solve a problem in R. We now know how to access vectors and elements in data frames, e.g. to compute means. We also know how to modify and add variables to data frames. Indeed, you can do just about anything using the functions in base R. Sometimes, however, those solutions become rather cumbersome, as they can require a fair amount of programming and verbose code. `data.table` and the tidyverse packages offer simpler solutions and speed up the workflow for these types of problems. Both can be used for the same tasks. You can learn one of them or both. The syntax used for `data.table` is often more concise and arguably more consistent than that in `dplyr` (it is in essence an extension of the `[i, j]` notation that we have already used for data frames). Second, it is fast and memory-efficient, which makes a huge difference if you are working with big data (you'll see this for yourself in Section 6.6). On the other hand, many people prefer the syntax in `dplyr` and `tidyr`, which lends itself exceptionally well for usage with pipes. If you work with small or medium-sized datasets, the difference in performance between the two packages is negligible. `dplyr` is also much better suited for working directly with databases, which is a huge selling point if your data already is in a database[8].

In the sections below, we will see how to perform different operations using both `data.table` and the tidyverse packages. Perhaps you already know which one that you want to use (`data.table` if performance is important to you, `dplyr`+`tidyr` if you like to use pipes or will be doing a lot of work with databases). If not, you can use these examples to guide your choice. Or not choose at all! I regularly use both packages myself, to harness the strength of both. There is no harm in knowing how to use both a hammer and a screwdriver.

[8] There is also a package called `dtplyr`, which allows you to use the fast functions from `data.table` with `dplyr` syntax. It is useful if you are working with big data, already know `dplyr` and don't want to learn `data.table`. If that isn't an accurate description of you, you can safely ignore `dtplyr` for now.

5.7.1 data.table and tidyverse syntax basics

`data.table` relies heavily on the `[i, j]` notation that is used for data frames in R. It also adds a third element: `[i, j, by]`. Using this, R selects the rows indicated by `i`, the columns indicated by `j` and groups them by `by`. This makes it easy e.g. to compute grouped summaries.

With the tidyverse packages you will instead use new functions with names like `filter` and `summarise` to perform operations on your data. These are typically combined using the pipe operator, `%>%`, which makes the code flow nicely from left to right.

It's almost time to look at some examples of what this actually looks like in practice. First though, now that you've installed `data.table` and `dplyr`, it's time to load them (we'll get to `tidyr` a little later). We'll also create a `data.table` version of the `airquality` data, which we'll use in the examples below. This is required in order to use `data.table` syntax, as it only works on `data.table` objects. Luckily, `dplyr` works perfectly when used on `data.table` objects, so we can use the same object for the examples for both packages.

```
library(data.table)
library(dplyr)

aq <- as.data.table(airquality)
```

When importing data from csv files, you can import them as `data.table` objects instead of `data.frame` objects by replacing `read.csv` with `fread` from the `data.table` package. The latter function also has the benefit of being substantially faster when importing large (several MB's) csv files.

Note that, similar to what we saw in Section 5.2.1, variables in imported data frames can have names that would not be allowed in base R, for instance including forbidden characters like `-`. `data.table` and `dplyr` allow you to work with such variables by wrapping their names in apostrophes: referring to the illegally named variable as `illegal-character-name` won't work, but `` `illegal-character-name` `` will.

5.7.2 Modifying a variable

As a first example, let's consider how to use `data.table` and `dplyr` to modify a variable in a data frame. The wind speed in `airquality` is measured in miles per hour. We can convert that to metres per second by multiplying the speed by 0.44704. Using only base R, we'd do this using `airquality$Wind <- airquality$Wind * 0.44704`. With `data.table` we can instead do this using `[i, j]` notation, and with `dplyr` we can do it by using a function called `mutate` (because it "mutates" your data).

Change wind speed to m/s instead of mph:

With `data.table`:

```
aq[, Wind := Wind * 0.44704]
```

With `dplyr`:

```
aq %>% mutate(Wind =
         Wind * 0.44704) -> aq
```

Note that when using `data.table`, there is not an explicit assignment. We don't use `<-` to assign the new data frame to `aq` - instead the assignment happens automatically. This means that you have to be a little bit careful, so that you don't inadvertently make changes to your data when playing around with it.

In this case, using `data.table` or `dplyr` doesn't make anything easier. Where these packages really shine is when we attempt more complicated operations. Before that though, let's look at a few more simple examples.

5.7.3 Computing a new variable based on existing variables

What if we wish to create new variables based on other variables in the data frame? For instance, maybe we want to create a *dummy variable* called `Hot`, containing a `logical` that describes whether a day was hot (temperature above 90 - `TRUE`) or not (`FALSE`). That is, we wish to check the condition `Temp > 90` for each row, and put the resulting `logical` in the new variable `Hot`.

Add a dummy variable describing whether it is hotter than 90:

With `data.table`:

```
aq[, Hot := Temp > 90]
```

With `dplyr`:

```
aq %>% mutate(Hot = Temp > 90) -> aq
```

5.7.4 Renaming a variable

To change the name of a variable, we can use `setnames` from `data.table` or `rename` from `dplyr`. Let's change the name of the variable `Hot` that we created in the previous section, to `HotDay`:

With `data.table`:

```
setnames(aq, "Hot", "HotDay")
```

With `dplyr`:

```
aq %>% rename(HotDay = Hot) -> aq
```

5.7.5 Removing a variable

Maybe adding `Hot` to the data frame wasn't such a great idea after all. How can we remove it?

Removing `Hot`:

With `data.table`:

```
aq[, Hot := NULL]
```

With `dplyr`:

```
aq %>% select(-Hot) -> aq
```

If we wish to remove multiple columns at once, the syntax is similar:

Removing multiple columns:

With `data.table`:

```
aq[, c("Month", "Day") := NULL]
```

With `dplyr`:

```
aq %>% select(-Month, -Day) -> aq
```

~

Exercise 5.16. Load the VAS pain data `vas.csv` from Exercise 3.8. Then do the following:

1. Remove the columns `X` and `X.1`.
2. Add a dummy variable called `highVAS` that indicates whether a patient's `VAS` is 7 or greater on any given day.

5.7.6 Recoding `factor` levels

Changing the names of `factor` levels in base R typically relies on using indices of level names, as in Section 5.4.2. This can be avoided using `data.table` or the `recode` function in `dplyr`. We return to the `smoke` example from Section 5.4 and put it in a `data.table`:

```
library(data.table)
library(dplyr)

smoke <- c("Never", "Never", "Heavy", "Never", "Occasionally",
           "Never", "Never", "Regularly", "Regularly", "No")

smoke2 <- factor(smoke, levels = c("Never", "Occasionally",
                                   "Regularly", "Heavy"),
                        ordered = TRUE)

smoke3 <- data.table(smoke2)
```

Suppose that we want to change the levels' names to abbreviated versions: *Nvr*, *Occ*, *Reg* and *Hvy*. Here's how to do this:

With `data.table`:

```
new_names = c("Nvr", "Occ", "Reg", "Hvy")
smoke3[.(smoke2 = levels(smoke2), to = new_names),
         on = "smoke2",
         smoke2 := i.to]
smoke3[, smoke2 := droplevels(smoke2)]
```

With `dplyr`:

```
smoke3 %>% mutate(smoke2 = recode(smoke2,
                  "Never" = "Nvr",
                  "Occasionally" = "Occ",
                  "Regularly" = "Reg",
                  "Heavy" = "Hvy"))
```

Next, we can combine the *Occ*, *Reg* and *Hvy* levels into a single level, called *Yes*:

With `data.table`:

```
smoke3[.(smoke2 = c("Occ", "Reg", "Hvy"), to = "Yes"),
       on = "smoke2",
       smoke2 := i.to]
```

With `dplyr`:

```
smoke3 %>% mutate(smoke2 = recode(smoke2,
                  "Occ" = "Yes",
                  "Reg" = "Yes",
                  "Hvy" = "Yes"))
```

~

Exercise 5.17. In Exercise 3.7 you learned how to create a `factor` variable from a `numeric` variable using `cut`. Return to your solution (or the solution at the back of the book) and do the following using `data.table` and/or `dplyr`:

1. Change the category names to `Mild`, `Moderate` and `Hot`.
2. Combine `Moderate` and `Hot` into a single level named `Hot`.

5.7.7 Grouped summaries

We've already seen how we can use `aggregate` and `by` to create grouped summaries. However, in many cases it is as easy or easier to use `data.table` or `dplyr` for such summaries.

To begin with, let's load the packages again (in case you don't already have them loaded), and let's recreate the `aq` `data.table`, which we made a bit of a mess of by removing some important columns in the previous section:

```
library(data.table)
library(dplyr)

aq <- data.table(airquality)
```

Now, let's compute the mean temperature for each month. Both `data.table` and `dplyr` will return a data frame with the results. In the `data.table` approach, assigning a name to the summary statistic (`mean`, in this case) is optional, but not in `dplyr`.

With `data.table`:

```
aq[, mean(Temp), Month]
# or, to assign a name:
aq[, .(meanTemp = mean(Temp)),
   Month]
```

With `dplyr`:

```
aq %>% group_by(Month) %>%
       summarise(meanTemp =
                   mean(Temp))
```

You'll recall that if we apply `mean` to a vector containing `NA` values, it will return `NA`:

With `data.table`:

```
aq[, mean(Ozone), Month]
```

With `dplyr`:

```
aq %>% group_by(Month) %>%
       summarise(meanTemp =
                   mean(Ozone))
```

In order to avoid this, we can pass the argument `na.rm = TRUE` to `mean`, just as we would in other contexts. To compute the mean ozone concentration for each month, ignoring `NA` values:

With `data.table`:

```
aq[, mean(Ozone, na.rm = TRUE),
   Month]
```

With `dplyr`:

```
aq %>% group_by(Month) %>%
       summarise(meanTemp =
                   mean(Ozone,
                    na.rm = TRUE))
```

What if we want to compute a grouped summary statistic involving two variables? For instance, the correlation between temperature and wind speed for each month?

With `data.table`:

```
aq[, cor(Temp, Wind), Month]
```

With `dplyr`:

```
aq %>% group_by(Month) %>%
       summarise(cor =
                   cor(Temp, Wind))
```

The syntax for computing multiple grouped statistics is similar. We compute both the mean temperature and the correlation for each month:

With `data.table`:

```
aq[, .(meanTemp = mean(Temp),
   cor = cor(Temp, Wind)),
   Month]
```

With `dplyr`:

```
aq %>% group_by(Month) %>%
       summarise(meanTemp =
                   mean(Temp),
                 cor =
                   cor(Temp, Wind))
```

At times, you'll want to compute summaries for all variables that share some property. As an example, you may want to compute the mean of all `numeric` variables in your data frame. In `dplyr` there is a convenience function called `across` that can be used for this: `summarise(across(where(is.numeric), mean))` will compute the mean of all numeric variables. In `data.table`, we can instead utilise the `apply` family of functions from base R, that we'll study in Section 6.5. To compute the mean of all `numeric` variables:

With `data.table`:

```
aq[, lapply(.SD, mean),
   Month,
   .SDcols = names(aq)[
     sapply(aq, is.numeric)]]
```

With `dplyr`:

```
aq %>% group_by(Month) %>%
       summarise(across(
         where(is.numeric),
         mean, na.rm = TRUE))
```

Both packages have special functions for counting the number of observations in groups: `.N` for `data.table` and `n` for `dplyr`. For instance, we can count the number of days in each month:

With `data.table`:

```
aq[, .N, Month]
```

With `dplyr`:

```
aq %>% group_by(Month) %>%
       summarise(days = n())
```

Similarly, you can count the number of unique values of variables using `uniqueN` for `data.table` and `n_distinct` for `dplyr`:

With `data.table`:

```
aq[, uniqueN(Month)]
```

With `dplyr`:

```
aq %>% summarise(months =
              n_distinct(Month))
```

~

Exercise 5.18. Load the VAS pain data `vas.csv` from Exercise 3.8. Then do the following using `data.table` and/or `dplyr`:

1. Compute the mean VAS for each patient.
2. Compute the lowest and highest VAS recorded for each patient.
3. Compute the number of high-VAS days, defined as days with where the VAS was at least 7, for each patient.

Exercise 5.19. We return to the `datasauRus` package and the `datasaurus_dozen` dataset from Exercise 3.13. Check its structure and then do the following using `data.table` and/or `dplyr`:

1. Compute the mean of `x`, mean of `y`, standard deviation of `x`, standard deviation of `y`, and correlation between `x` and `y`, grouped by `dataset`. Are there any differences between the 12 datasets?
2. Make a scatterplot of `x` against `y` for each dataset. Are there any differences between the 12 datasets?

5.7.8 Filling in missing values

In some cases, you may want to fill missing values of a variable with the previous non-missing entry. To see an example of this, let's create a version of `aq` where the value of `Month` are missing for some days:

```
aq$Month[c(2:3, 36:39, 70)] <- NA

# Some values of Month are now missing:
head(aq)
```

To fill the missing values with the last non-missing entry, we can now use `nafill` or `fill` as follows:

With `data.table`:

```
aq[, Month := nafill(
      Month, "locf")]
```

With `tidyr`:

```
aq %>% fill(Month) -> aq
```

To instead fill the missing values with the *next* non-missing entry:

With `data.table`:

```
aq[, Month := nafill(
      Month, "nocb")]
```

With `tidyr`:

```
aq %>% fill(Month,
        .direction = "up") -> aq
```

~

Exercise 5.20. Load the VAS pain data `vas.csv` from Exercise 3.8. Fill the missing values in the `Visit` column with the last non-missing value.

5.7.9 Chaining commands together

When working with tidyverse packages, commands are usually chained together using `%>%` pipes. When using `data.table`, commands are chained by repeated use of `[]` brackets on the same line. This is probably best illustrated using an example. Assume again that there are missing values in `Month` in `aq`:

```
aq$Month[c(2:3, 36:39, 70)] <- NA
```

To fill in the missing values with the last non-missing entry (Section 5.7.8) and then count the number of days in each month (Section 5.7.7), we can do as follows.

With `data.table`:

```
aq[, Month := nafill(Month, "locf")][, .N, Month]
```

With `tidyr` and `dplyr`:

```
aq %>% fill(Month) %>%
       group_by(Month) %>%
       summarise(days = n())
```

5.8 Filtering: select rows

You'll frequently want to filter away some rows from your data. Perhaps you only want to select rows where a variable exceeds some value, or want to exclude rows with `NA` values. This can be done in several different ways: using row numbers, using conditions, at random, or using regular expressions. Let's have a look at them, one by one. We'll use `aq`, the `data.table` version of `airquality` that we created before, for the examples.

```
library(data.table)
library(dplyr)

aq <- data.table(airquality)
```

5.8.1 Filtering using row numbers

If you know the row numbers of the rows that you wish to remove (perhaps you've found them using `which`, as in Section 3.2.3?), you can use those numbers for filtering. Here are four examples.

To select the third row:

With `data.table`:

```
aq[3,]
```

With `dplyr`:

```
aq %>% slice(3)
```

To select rows 3 to 5:

With `data.table`:

```
aq[3:5,]
```

With `dplyr`:

```
aq %>% slice(3:5)
```

To select rows 3, 7 and 15:

With `data.table`:

```
aq[c(3, 7, 15),]
```

With `dplyr`:

```
aq %>% slice(c(3, 7, 15))
```

To select all rows except rows 3, 7 and 15:

With `data.table`:

```
aq[-c(3, 7, 15),]
```

With `dplyr`:

```
aq %>% slice(-c(3, 7, 15))
```

5.8.2 Filtering using conditions

Filtering is often done using conditions, e.g. to select observations with certain properties. Here are some examples:

To select rows where `Temp` is greater than 90:

With `data.table`:

```
aq[Temp > 90,]
```

With `dplyr`:

```
aq %>% filter(Temp > 90)
```

To select rows where `Month` is 6 (June):

With `data.table`:

```
aq[Month == 6,]
```

With `dplyr`:

```
aq %>% filter(Month == 6)
```

To select rows where `Temp` is greater than 90 and `Month` is 6 (June):

With `data.table`:

```
aq[Temp > 90 & Month == 6,]
```

With `dplyr`:

```
aq %>% filter(Temp > 90,
              Month == 6)
```

To select rows where `Temp` is between 80 and 90 (including 80 and 90):

With `data.table`:

```
aq[Temp %between% c(80, 90),]
```

With `dplyr`:

```
aq %>% filter(between(Temp,
                      80, 90))
```

To select the 5 rows with the highest `Temp`:

With `data.table`:

```
aq[frankv(-Temp,
      ties.method = "min") <= 5,
   ]
```

With `dplyr`:

```
aq %>% top_n(5, Temp)
```

In this case, the above code returns more than 5 rows because of ties.

To remove duplicate rows:

With `data.table`:

```
unique(aq)
```

With `dplyr`:

```
aq %>% distinct
```

To remove rows with missing data (`NA` values) in at least one variable:

With `data.table`:

```
na.omit(aq)
```

With `tidyr`:

```
library(tidyr)
aq %>% drop_na
```

To remove rows with missing `Ozone` values:

With `data.table`:

```
na.omit(aq, "Ozone")
```

With `tidyr`:

```
library(tidyr)
aq %>% drop_na("Ozone")
```

At times, you want to filter your data based on whether the observations are connected to observations in a different dataset. Such filters are known as semijoins and antijoins, and are discussed in Section 5.12.4.

5.8.3 Selecting rows at random

In some situations, for instance when training and evaluating machine learning models, you may wish to draw a random sample from your data. This is done using the `sample` (`data.table`) and `sample_n` (`dplyr`) functions.

To select 5 rows at random:

With `data.table`:

```
aq[sample(.N, 5),]
```

With `dplyr`:

```
aq %>% sample_n(5)
```

If you run the code multiple times, you will get different results each time. See Section 7.1 for more on random sampling and how it can be used.

5.8.4 Using regular expressions to select rows

In some cases, particularly when working with text data, you'll want to filter using regular expressions (see Section 5.5.3). `data.table` has a convenience function called `%like%` that can be used to call `grepl` in an alternative (less opaque?) way. With `dplyr` we use `grepl` in the usual fashion. To have some text data to try this out on, we'll use this data frame, which contains descriptions of some dogs:

```
dogs <- data.table(Name = c("Bianca", "Bella", "Mimmi", "Daisy",
                            "Ernst", "Smulan"),
                   Breed = c("Greyhound", "Greyhound", "Pug", "Poodle",
                             "Bedlington Terrier", "Boxer"),
                   Desc = c("Fast, playful", "Fast, easily worried",
```

```
                            "Intense, small, loud",
                            "Majestic, protective, playful",
                            "Playful, relaxed",
                            "Loving, cuddly, playful"))
View(dogs)
```

To select all rows with names beginning with B:

With `data.table`:

```
dogs[Name %like% "^B",]
# or:
dogs[grepl("^B", Name),]
```

With `dplyr`:

```
dogs %>% filter(grepl("B[a-z]",
                      Name))
```

To select all rows where `Desc` includes the word `playful`:

With `data.table`:

```
dogs[Desc %like% "[pP]layful",]
```

With `dplyr`:

```
dogs %>% filter(grepl("[pP]layful",
                      Desc))
```

~

Exercise 5.21. Download the `ucdp-onesided-191.csv` data file from the book's web page. It contains data about international attacks on civilians by governments and formally organised armed groups during the period 1989-2018, collected as part of the Uppsala Conflict Data Program (Eck & Hultman, 2007; Petterson et al., 2019). Among other things, it contains information about the actor (attacker), the fatality rate, and attack location. Load the data and check its structure.

1. Filter the rows so that only conflicts that took place in Colombia are retained. How many different actors were responsible for attacks in Colombia during the period?
2. Using the `best_fatality_estimate` column to estimate fatalities, calculate the number of worldwide fatalities caused by government attacks on civilians during 1989-2018.

Exercise 5.22. Load the `oslo-biomarkers.xlsx` data from Exercise 5.8. Use `data.table` and/or `dplyr` to do the following:

1. Select only the measurements from blood samples taken at 12 months.
2. Select only the measurements from the patient with ID number 6.

5.9 Subsetting: select columns

Another common situation is that you want to remove some variables from your data. Perhaps the variables aren't of interest in a particular analysis that you're going to perform, or perhaps you've simply imported more variables than you need. As with rows, this can be done using numbers, names or regular expressions. Let's look at some examples using the `aq` data:

```
library(data.table)
library(dplyr)

aq <- data.table(airquality)
```

5.9.1 Selecting a single column

When selecting a single column from a data frame, you sometimes want to extract the column as a vector and sometimes as a single-column data frame (for instance if you are going to pass it to a function that takes a data frame as input). You should be a little bit careful when doing this, to make sure that you get the column in the correct format:

With `data.table`:

```
# Return a vector:
aq$Temp
# or
aq[, Temp]

# Return a data.table:
aq[, "Temp"]
```

With `dplyr`:

```
# Return a vector:
aq$Temp
# or
aq %>% pull(Temp)

# Return a tibble:
aq %>% select(Temp)
```

5.9.2 Selecting multiple columns

Selecting multiple columns is more straightforward, as the object that is returned always will be a data frame. Here are some examples.

To select `Temp`, `Month` and `Day`:

With `data.table`:

```
aq[, .(Temp, Month, Day)]
```

With `dplyr`:

```
aq %>% select(Temp, Month, Day)
```

To select all columns between `Wind` and `Month`:

With `data.table`:

```
aq[, Wind:Month]
```

With `dplyr`:

```
aq %>% select(Wind:Month)
```

To select all columns except `Month` and `Day`:

With `data.table`:

```
aq[, -c("Month", "Day")]
```

With `dplyr`:

```
aq %>% select(-Month, -Day)
```

To select all numeric variables (which for the `aq` data is all variables!):

With `data.table`:

```
aq[, sapply(msleep, class) ==
                    "numeric"]
```

With `dplyr`:

```
aq %>% select_if(is.numeric)
```

To remove columns with missing (`NA`) values:

With `data.table`:

```
aq[, .SD,
   .SDcols = colSums(
     is.na(aq)) == 0]
```

With `dplyr`:

```
aq %>% select_if(~all(!is.na(.)))
```

5.9.3 Using regular expressions to select columns

In `data.table`, using regular expressions to select columns is done using `grep`. `dplyr` differs in that it has several convenience functions for selecting columns, like `starts_with`, `ends_with`, `contains`. As an example, we can select variables the name of which contains the letter `n`:

With `data.table`:

```
vars <- grepl("n", names(aq))
aq[, ..vars]
```

With `dplyr`:

```
# contains is a convenience
# function for checking if a name
# contains a string:
aq %>% select(contains("n"))
# matches can be used with any
# regular expression:
aq %>% select(matches("n"))
```

5.9.4 Subsetting using column numbers

It is also possible to subsetting using column numbers, but you need to be careful if you want to use that approach. Column numbers can change, for instance if a variable is removed from the data frame. More importantly, however, using column numbers can yield different results depending on what type of data table you're using. Let's have a look at what happens if we use this approach with different types of data tables:

```
# data.frame:
aq <- as.data.frame(airquality)
str(aq[,2])

# data.table:
aq <- as.data.table(airquality)
str(aq[,2])

# tibble:
aq <- as_tibble(airquality)
str(aq[,2])
```

As you can see, `aq[, 2]` returns a vector, a data table or a tibble, depending on what type of object `aq` is. Unfortunately, this approach is used by several R packages, and can cause problems, because it may return the wrong type of object.

A better approach is to use `aq[[2]]`, which works the same for data frames, data tables and tibbles, returning a vector:

```
# data.frame:
aq <- as.data.frame(airquality)
str(aq[[2]])

# data.table:
aq <- as.data.table(airquality)
str(aq[[2]])

# tibble:
aq <- as_tibble(airquality)
str(aq[[2]])
```

~

Exercise 5.23. Return to the `ucdp-onesided-191.csv` data from Exercise 5.21. To have a cleaner and less bloated dataset to work with, it can make sense to remove some columns. Select only the `actor_name`, `year`, `best_fatality_estimate` and `location` columns.

5.10 Sorting

Sometimes you don't want to filter rows, but rearrange their order according to their values for some variable. Similarly, you may want to change the order of the columns in your data. I often do this after merging data from different tables (as we'll do in Section 5.12). This is often useful for presentation purposes, but can at times also aid in analyses.

5.10.1 Changing the column order

It is straightforward to change column positions using `setcolorder` in `data.table` and `relocate` in `dplyr`.

To put `Month` and `Day` in the first two columns, without rearranging the other columns:

With `data.table`:

```
setcolorder(aq, c("Month", "Day"))
```

With `dplyr`:

```
aq %>% relocate("Month", "Day")
```

5.10.2 Changing the row order

In `data.table`, `order` is used for sorting rows, and in `dplyr`, `arrange` is used (sometimes in combination with `desc`). The syntax differs depending on whether you wish to sort your rows in ascending or descending order. We will illustrate this using the `airquality` data.

```
library(data.table)
library(dplyr)

aq <- data.table(airquality)
```

First of all, if you're just looking to sort a single vector, rather than an entire data frame, the quickest way to do so is to use `sort`:

```
sort(aq$Wind)
sort(aq$Wind, decreasing = TRUE)
sort(c("C", "B", "A", "D"))
```

If you're looking to sort an entire data frame by one or more variables, you need to move beyond `sort`. To sort rows by `Wind` (*ascending* order):

With `data.table`:

```
aq[order(Wind),]
```

With `dplyr`:

```
aq %>% arrange(Wind)
```

To sort rows by `Wind` (*descending* order):

With `data.table`:

```
aq[order(-Wind),]
```

With `dplyr`:

```
aq %>% arrange(-Wind)
# or
aq %>% arrange(desc(Wind))
```

To sort rows, first by `Temp` (ascending order) and then by `Wind` (descending order):

With `data.table`:

```
aq[order(Temp, -Wind),]
```

With `dplyr`:

```
aq %>% arrange(Temp, desc(Wind))
```

~

Exercise 5.24. Load the `oslo-biomarkers.xlsx` data from Exercise 5.8. Note that it is not ordered in a natural way. Reorder it by patient ID instead.

5.11 Reshaping data

The `gapminder` dataset from the `gapminder` package contains information about life expectancy, population size and GDP per capita for 142 countries for 12 years from the period 1952-2007. To begin with, let's have a look at the data[9]:

```
library(gapminder)
?gapminder
View(gapminder)
```

Each row contains data for one country and one year, meaning that the data for each country is spread over 12 rows. This is known as *long data* or *long format*. As another option, we could store it in *wide format*, where the data is formatted so that all observations corresponding to a country are stored on the same row:

```
Country        Continent   lifeExp1952 lifeExp1957 lifeExp1962 ...
Afghanistan    Asia         28.8        30.2        32.0       ...
Albania        Europe       55.2        59.3        64.8       ...
```

Sometimes it makes sense to spread an observation over multiple rows (long format), and sometimes it makes more sense to spread a variable across multiple columns (wide format). Some analyses require long data, whereas others require wide data.

[9] You may need to install the package first, using `install.packages("gapminder")`.

And if you're unlucky, data will arrive in the wrong format for the analysis you need to do. In this section, you'll learn how to transform your data from long to wide, and back again.

5.11.1 From long to wide

When going from a long format to a wide format, you choose columns to group the observations by (in the `gapminder` case: `country` and maybe also `continent`), columns to take values names from (`lifeExp`, `pop` and `gdpPercap`), and columns to create variable names from (`year`).

In `data.table`, the transformation from long to wide is done using the `dcast` function. `dplyr` does not contain functions for such transformations, but its sibling, the tidyverse package `tidyr`, does.

The `tidyr` function used for long-to-wide formatting is `pivot_wider`. First, we convert the `gapminder` data frame to a `data.table` object:

```
library(data.table)
library(tidyr)

gm <- as.data.table(gapminder)
```

To transform the `gm` data from long to wide and store it as `gmw`:

With `data.table`:

```
gmw <- dcast(gm, country + continent ~ year,
             value.var = c("pop", "lifeExp", "gdpPercap"))
```

With `tidyr`:

```
gm %>% pivot_wider(id_cols = c(country, continent),
                   names_from = year,
                   values_from =
                    c(pop, lifeExp, gdpPercap)) -> gmw
```

5.11.2 From wide to long

We've now seen how to transform the long format `gapminder` data to the wide format `gmw` data. But what if we want to go from wide format to long? Let's see if we can transform `gmw` back to the long format.

In `data.table`, wide-to-long formatting is done using `melt`, and in `dplyr` it is done using `pivot_longer`.

To transform the `gmw` data from long to wide:

With `data.table`:

```
gm <- melt(gmw, id.vars = c("country", "continent"),
           measure.vars = 2:37)
```

With `tidyr`:

```
gmw %>% pivot_longer(names(gmw)[2:37],
                     names_to = "variable",
                     values_to = "value") -> gm
```

The resulting data frames are perhaps *too* long, with each variable (`pop`, `lifeExp` and `gdpPercapita`) being put on a different row. To make it look like the original dataset, we must first split the `variable` variable (into a column with variable names and column with years) and then make the data frame a little wider again. That is the topic of the next section.

5.11.3 Splitting columns

In the too long `gm` data that you created at the end of the last section, the observations in the `variable` column look like `pop_1952` and `gdpPercap_2007`, i.e. are of the form `variableName_year`. We'd like to split them into two columns: one with variable names and one with years. `dplyr` has a function called `tstrsplit` for this purpose, and `tidyr` has `separate`.

To split the `variable` column at the underscore `_`, and then reformat `gm` to look like the original `gapminder` data:

With `data.table`:

```
gm[, c("variable", "year") := tstrsplit(variable,
                                        "_", fixed = TRUE)]
gm <- dcast(gm, country + year ~  variable,
            value.var = c("value"))
```

With `tidyr`:

```
gm %>% separate(variable,
                into = c("variable", "year"),
                sep = "_") %>%
       pivot_wider(id_cols = c(country, continent, year),
                   names_from = variable,
                   values_from = value) -> gm
```

5.11.4 Merging columns

Similarly, you may at times want to merge two columns, for instance if one contains the day+month part of a date and the other contains the year. An example of such a situation can be found in the `airquality` dataset, where we may want to merge the

Day and Month columns into a new Date column. Let's re-create the aq data.table object one last time:

```
library(data.table)
library(tidyr)

aq <- as.data.table(airquality)
```

If we wanted to create a Date column containing the year (1973), month and day for each observation, we could use paste and as.Date:

```
as.Date(paste(1973, aq$Month, aq$Day, sep = "-"))
```

The natural data.table approach is just this, whereas tidyr offers a function called unite to merge columns, which can be combined with mutate to paste the year to the date. To merge the Month and Day columns with a year and convert it to a Date object:

With data.table:

```
aq[, Date := as.Date(paste(1973,
                           aq$Month,
                           aq$Day,
                           sep = "-"))]
```

With tidyr and dplyr:

```
aq %>% unite("Date", Month, Day,
             sep = "-") %>%
       mutate(Date = as.Date(
                paste(1973,
                Date,
                sep = "-")))
```

~

Exercise 5.25. Load the oslo-biomarkers.xlsx data from Exercise 5.8. Then do the following using data.table and/or dplyr/tidyr:

1. Split the PatientID.timepoint column in two parts: one with the patient ID and one with the timepoint.
2. Sort the table by patient ID, in numeric order.
3. Reformat the data from long to wide, keeping only the IL-8 and VEGF-A measurements.

Save the resulting data frame - you will need it again in Exercise 5.26!

5.12 Merging data from multiple tables

It is common that data is spread over multiple tables: different sheets in Excel files, different .csv files, or different tables in databases. Consequently, it is important to

be able to merge data from different tables.

As a first example, let's study the sales datasets available from the books web page: `sales-rev.csv` and `sales-weather.csv`. The first dataset describes the daily revenue for a business in the first quarter of 2020, and the second describes the weather in the region (somewhere in Sweden) during the same period[10]. Store their respective paths as `file_path1` and `file_path2` and then load them:

```
rev_data <- read.csv(file_path1, sep = ";")
weather_data <- read.csv(file_path2, sep = ";")

str(rev_data)
View(rev_data)

str(weather_data)
View(weather_data)
```

5.12.1 Binds

The simplest types of merges are *binds*, which can be used when you have two tables where either the rows or the columns *match each other exactly*. To illustrate what this may look like, we will use `data.table`/`dplyr` to create subsets of the business revenue data. First, we format the tables as `data.table` objects and the `DATE` columns as `Date` objects:

```
library(data.table)
library(dplyr)

rev_data <- as.data.table(rev_data)
rev_data$DATE <- as.Date(rev_data$DATE)

weather_data <- as.data.table(weather_data)
weather_data$DATE <- as.Date(weather_data$DATE)
```

Next, we wish to subtract three subsets: the revenue in January (`rev_jan`), the revenue in February (`rev_feb`) and the weather in January (`weather_jan`).

With `data.table`:

```
rev_jan <- rev_data[DATE %between%
                    c("2020-01-01",
                    "2020-01-31"),]
rev_feb <- rev_data[DATE %between%
                    c("2020-02-01",
                    "2020-02-29"),]
weather_jan <- weather_data[DATE
                    %between%
                    c("2020-01-01",
                    "2020-01-31"),]
```

[10] I've intentionally left out the details regarding the business - these are real sales data from a client, which can be sensitive information.

With dplyr:

```
rev_data %>% filter(between(DATE,
            as.Date("2020-01-01"),
```

```
            as.Date("2020-01-31"))
                    ) -> rev_jan
rev_data %>% filter(between(DATE,
            as.Date("2020-02-01"),
            as.Date("2020-02-29"))
                    ) -> rev_feb
weather_data %>% filter(between(
            DATE,
            as.Date("2020-01-01"),
            as.Date("2020-01-31"))
                 ) -> weather_jan
```

A quick look at the structure of the data reveals some similarities:

```
str(rev_jan)
str(rev_feb)
str(weather_jan)
```

The rows in `rev_jan` correspond one-to-one to the *rows* in `weather_jan`, with both tables being sorted in exactly the same way. We could therefore *bind their columns*, i.e. add the columns of `weather_jan` to `rev_jan`.

`rev_jan` and `rev_feb` contain the same *columns*. We could therefore *bind their rows*, i.e. add the rows of `rev_feb` to `rev_jan`. To perform these operations, we can use either base R or `dplyr`:

With base R:

```
# Join columns of datasets that
# have the same rows:
cbind(rev_jan, weather_jan)

# Join rows of datasets that have
# the same columns:
rbind(rev_jan, rev_feb)
```

With `dplyr`:

```
# Join columns of datasets that
# have the same rows:
bind_cols(rev_jan, weather_jan)

# Join rows of datasets that have
# the same columns:
bind_rows(rev_jan, rev_feb)
```

5.12.2 Merging tables using keys

A closer look at the business revenue data reveals that `rev_data` contains observations from 90 days whereas `weather_data` only contains data for 87 days; revenue data for 2020-03-01 is missing, and weather data for 2020-02-05, 2020-02-06, 2020-03-10, and 2020-03-29 are missing.

Suppose that we want to study how weather affects the revenue of the business. In order to do so, we must merge the two tables. We cannot use a simple column bind,

because the two tables have different numbers of rows. If we attempt a bind, R will produce a merged table by recycling the first few rows from `rev_data` - note that the two `DATE` columns aren't properly aligned:

```
tail(cbind(rev_data, weather_data))
```

Clearly, this is not the desired output! We need a way to connect the rows in `rev_data` with the right rows in `weather_data`. Put differently, we need something that allows us to connect the observations in one table to those in another. Variables used to connect tables are known as *keys*, and must in some way uniquely identify observations. In this case the `DATE` column gives us the key - each observation is uniquely determined by it's `DATE`. So to combine the two tables, we can combine rows from `rev_data` with the rows from `weather_data` that have the same `DATE` values. In the following sections, we'll look at different ways of merging tables using `data.table` and `dplyr`.

But first, a word of warning: finding the right keys for merging tables is not always straightforward. For a more complex example, consider the `nycflights13` package, which contains five separate but connected datasets:

```
library(nycflights13)
?airlines  # Names and carrier codes of airlines.
?airports  # Information about airports.
?flights   # Departure and arrival times and delay information for
           # flights.
?planes    # Information about planes.
?weather   # Hourly meteorological data for airports.
```

Perhaps you want to include weather information with the flight data, to study how weather affects delays. Or perhaps you wish to include information about the longitude and latitude of airports (from `airports`) in the `weather` dataset. In `airports`, each observation can be uniquely identified in three different ways: either by its airport code `faa`, its name `name` or its latitude and longitude, `lat` and `lon`:

```
?airports
head(airports)
```

If we want to use either of these options as a key when merging with `airports` data with another table, that table should also contain the same key.

The `weather` data requires no less than four variables to identify each observation: `origin`, `month`, `day` and `hour`:

```
?weather
head(weather)
```

It is not perfectly clear from the documentation, but the `origin` variable is actually the FAA airport code of the airport corresponding to the weather measurements. If

we wish to add longitude and latitude to the weather data, we could therefore use `faa` from `airports` as a key.

5.12.3 Inner and outer joins

An operation that combines columns from two tables is called a *join*. There are two main types of joins: *inner joins* and *outer joins*.

- *Inner joins*: create a table containing all observations for which the key appeared in both tables. So if we perform an inner join on the `rev_data` and `weather_data` tables using `DATE` as the key, it won't contain data for the days that are missing from either the revenue table or the weather table.

In contrast, outer joins create a table retaining rows, even if there is no match in the other table. There are three types of outer joins:

- *Left join*: retains all rows from the first table. In the revenue example, this means all dates present in `rev_data`.
- *Right join*: retains all rows from the second table. In the revenue example, this means all dates present in `weather_data`.
- *Full join*: retains all rows present in at least one of the tables. In the revenue example, this means all dates present in at least one of `rev_data` and `weather_data`.

We will use the `rev_data` and `weather_data` datasets to exemplify the different types of joins. To begin with, we convert them to `data.table` objects (which is optional if you wish to use `dplyr`):

```
library(data.table)
library(dplyr)

rev_data <- as.data.table(rev_data)
weather_data <- as.data.table(weather_data)
```

Remember that revenue data for 2020-03-01 is missing, and weather data for 2020-02-05, 2020-02-06, 2020-03-10, and 2020-03-29 are missing. This means that out of the 91 days in the period, only 86 have complete data. If we perform an inner join, the resulting table should therefore have 86 rows.

To perform and inner join of `rev_data` and `weather_data` using `DATE` as key:

With `data.table`:

```
merge(rev_data, weather_data,
      by = "DATE")

# Or:
setkey(rev_data, DATE)
rev_data[weather_data, nomatch = 0]
```

With `dplyr`:

```
rev_data %>% inner_join(
                weather_data,
                by = "DATE")
```

A left join will retain the 90 dates present in `rev_data`. To perform a(n outer) left join of `rev_data` and `weather_data` using `DATE` as key:

With `data.table`:

```
merge(rev_data, weather_data,
      all.x = TRUE, by = "DATE")

# Or:
setkey(weather_data, DATE)
weather_data[rev_data]
```

With `dplyr`:

```
rev_data %>% left_join(
                weather_data,
                by = "DATE")
```

A right join will retain the 87 dates present in `weather_data`. To perform a(n outer) right join of `rev_data` and `weather_data` using `DATE` as key:

With `data.table`:

```
merge(rev_data, weather_data,
      all.y = TRUE, by = "DATE")

# Or:
setkey(rev_data, DATE)
rev_data[weather_data]
```

With `dplyr`:

```
rev_data %>% right_join(
                  weather_data,
                  by = "DATE")
```

A full join will retain the 91 dates present in at least one of `rev_data` and `weather_data`. To perform a(n outer) full join of `rev_data` and `weather_data` using `DATE` as key:

With `data.table`:

```
merge(rev_data, weather_data,
      all = TRUE, by = "DATE")
```

With `dplyr`:

```
rev_data %>% full_join(
                weather_data,
                by = "DATE")
```

5.12.4 Semijoins and antijoins

Semijoins and antijoins are similar to joins, but work on observations rather than variables. That is, they are used for filtering one table using data from another table:

- *Semijoin*: retains all observations in the first table that have a match in the second table.
- *Antijoin*: retains all observations in the first table that *do not* have a match in the second table.

The same thing can be achieved using the filtering techniques of Section 5.8, but semijoins and antijoins are simpler to use when the filtering relies on conditions from another table.

Suppose that we are interested in the revenue of our business for days in February with subzero temperatures. First, we can create a table called `filter_data` listing all such days:

With `data.table`:

```
rev_data$DATE <- as.Date(rev_data$DATE)
weather_data$DATE <- as.Date(weather_data$DATE)
filter_data <- weather_data[TEMPERATURE < 0 &
                            DATE %between%
                            c("2020-02-01",
                              "2020-02-29"),]
```

With `dplyr`:

```
rev_data$DATE <- as.Date(rev_data$DATE)
weather_data$DATE <- as.Date(weather_data$DATE)
weather_data %>% filter(TEMPERATURE < 0,
                        between(DATE,
                        as.Date("2020-02-01"),
                        as.Date("2020-02-29"))
                                 ) -> filter_data
```

Next, we can use a semijoin to extract the rows of `rev_data` corresponding to the days of `filter_data`:

With `data.table`:

```
setkey(rev_data, DATE)
rev_data[rev_data[filter_data,
                  which = TRUE]]
```

With `dplyr`:

```
rev_data %>% semi_join(
                filter_data,
                by = "DATE")
```

If instead we wanted to find all days *except* the days in February with subzero temperatures, we could perform an antijoin:

With `data.table`:

```
setkey(rev_data, DATE)
rev_data[!filter_data]
```

With `dplyr`:

```
rev_data %>% anti_join(
                filter_data,
                by = "DATE")
```

~

Exercise 5.26. We return to the `oslo-biomarkers.xlsx` data from Exercises 5.8 and 5.25. Load the data frame that you created in Exercise 5.25 (or copy the code from its solution). You should also load the `oslo-covariates.xlsx` data from the book's web page - it contains information about the patients, such as age, gender and smoking habits.

Then do the following using `data.table` and/or `dplyr`/`tidyr`:

1. Merge the wide data frame from Exercise 5.25 with the `oslo-covariates.xlsx` data, using patient ID as key.
2. Use the `oslo-covariates.xlsx` data to select data for smokers from the wide data frame Exercise 5.25.

5.13 Scraping data from websites

Web scraping is the process of extracting data from a webpage. For instance, let's say that we'd like to download the list of Nobel laureates from the Wikipedia page https://en.wikipedia.org/wiki/List_of_Nobel_laureates. As with most sites, the text and formatting of the page is stored in an HTML file. In most browsers, you can view the HTML code by right-clicking on the page and choosing *View page source*. As you can see, all the information from the table can be found there, albeit in a format that is only just human-readable:

```
...
<tbody><tr>
<th>Year
</th>
<th width="18%"><a href="/wiki/List_of_Nobel_laureates_in_Physics"
  title="List of Nobel laureates in Physics">Physics</a>
</th>
<th width="16%"><a href="/wiki/List_of_Nobel_laureates_in_Chemistry"
title="List of Nobel laureates in Chemistry">Chemistry</a>
</th>
<th width="18%"><a href="/wiki/List_of_Nobel_laureates_in_Physiology_
or_Medicine" title="List of Nobel laureates in Physiology or Medicine
">Physiology<br />or Medicine</a>
</th>
<th width="16%"><a href="/wiki/List_of_Nobel_laureates_in_Literature"
title="List of Nobel laureates in Literature">Literature</a>
</th>
<th width="16%"><a href="/wiki/List_of_Nobel_Peace_Prize_laureates"
title="List of Nobel Peace Prize laureates">Peace</a>
</th>
<th width="15%"><a href="/wiki/List_of_Nobel_laureates_in_Economics"
class="mw-redirect" title="List of Nobel laureates in Economics">
```

```
  Economics</a><br />(The Sveriges Riksbank Prize)<sup id="cite_ref-
11" class="reference"><a href="#cite_note-11">&#91;11&#93;</a></sup>
</th></tr>
<tr>
<td align="center">1901
</td>
<td><span data-sort-value="Röntgen, Wilhelm"><span class="vcard"><span
class="fn"><a href="/wiki/Wilhelm_R%C3%B6ntgen" title="Wilhelm
Röntgen">  Wilhelm Röntgen</a></span></span></span>
</td>
<td><span data-sort-value="Hoff, Jacobus Henricus van &#39;t"><span
class="vcard"><span class="fn"><a href="/wiki/Jacobus_Henricus_van_
%27t_Hoff" title="Jacobus Henricus van &#39;t Hoff">Jacobus Henricus
van 't Hoff</a></span></span></span>
</td>
<td><span data-sort-value="von Behring, Emil Adolf"><span class=
"vcard">
<span class="fn"><a href="/wiki/Emil_Adolf_von_Behring" class="mw-
redirect" title="Emil Adolf von Behring">Emil Adolf von Behring</a>
</span></span></span>
</td>
...
```

To get hold of the data from the table, we could perhaps select all rows, copy them and paste them into a spreadsheet software such as Excel. But it would be much more convenient to be able to just import the table to R straight from the HTML file. Because tables written in HTML follow specific formats, it is possible to write code that automatically converts them to data frames in R. The `rvest` package contains a number of functions for that. Let's install it:

```
install.packages("rvest")
```

To read the entire Wikipedia page, we use:

```
library(rvest)
url <- "https://en.wikipedia.org/wiki/List_of_Nobel_laureates"
wiki <- read_html(url)
```

The object `wiki` now contains all the information from the page - you can have a quick look at it by using `html_text`:

```
html_text(wiki)
```

That is more information than we need. To extract all tables from `wiki`, we can use `html_nodes`:

```
tables <- html_nodes(wiki, "table")
tables
```

The first table, starting with the HTML code

```
<table class="wikitable sortable"><tbody>\n<tr>\n<th>Year\n</th>
```

is the one we are looking for. To transform it to a data frame, we use `html_table` as follows:

```
laureates <- html_table(tables[[1]], fill = TRUE)
View(laureates)
```

The `rvest` package can also be used for extracting data from more complex website structures using the SelectorGadget tool in the web browser Chrome. This let's you select the page elements that you wish to scrape in your browser, and helps you create the code needed to import them to R. For an example of how to use it, run `vignette("selectorgadget")`.

~

Exercise 5.27. Scrape the table containing different keytar models from https://en.wikipedia.org/wiki/List_of_keytars. Perform the necessary operations to convert the `Dates` column to `numeric`.

5.14 Other commons tasks

5.14.1 Deleting variables

If you no longer need a variable, you can delete it using `rm`:

```
my_variable <- c(1, 8, pi)
my_variable
rm(my_variable)
my_variable
```

This can be useful for instance if you have loaded a data frame that no longer is needed and takes up a lot of memory. If you, for some reason, want to wipe all your variables, you can use `ls`, which returns a vector containing the names of all variables, in combination with `rm`:

```
# Use this at your own risk! This deletes all currently loaded
# variables.

# Uncomment to run:
# rm(list = ls())
```

Variables are automatically deleted when you exit R (unless you choose to save your workspace). On the rare occasions where I want to wipe all variables from memory, I usually do a restart instead of using `rm`.

5.14.2 Importing data from other statistical packages

The `foreign` library contains functions for importing data from other statistical packages, such as Stata (`read.dta`), Minitab (`read.mtp`), SPSS (`read.spss`), and SAS (XPORT files, `read.xport`). They work just like `read.csv` (see Section 3.3), with additional arguments specific to the file format used for the statistical package in question.

5.14.3 Importing data from databases

R and RStudio have excellent support for connecting to databases. However, this requires some knowledge about databases and topics like ODBC drivers, and is therefore beyond the scope of the book. More information about using databases with R can be found at https://db.rstudio.com/.

5.14.4 Importing data from JSON files

JSON is a common file format for transmitting data between different systems. It is often used in web server systems where users can request data. One example of this is found in the JSON file at https://opendata-download-metobs.smhi.se/api/version/1.0/parameter/2/station/98210/period/latest-months/data.json. It contains daily mean temperatures from Stockholm, Sweden, during the last few months, accessible from the Swedish Meteorological and Hydrological Institute's server. Have a look at it in your web browser, and then install the `jsonlite` package:

```
install.packages("jsonlite")
```

We'll use the `fromJSON` function from `jsonlite` to import the data:

```
library(jsonlite)
url <- paste("https://opendata-download-metobs.smhi.se/api/version/",
             "1.0/parameter/2/station/98210/period/latest-months/",
             "data.json",
             sep = "")
stockholm <- fromJSON(url)
stockholm
```

By design, JSON files contain lists, and so `stockholm` is a `list` object. The temperature data that we were looking for is (in this particular case) contained in the list element called `value`:

```
stockholm$value
```

Chapter 6

R programming

The tools in Chapters 2-5 will allow you to manipulate, summarise and visualise your data in all sorts of ways. But what if you need to compute some statistic that there isn't a function for? What if you need automatic checks of your data and results? What if you need to repeat the same analysis for a large number of files? This is where the programming tools you'll learn about in this chapter, like loops and conditional statements, come in handy. And this is where you take the step from being able to use R for routine analyses to being able to use R for *any* analysis.

After working with the material in this chapter, you will be able to use R to:

- Write your own R functions,
- Use several new pipe operators,
- Use conditional statements to perform different operations depending on whether or not a condition is satisfied,
- Iterate code operations multiple times using loops,
- Iterate code operations multiple times using functionals,
- Measure the performance of your R code.

6.1 Functions

Suppose that we wish to compute the mean of a vector `x`. One way to do this would be to use `sum` and `length`:

```
x <- 1:100
# Compute mean:
sum(x)/length(x)
```

Now suppose that we wish to compute the mean of several vectors. We could do this by repeated use of `sum` and `length`:

```
x <- 1:100
y <- 1:200
z <- 1:300

# Compute means:
sum(x)/length(x)
sum(y)/length(y)
sum(z)/length(x)
```

But wait! I made a mistake when I copied the code to compute the mean of `z` - I forgot to change `length(x)` to `length(z)`! This is an easy mistake to make when you repeatedly copy and paste code. In addition, repeating the same code multiple times just doesn't look good. It would be much more convenient to have a single function for computing the means. Fortunately, such a function exists - `mean`:

```
# Compute means
mean(x)
mean(y)
mean(z)
```

As you can see, using `mean` makes the code shorter and easier to read and reduces the risk of errors induced by copying and pasting code (we only have to change the argument of one function instead of two).

You've already used a ton of different functions in R: functions for computing means, manipulating data, plotting graphics, and more. All these functions have been written by somebody who thought that they needed to repeat a task (e.g. computing a mean or plotting a bar chart) over and over again. And in such cases, it is much more convenient to have a function that does that task than to have to write or copy code every time you want to do it. This is true also for your own work - whenever you need to repeat the same task several times, it is probably a good idea to write a function for it. It will reduce the amount of code you have to write and lessen the risk of errors caused by copying and pasting old code. In this section, you will learn how to write your own functions.

6.1.1 Creating functions

For the sake of the example, let's say that we wish to compute the mean of several vectors but that the function `mean` doesn't exist. We would therefore like to write our own function for computing the mean of a vector. An R function takes some variables as input (arguments or parameters) and returns an object. Functions are defined using `function`. The definition follows a particular format:

```
function_name <- function(argument1, argument2, ...)
{
      # ...
```

```
      # Some rows with code that creates some_object
      # ...
      return(some_object)
}
```

In the case of our function for computing a mean, this could look like:

```
average <- function(x)
{
      avg <- sum(x)/length(x)
      return(avg)
}
```

This defines a function called `average`, that takes an object called `x` as input. It computes the sum of the elements of `x`, divides that by the number of elements in `x`, and returns the resulting mean.

If we now make a call to `average(x)`, our function will compute the mean value of the vector `x`. Let's try it out, to see that it works:

```
x <- 1:100
y <- 1:200
average(x)
average(y)
```

6.1.2 Local and global variables

Note that despite the fact that the vector was called `x` in the code we used to define the function, `average` works regardless of whether the input is called `x` or `y`. This is because R distinguishes between *global variables* and *local variables*. A global variable is created in the *global environment* outside a function, and is available to all functions (these are the variables that you can see in the Environment panel in RStudio). A local variable is created in the *local environment* inside a function, and is only available to that particular function. For instance, our `average` function creates a variable called `avg`, yet when we attempt to access `avg` after running `average` this variable doesn't seem to exist:

```
average(x)
avg
```

Because `avg` is a local variable, it is only available inside of the `average` function. Local variables take precedence over global variables inside the functions to which they belong. Because we named the argument used in the function `x`, `x` becomes the name of a local variable in `average`. As far as `average` is concerned, there is only one variable named `x`, and that is whatever object that was given as input to the function, regardless of what its original name was. Any operations performed on the

local variable `x` won't affect the global variable `x` at all.

Functions can access global variables:

```
y_squared <- function()
{
      return(y^2)
}

y <- 2
y_squared()
```

But operations performed on global variables inside functions won't affect the global variable:

```
add_to_y <- function(n)
{
      y <- y + n
}

y <- 1
add_to_y(1)
y
```

Suppose you really need to change a global variable inside a function[1]. In that case, you can use an alternative assignment operator, `<<-`, which assigns a value to the variable in the *parent environment* to the current environment. If you use `<<-` for assignment inside a function that is called from the global environment, this means that the assignment takes place in the global environment. But if you use `<<-` in a function (function 1) that is called by another function (function 2), the assignment will take place in the environment for function 2, thus affecting a local variable in function 2. Here is an example of a global assignment using `<<-`:

```
add_to_y_global <- function(n)
{
      y <<- y + n
}

y <- 1
add_to_y_global(1)
y
```

[1] Do you *really*?

6.1.3 Will your function work?

It is always a good idea to test if your function works as intended, and to try to figure out what can cause it to break. Let's return to our `average` function:

```
average <- function(x)
{
      avg <- sum(x)/length(x)
      return(avg)
}
```

We've already seen that it seems to work when the input `x` is a numeric vector. But what happens if we input something else instead?

```
average(c(1, 5, 8)) # Numeric input
average(c(TRUE, TRUE, FALSE)) # Logical input
average(c("Lady Gaga", "Tool", "Dry the River")) # Character input
average(data.frame(x = c(1, 1, 1), y = c(2, 2, 1))) # Numeric df
average(data.frame(x = c(1, 5, 8), y = c("A", "B", "C"))) # Mixed type
```

The first two of these render the desired output (the `logical` values being represented by 0's and 1's), but the rest don't. Many R functions include checks that the input is of the correct type, or checks to see which method should be applied depending on what data type the input is. We'll learn how to perform such checks in Section 6.3.

As a side note, it is possible to write functions that don't end with `return`. In that case, the output (i.e. what would be written in the Console if you'd run the code there) from the last line of the function will automatically be returned. I prefer to (almost) always use `return` though, as it is easy to accidentally make the function return nothing by finishing it with a line that yields no output. Below are two examples of how we could have written `average` without a call to `return`. The first doesn't work as intended, because the function's final (and only) line doesn't give any output.

```
average_bad <- function(x)
{
      avg <- sum(x)/length(x)
}

average_ok <- function(x)
{
      sum(x)/length(x)
}

average_bad(c(1, 5, 8))
average_ok(c(1, 5, 8))
```

6.1.4 More on arguments

It is possible to create functions with as many arguments as you like, but it will become quite unwieldy if the user has to supply too many arguments to your function. It is therefore common to provide default values to arguments, which is done by setting a value in the `function` call. Here is an example of a function that computes x^n, using $n = 2$ as the default:

```
power_n <- function(x, n = 2)
{
      return(x^n)
}
```

If we don't supply `n`, `power_n` uses the default `n = 2`:

```
power_n(3)
```

But if we supply an `n`, `power_n` will use that instead:

```
power_n(3, 1)
power_n(3, 3)
```

For clarity, you can specify which value corresponds to which argument:

```
power_n(x = 2, n = 5)
```

…and can then even put the arguments in the wrong order:

```
power_n(n = 5, x = 2)
```

However, if we only supply `n` we get an error, because there is no default value for `x`:

```
power_n(n = 5)
```

It is possible to pass a function as an argument. Here is a function that takes a vector and a function as input, and applies the function to the first two elements of the vector:

```
apply_to_first2 <- function(x, func)
{
      result <- func(x[1:2])
      return(result)
}
```

By supplying different functions to `apply_to_first2`, we can make it perform different tasks:

```
x <- c(4, 5, 6)
apply_to_first2(x, sqrt)
apply_to_first2(x, is.character)
```

```
apply_to_first2(x, power_n)
```

But what if the function that we supply requires additional arguments? Using `apply_to_first2` with `sum` and the vector `c(4, 5, 6)` works fine:

```
apply_to_first2(x, sum)
```

But if we instead use the vector `c(4, NA, 6)` the function returns `NA` :

```
x <- c(4, NA, 6)
apply_to_first2(x, sum)
```

Perhaps we'd like to pass `na.rm = TRUE` to `sum` to ensure that we get a `numeric` result, if at all possible. This can be done by adding `...` to the list of arguments for both functions, which indicates additional parameters (to be supplied by the user) that will be passed to `func`:

```
apply_to_first2 <- function(x, func, ...)
{
      result <- func(x[1:2], ...)
      return(result)
}

x <- c(4, NA, 6)
apply_to_first2(x, sum)
apply_to_first2(x, sum, na.rm = TRUE)
```

~

Exercise 6.1. Write a function that converts temperature measurements in degrees Fahrenheit to degrees Celsius, and apply it to the `Temp` column of the `airquality` data.

Exercise 6.2. Practice writing functions by doing the following:

1. Write a function that takes a vector as input and returns a vector containing its minimum and maximum, without using `min` and `max`.
2. Write a function that computes the mean of the squared values of a vector using `mean`, and that takes additional arguments that it passes on to `mean` (e.g. `na.rm`).

6.1.5 Namespaces

It is possible, and even likely, that you will encounter functions in packages with the same name as functions in other packages. Or, similarly, that there are functions in

packages with the same names as those you have written yourself. This is of course a bit of a headache, but it's actually something that can be overcome without changing the names of the functions. Just like variables can live in different environments, R functions live in *namespaces*, usually corresponding to either the global environment or the package they belong to. By specifying which namespace to look for the function in, you can use multiple functions that all have the same name.

For example, let's create a function called `sqrt`. There is already such a function in the `base` package[2] (see `?sqrt`), but let's do it anyway:

```
sqrt <- function(x)
{
      return(x^10)
}
```

If we now apply `sqrt` to an object, the function that we just defined will be used:

```
sqrt(4)
```

But if we want to use the `sqrt` from `base`, we can specify that by writing the namespace (which almost always is the package name) followed by `::` and the function name:

```
base::sqrt(4)
```

The `::` notation can also be used to call a function or object from a package without loading the package's namespace:

```
msleep # Doesn't work if ggplot2 isn't loaded
ggplot2::msleep # Works, without loading the ggplot2 namespace!
```

When you call a function, R will look for it in all active namespaces, following a particular order. To see the order of the namespaces, you can use `search`:

```
search()
```

Note that the global environment is first in this list - meaning that the functions that you define always will be preferred to functions in packages.

All this being said, note that it is bad practice to give your functions and variables the same names as common functions. Don't name them `mean`, `c` or `sqrt`. Nothing good can ever come from that sort of behaviour.

Nothing.

[2] `base` is automatically loaded when you start R, and contains core functions such as `sqrt`.

6.1.6 Sourcing other scripts

If you want to reuse a function that you have written in a new script, you can of course copy it into that script. But if you then make changes to your function, you will quickly end up with several different versions of it. A better idea can therefore be to put the function in a separate script, which you then can call in each script where you need the function. This is done using `source`. If, for instance, you have code that defines some functions in a file called `helper-functions.R` in your working directory, you can run it (thus defining the functions) when the rest of your code is run by adding `source("helper-functions.R")` to your code.

Another option is to create an R package containing the function, but that is beyond the scope of this book. Should you choose to go down that route, I highly recommend reading *R Packages* by Wickham and Bryan.

6.2 More on pipes

We have seen how the `magrittr` pipe `%>%` can be used to chain functions together. But there are also other pipe operators that are useful. In this section we'll look at some of them, and see how you can create functions using pipes.

6.2.1 *Ce ne sont pas non plus des pipes*

Although `%>%` is the most used pipe operator, the `magrittr` package provides a number of other pipes that are useful in certain situations.

One example is when you want to pass variables rather than an entire dataset to the next function. This is needed for instance if you want to use `cor` to compute the correlation between two variables, because `cor` takes two vectors as input instead of a data frame. You can do it using ordinary `%>%` pipes:

```
library(magrittr)
airquality %>%
      subset(Temp > 80) %>%
      {cor(.$Temp, .$Wind)}
```

However, the curly brackets `{}` and the dots `.` makes this a little awkward and difficult to read. A better option is to use the `%$%` pipe, which passes on the names of all variables in your data frame instead:

```
airquality %>%
      subset(Temp > 80) %$%
      cor(Temp, Wind)
```

If you want to modify a variable using a pipe, you can use the *compound assignment* pipe `%<>%`. The following three lines all yield exactly the same result:

```
x <- 1:8;   x <- sqrt(x);        x
x <- 1:8;   x %>% sqrt -> x;     x
x <- 1:8;   x %<>% sqrt;         x
```

As long as the first pipe in the pipeline is the compound assignment operator `%<>%`, you can combine it with other pipes:

```
x <- 1:8
x %<>% subset(x > 5) %>% sqrt
x
```

Sometimes you want to do something in the middle of a pipeline, like creating a plot, before continuing to the next step in the chain. The *tee* operator `%T>%` can be used to execute a function without passing on its output (if any). Instead, it passes on the output to its left. Here is an example:

```
airquality %>%
      subset(Temp > 80) %T>%
      plot %$%
      cor(Temp, Wind)
```

Note that if we'd used an ordinary pipe `%>%` instead, we'd get an error:

```
airquality %>%
      subset(Temp > 80) %>%
      plot %$%
      cor(Temp, Wind)
```

The reason is that `cor` looks for the variables `Temp` and `Wind` in the plot object, and not in the data frame. The tee operator takes care of this by passing on the data from its left side.

Remember that if you have a function where data only appears within parentheses, you need to wrap the function in curly brackets:

```
airquality %>%
      subset(Temp > 80) %T>%
      {cat("Number of rows in data:", nrow(.), "\n")} %$%
      cor(Temp, Wind)
```

When using the tee operator, this is true also for call to `ggplot`, where you additionally need to wrap the plot object in a call to `print`:

```
library(ggplot2)
airquality %>%
      subset(Temp > 80) %T>%
      {print(ggplot(., aes(Temp, Wind)) + geom_point())} %$%
      cor(Temp, Wind)
```

6.2.2 Writing functions with pipes

If you will be reusing the same pipeline multiple times, you may want to create a function for it. Let's say that you have a data frame containing only `numeric` variables, and that you want to create a scatterplot matrix (which can be done using `plot`) and compute the correlations between all variables (using `cor`). As an example, you could do this for `airquality` as follows:

```
airquality %T>% plot %>% cor
```

To define a function for this combination of operators, we simply write:

```
plot_and_cor <- . %T>% plot %>% cor
```

Note that we don't have to write `function(...)` when defining functions with pipes!

We can now use this function just like any other:

```
# With the airquality data:
airquality %>% plot_and_cor
plot_and_cor(airquality)

# With the bookstore data:
age <- c(28, 48, 47, 71, 22, 80, 48, 30, 31)
purchase <- c(20, 59, 2, 12, 22, 160, 34, 34, 29)
bookstore <- data.frame(age, purchase)
bookstore %>% plot_and_cor
```

~

Exercise 6.3. Write a function that takes a data frame as input and uses pipes to print the number of `NA` values in the data, remove all rows with `NA` values and return a summary of the remaining data.

Exercise 6.4. Pipes are operators, that is, functions that take two variables as input and can be written without parentheses (other examples of operators are `+` and `*`). You can define your own operators just as you would any other function. For instance, we can define an operator called `quadratic` that takes two numbers `a` and `b` as input and computes the quadratic expression $(a + b)^2$:

```
`%quadratic%` <- function(a, b) { (a + b)^2 }
2 %quadratic% 3
```

Create an operator called `%against%` that takes two vectors as input and draws a scatterplot of them.

6.3 Checking conditions

Sometimes you'd like your code to perform different operations depending on whether or not a certain condition is fulfilled. Perhaps you want it to do something different if there is missing data, if the input is a `character` vector, or if the largest value in a `numeric` vector is greater than some number. In Section 3.2.3 you learned how to filter data using conditions. In this section, you'll learn how to use conditional statements for a number of other tasks.

6.3.1 `if` and `else`

The most important functions for checking whether a condition is fulfilled are `if` and `else`. The basic syntax is

```
if(condition) { do something } else { do something else }
```

The condition should return a single `logical` value, so that it evaluates to either `TRUE` or `FALSE`. If the condition is fulfilled, i.e. if it is `TRUE`, the code inside the first pair of curly brackets will run, and if it's not (`FALSE`), the code within the second pair of curly brackets will run instead.

As a first example, assume that you want to compute the reciprocal of x, $1/x$, unless $x = 0$, in which case you wish to print an error message:

```
x <- 2
if(x == 0) { cat("Error! Division by zero.") } else { 1/x }
```

Now try running the same code with `x` set to `0`:

```
x <- 0
if(x == 0) { cat("Error! Division by zero.") } else { 1/x }
```

Alternatively, we could check if $x \neq 0$ and then change the order of the segments within the curly brackets:

```
x <- 0
if(x != 0) { 1/x } else { cat("Error! Division by zero.") }
```

You don't have to write all of the code on the same line, but you must make sure that the `else` part is on the same line as the first `}`:

```
if(x == 0)
{
    cat("Error! Division by zero.")
} else
{
    1/x
}
```

You can also choose not to have an `else` part at all. In that case, the code inside the curly brackets will run if the condition is satisfied, and if not, nothing will happen:

```
x <- 0
if(x == 0) { cat("x is 0.") }

x <- 2
if(x == 0) { cat("x is 0.") }
```

Finally, if you need to check a number of conditions one after another, in order to list different possibilities, you can do so by repeated use of `if` and `else`:

```
if(x == 0)
{
    cat("Error! Division by zero.")
} else if(is.infinite((x)))
{
    cat("Error! Divison by infinity.")
} else if(is.na((x)))
{
    cat("Error! Divison by NA.")
} else
{
    1/x
}
```

6.3.2 `&` & `&&`

Just as when we used conditions for filtering in Sections 3.2.3 and 5.8.2, it is possible to combine several conditions into one using `&` (AND) and `|` (OR). However, the `&` and `|` operators are vectorised, meaning that they will return a vector of `logical` values whenever possible. This is not desirable in conditional statements, where the condition must evaluate to a single value. Using a condition that returns a vector results in a warning message:

```
if(c(1, 2) == 2) { cat("The vector contains the number 2.\n") }
if(c(2, 1) == 2) { cat("The vector contains the number 2.\n") }
```

As you can see, only the first element of the `logical` vector is evaluated by `if`. Usually, if a condition evaluates to a vector, it is because you've made an error in your code. Remember, if you really need to evaluate a condition regarding the elements in a vector, you can collapse the resulting `logical` vector to a single value using `any` or `all`.

Some texts recommend using the operators `&&` and `||` instead of `&` and `|` in conditional statements. These work almost like `&` and `|`, but force the condition to evaluate to a single `logical`. I prefer to use `&` and `|`, because I want to be notified if my condition

evaluates to a vector - once again, that likely means that there is an error somewhere in my code!

There is, however, one case where I much prefer `&&` and `||`. `&` and `|` always evaluate all the conditions that you're combining, while `&&` and `||` don't: `&&` stops as soon as it encounters a `FALSE` and `||` stops as soon as it encounters a `TRUE`. Consequently, you can put the conditions you wish to combine in a particular order to make sure that they can be evaluated. For instance, you may want first to check that a variable exists, and then check a property. This can be done using `exists` to check whether or not it exists - note that the variable name must be written within quotes:

```
# a is a variable that doesn't exist

# Using && works:
if(exists("a") && a > 0)
{
    cat("The variable exists and is positive.")
} else { cat("a doesn't exist or is negative.") }

# But using & doesn't, because it attempts to evaluate a>0
# even though a doesn't exist:
if(exists("a") & a > 0)
{
    cat("The variable exists and is positive.")
} else { cat("a doesn't exist or is negative.") }
```

6.3.3 ifelse

It is common that you want to assign different values to a variable depending on whether or not a condition is satisfied:

```
x <- 2

if(x == 0)
{
    reciprocal <- "Error! Division by zero."
} else
{
    reciprocal <- 1/x
}

reciprocal
```

In fact, this situation is so common that there is a special function for it: `ifelse`:

```
reciprocal <- ifelse(x == 0, "Error! Division by zero.", 1/x)
```

`ifelse` evaluates a condition and then returns different answers depending on whether the condition is `TRUE` or `FALSE`. It can also be applied to vectors, in which case it checks the condition for each element of the vector and returns an answer for each element:

```
x <- c(-1, 1, 2, -2, 3)
ifelse(x > 0, "Positive", "Negative")
```

6.3.4 `switch`

For the sake of readability, it is usually a good idea to try to avoid chains of the type `if() {} else if() {} else if() {} else {}`. One function that can be useful for this is `switch`, which lets you list a number of possible results, either by position (a number) or by name:

```
position <- 2
switch(position,
      "First position",
      "Second position",
      "Third position")

name <- "First"
switch(name,
      First = "First name",
      Second = "Second name",
      Third = "Third name")
```

You can for instance use this to decide what function should be applied to your data:

```
x <- 1:3
y <- c(3, 5, 4)
method <- "nonparametric2"
cor_xy <- switch(method,
      parametric = cor(x, y, method = "pearson"),
      nonparametric1 = cor(x, y, method = "spearman"),
      nonparametric2 = cor(x, y, method = "kendall"))
cor_xy
```

6.3.5 Failing gracefully

Conditional statements are useful for ensuring that the input to a function you've written is of the correct type. In Section 6.1.3 we saw that our `average` function failed if we applied it to a `character` vector:

```
average <- function(x)
{
      avg <- sum(x)/length(x)
      return(avg)
}

average(c("Lady Gaga", "Tool", "Dry the River"))
```

By using a conditional statement, we can provide a more informative error message. We can check that the input is `numeric` and, if it's not, stop the function and print an error message, using `stop`:

```
average <- function(x)
{
      if(is.numeric(x))
      {
            avg <- sum(x)/length(x)
            return(avg)
      } else
      {
            stop("The input must be a numeric vector.")
      }
}

average(c(1, 5, 8))
average(c("Lady Gaga", "Tool", "Dry the River"))
```

~

Exercise 6.5. Which of the following conditions are `TRUE`? First think about the answer, and then check it using R.

```
x <- 2
y <- 3
z <- -3
```

1. `x > 2`
2. `x > y | x > z`
3. `x > y & x > z`
4. `abs(x*z) >= y`

Exercise 6.6. Fix the errors in the following code:

```
x <- c(1, 2, pi, 8)

# Only compute square roots if x exists
# and contains positive values:
if(exists(x)) { if(x > 0) { sqrt(x) } }
```

6.4 Iteration using loops

We have already seen how you can use functions to make it easier to repeat the same task over and over. But there is still a part of the puzzle missing - what if, for example, you wish to apply a function to each column of a data frame? What if you want to apply it to data from a number of files, one at a time? The solution to these problems is to use *iteration*. In this section, we'll explore how to perform iteration using *loops*.

6.4.1 `for` loops

`for` loops can be used to run the same code several times, with different settings, e.g. different data, in each iteration. Their use is perhaps best explained by some examples. We create the loop using `for`, give the name of a *control variable* and a vector containing its values (the control variable controls how many iterations to run) and then write the code that should be repeated in each iteration of the loop. In each iteration, a new value of the control variable is used in the code, and the loop stops when all values have been used.

As a first example, let's write a `for` loop that runs a block of code five times, where the block prints the current iteration number:

```
for(i in 1:5)
{
    cat("Iteration", i, "\n")
}
```

This is equivalent to writing:

```
cat("Iteration", 1, "\n")
cat("Iteration", 2, "\n")
cat("Iteration", 3, "\n")
cat("Iteration", 4, "\n")
cat("Iteration", 5, "\n")
```

The upside is that we didn't have to copy and edit the same code multiple times - and as you can imagine, this benefit becomes even more pronounced if you have more complicated code blocks.

The values for the control variable are given in a vector, and the code block will be run once for each element in the vector - we say the we *loop over the values in the vector*. The vector doesn't have to be `numeric` - here is an example with a `character` vector:

```
for(word in c("one", "two", "five hundred and fifty five"))
{
    cat("Iteration", word, "\n")
}
```

Of course, loops are used for so much more than merely printing text on the screen. A common use is to perform some computation and then store the result in a vector. In this case, we must first create an empty vector to store the result in, e.g. using `vector`, which creates an empty vector of a specific type and length:

```
squares <- vector("numeric", 5)

for(i in 1:5)
{
    squares[i] <- i^2
}
squares
```

In this case, it would have been both simpler and computationally faster to compute the squared values by running `(1:5)^2`. This is known as a *vectorised* solution, and is very important in R. We'll discuss vectorised solutions in detail in Section 6.5.

When creating the values used for the control variable, we often wish to create different sequences of numbers. Two functions that are very useful for this are `seq`, which creates sequences, and `rep`, which repeats patterns:

```
seq(0, 100)
seq(0, 100, by = 10)
seq(0, 100, length.out = 21)

rep(1, 4)
rep(c(1, 2), 4)
rep(c(1, 2), c(4, 2))
```

Finally, `seq_along` can be used to create a sequence of indices for a vector of a data frame, which is useful if you wish to iterate some code for each element of a vector or each column of a data frame:

```
seq_along(airquality) # Gives the indices of all column of the data
                      # frame
seq_along(airquality$Temp) # Gives the indices of all elements of the
                           # vector
```

Here is an example of how to use `seq_along` to compute the mean of each column of a data frame:

```
# Compute the mean for each column of the airquality data:
means <- vector("double", ncol(airquality))

# Loop over the variables in airquality:
for(i in seq_along(airquality))
{
      means[i] <- mean(airquality[[i]], na.rm = TRUE)
}

# Check that the results agree with those from the colMeans function:
means
colMeans(airquality, na.rm = TRUE)
```

The line inside the loop could have read `means[i] <- mean(airquality[,i], na.rm = TRUE)`, but that would have caused problems if we'd used it with a `data.table` or `tibble` object; see Section 5.9.4.

Finally, we can also change the values of the data in each iteration of the loop. Some machine learning methods require that the data is *standardised*, i.e. that all columns have mean 0 and standard deviation 1. This is achieved by subtracting the mean from each variable and then dividing each variable by its standard deviation. We can write a function for this that uses a loop, changing the values of a column in each iteration:

```
standardise <- function(df, ...)
{
      for(i in seq_along(df))
      {
          df[[i]] <- (df[[i]] - mean(df[[i]], ...))/sd(df[[i]], ...)
      }
      return(df)
}

# Try it out:
aqs <- standardise(airquality, na.rm = TRUE)
colMeans(aqs, na.rm = TRUE) # Non-zero due to floating point
                            # arithmetics!
sd(aqs$Wind)
```

∼

Exercise 6.7. Practice writing `for` loops by doing the following:

1. Compute the mean temperature for each month in the `airquality` dataset using a loop rather than an existing function.
2. Use a `for` loop to compute the maximum and minimum value of each column of the `airquality` data frame, storing the results in a data frame.
3. Make your solution to the previous task reusable by writing a function that returns the maximum and minimum value of each column of a data frame.

Exercise 6.8. Use `rep` or `seq` to create the following vectors:

1. `0.25 0.5 0.75 1`
2. `1 1 1 2 2 5`

Exercise 6.9. As an alternative to `seq_along(airquality)` and `seq_along(airquality$Temp)`, we could create the same sequences using `1:ncol(airquality)` and `1:length(airquality$Temp)`. Use `x <- c()` to create a vector of length zero. Then create loops that use `seq_along(x)` and `1:length(x)` as values for the control variable. How many iterations are the two loops run? Which solution is preferable?

Exercise 6.10. An alternative to standardisation is *normalisation*, where all `numeric` variables are rescaled so that their smallest value is 0 and their largest value is 1. Write a function that normalises the variables in a data frame containing `numeric` columns.

Exercise 6.11. The function `list.files` can be used to create a vector containing the names of all files in a folder. The `pattern` argument can be used to supply a regular expression describing a file name pattern. For instance, if `pattern = "\\.csv$"` is used, only `.csv` files will be listed.

Create a loop that goes through all `.csv` files in a folder and prints the names of the variables for each file.

6.4.2 Loops within loops

In some situations, you'll want to put a loop inside another loop. Such loops are said to be *nested*. An example is if we want to compute the correlation between all pairs of variables in `airquality`, and store the result in a matrix:

```
cor_mat <- matrix(NA, nrow = ncol(airquality),
                  ncol = ncol(airquality))
for(i in seq_along(airquality))
{
    for(j in seq_along(airquality))
    {
        cor_mat[i, j] <- cor(airquality[[i]], airquality[[j]],
```

```
                                use = "pairwise.complete")
    }
}

# Element [i, j] of the matrix now contains the correlation between
# variables i and j:
cor_mat
```

Once again, there is a vectorised solution to this problem, given by `cor(airquality, use = "pairwise.complete")`. As we will see in Section 6.6, vectorised solutions like this can be several times faster than solutions that use nested loops. In general, solutions involving nested loops tend to be fairly slow - but on the other hand, they are often easy and straightforward to implement.

6.4.3 Keeping track of what's happening

Sometimes each iteration of your loop takes a long time to run, and you'll want to monitor its progress. This can be done using printed messages or a progress bar in the Console panel, or sound notifications. We'll showcase each of these using a loop containing a call to `Sys.sleep`, which pauses the execution of R commands for a short time (determined by the user).

First, we can use `cat` to print a message describing the progress. Adding `\r` to the end of a string allows us to print all messages on the same line, with each new message replacing the old one:

```
# Print each message on a new same line:
for(i in 1:5)
{
    cat("Step", i, "out of 5\n")
    Sys.sleep(1) # Sleep for 1 second
}

# Replace the previous message with the new one:
for(i in 1:5)
{
    cat("Step", i, "out of 5\r")
    Sys.sleep(1) # Sleep for one second
}
```

Adding a progress bar is a little more complicated, because we must first start the bar by using `txtProgressBar` and the update it using `setTxtProgressBar`:

```
sequence <- 1:5
pbar <- txtProgressBar(min = 0, max = max(sequence), style = 3)
for(i in sequence)
```

```
{
    Sys.sleep(1) # Sleep for 1 second
    setTxtProgressBar(pbar, i)
}
close(pbar)
```

Finally, the `beepr` package[3] can be used to play sounds, with the function `beep`:

```
install.packages("beepr")

library(beepr)
# Play all 11 sounds available in beepr:
for(i in 1:11)
{
    beep(sound = i)
    Sys.sleep(2) # Sleep for 2 seconds
}
```

6.4.4 Loops and lists

In our previous examples of loops, it has always been clear from the start how many iterations the loop should run and what the length of the output vector (or data frame) should be. This isn't always the case. To begin with, let's consider the case where the length of the output is unknown or difficult to know in advance. Let's say that we want to go through the `airquality` data to find days that are extreme in the sense that at least one variable attains its maximum on those days. That is, we wish to find the index of the maximum of each variable, and store them in a vector. Because several days can have the same temperature or wind speed, there may be more than one such maximal index for each variable. For that reason, we don't know the length of the output vector in advance.

In such cases, it is usually a good idea to store the result from each iteration in a `list` (Section 5.2), and then collect the elements from the list once the loop has finished. We can create an empty list with one element for each variable in `airquality` using `vector`:

```
# Create an empty list with one element for each variable in
# airquality:
max_list <- vector("list", ncol(airquality))

# Naming the list elements will help us see which variable the maximal
# indices belong to:
names(max_list) <- names(airquality)
```

[3] Arguably the best add-on package for R.

```
# Loop over the variables to find the maxima:
for(i in seq_along(airquality))
{
      # Find indices of maximum values:
      max_index <- which(airquality[[i]] == max(airquality[[i]],
                                                na.rm = TRUE))

      # Add indices to list:
      max_list[[i]] <- max_index
}

# Check results:
max_list

# Collapse to a vector:
extreme_days <- unlist(max_list)
```

(In this case, only the variables `Month` and `Days` have duplicate maximum values.)

6.4.5 `while` loops

In some situations, we want to run a loop until a certain condition is met, meaning that we don't know in advance how many iterations we'll need. This is more common in numerical optimisation and simulation, but sometimes also occurs in data analyses.

When we don't know in advance how many iterations that are needed, we can use `while` loops. Unlike `for` loops, that iterate a fixed number of times, `while` loops keep iterating as long as some specified condition is met. Here is an example where the loop keeps iterating until `i` squared is greater than 100:

```
i <- 1

while(i^2 <= 100)
{
      cat(i,"squared is", i^2, "\n")
      i <- i +1
}

i
```

The code block inside the loop keeps repeating until the condition `i^2 <= 100` no longer is satisfied. We have to be a little bit careful with this condition - if we set it in such a way that it is possible that the condition *always* will be satisfied, the loop will just keep running and running - creating what is known as an *infinite loop*. If you've accidentally created an infinite loop, you can break it by pressing the Stop

button at the top of the Console panel in RStudio.

In Section 5.3.3 we saw how `rle` can be used to find and compute the lengths of runs of equal values in a vector. We can use nested `while` loops to create something similar. `while` loops are a good choice here, because we don't know how many runs are in the vector in advance. Here is an example, which you'll study in more detail in Exercise 6.12:

```
# Create a vector of 0's and 1's:
x <- rep(c(0, 1, 0, 1, 0), c(5, 1, 4, 2, 7))

# Create empty vectors where the results will be stored:
run_values <- run_lengths <- c()

# Set the initial condition:
i <- 1

# Iterate over the entire vector:
while(i < length(x))
{
    # A new run starts:
    run_length <- 1
    cat("A run starts at i =", i, "\n")

    # Check how long the run continues:
    while(x[i+1] == x[i] & i < length(x))
    {
          run_length <- run_length + 1
          i <- i + 1
    }

    i <- i + 1

    # Save results:
    run_values <- c(run_values, x[i-1])
    run_lengths <- c(run_lengths, run_length)
}

# Present the results:
data.frame(run_values, run_lengths)
```

~

Exercise 6.12. Consider the nested `while` loops in the run length example above.

Go through the code and think about what happens in each step. What happens when `i` is `1`? When it is `5`? When it is `6`? Answer the following questions:

1. What does the condition for the outer while loop check? Why is it needed?
2. What does the condition for the inner while loop check? Why is it needed?
3. What does the line `run_values <- c(run_values, x[i-1])` do?

Exercise 6.13. The *control statements* `break` and `next` can be used inside both `for` and `while` loops to control their behaviour further. `break` stops a loop, and `next` skips to the next iteration of it. Use these functions to modify the following piece of code so that the loop skips to the next iteration if `x[i]` is `0`, and breaks if `x[i]` is `NA`:

```
x <- c(1, 5, 8, 0, 20, 0, 3, NA, 18, 2)

for(i in seq_along(x))
{
      cat("Step", i, "- reciprocal is", 1/x[i], "\n")
}
```

Exercise 6.14. Using the `cor_mat` computation from Section 6.4.2, write a function that computes all pairwise correlations in a data frame, and uses `next` to only compute correlations for `numeric` variables. Test your function by applying it to the `msleep` data from `ggplot2`. Could you achieve the same thing without using `next`?

6.5 Iteration using vectorisation and functionals

Many operators and functions in R take vectors as input and handle them in a highly efficient way, usually by passing the vector on to an optimised function written in the C programming language[4]. So if we want to compute the squares of the numbers in a vector, we don't need to write a loop:

```
squares <- vector("numeric", 5)

for(i in 1:5)
{
    squares[i] <- i^2
}
squares
```

Instead, we can simply apply the `^` operator, which uses fast C code to compute the squares:

[4]Unlike R, C is a low-level language that allows the user to write highly specialised (and complex) code to perform operations very quickly.

```
squares <- (1:5)^2
```

These types of functions and operators are called *vectorised*. They take a vector as input and apply a function to all its elements, meaning that we can avoid slower solutions utilising loops in R[5]. Try to use vectorised solutions rather than loops whenever possible - it makes your code both easier to read and faster to run.

A related concept is *functionals*, which are functions that contain a `for` loop. Instead of writing a `for` loop, you can use a functional, supplying data, a function that should be applied in each iteration of the loop, and a vector to loop over. This won't necessarily make your loop run faster, but it does have other benefits:

- *Shorter code*: functionals allow you to write more concise code. Some would argue that they also allow you to write code that is easier to read, but that is obviously a matter of taste.
- *Efficient*: functionals handle memory allocation and other small tasks efficiently, meaning that you don't have to worry about creating a vector of an appropriate size to store the result.
- *No changes to your environment*: because all operations now take place in the local environment of the functional, you don't run the risk of accidentally changing variables in your global environment.
- *No left-overs*: `for` leaves the control variable (e.g. `i`) in the environment, functionals do not.
- *Easy to use with pipes*: because the loop has been wrapped in a function, it lends itself well to being used in a `%>%` pipeline.

Explicit loops are preferable when:

- You think that they are easier to read and write.
- Your functions take data frames or other non-vector objects as input.
- Each iteration of your loop depends on the results from previous iterations.

In this section, we'll see how we can apply functionals to obtain elegant alternatives to (explicit) loops.

6.5.1 A first example with `apply`

The prototypical functional is `apply`, which loops over either the rows or the columns of a data frame[6]. The arguments are a dataset, the margin to loop over (`1` for rows, `2` for columns) and then the function to be applied.

In Section 6.4.1 we wrote a `for` loop for computing the mean value of each column in a data frame:

[5] The vectorised functions often use loops, but loops written in C, which are much faster.

[6] Actually, over the rows or columns of a matrix - `apply` converts the data frame to a `matrix` object.

```
# Compute the mean for each column of the airquality data:
means <- vector("double", ncol(airquality))

# Loop over the variables in airquality:
for(i in seq_along(airquality))
{
      means[i] <- mean(airquality[[i]], na.rm = TRUE)
}
```

Using apply, we can reduce this to a single line. We wish to use the `airquality` data, loop over the columns (margin `2`) and apply the function `mean` to each column:

```
apply(airquality, 2, mean)
```

Rather elegant, don't you think?

Additional arguments can be passed to the function inside apply by adding them to the end of the function call:

```
apply(airquality, 2, mean, na.rm = TRUE)
```

~

Exercise 6.15. Use `apply` to compute the maximum and minimum value of each column of the `airquality` data frame. Can you write a function that allows you to compute both with a single call to `apply`?

6.5.2 Variations on a theme

There are several variations of `apply` that are tailored to specific problems:

- `lapply`: takes a function and vector/list as input, and returns a list.
- `sapply`: takes a function and vector/list as input, and returns a vector or matrix.
- `vapply`: a version of `sapply` with additional checks of the format of the output.
- `tapply`: for looping over groups, e.g. when computing grouped summaries.
- `rapply`: a recursive version of `tapply`.
- `mapply`: for applying a function to multiple arguments; see Section 6.5.7.
- `eapply`: for applying a function to all objects in an environment.

We have already seen several ways to compute the mean temperature for different months in the `airquality` data (Sections 3.8 and 5.7.7, and Exercise 6.7). The `*apply` family offer several more:

```
# Create a list:
temps <- split(airquality$Temp, airquality$Month)
```

```
lapply(temps, mean)
sapply(temps, mean)
vapply(temps, mean, vector("numeric", 1))
tapply(airquality$Temp, airquality$Month, mean)
```

There is, as that delightful proverb goes, more than one way to skin a cat.

~

Exercise 6.16. Use an `*apply` function to simultaneously compute the monthly maximum and minimum temperature in the `airquality` data frame.

Exercise 6.17. Use an `*apply` function to simultaneously compute the monthly maximum and minimum temperature and windspeed in the `airquality` data frame.

Hint: start by writing a function that simultaneously computes the maximum and minimum temperature and windspeed for a data frame containing data from a single month.

6.5.3 `purrr`

If you feel enthusiastic about ~~skinning cats~~ using functionals instead of loops, the tidyverse package `purrr` is a great addition to your toolbox. It contains a number of specialised alternatives to the `*apply` functions. More importantly, it also contains certain shortcuts that come in handy when working with functionals. For instance, it is fairly common to define a short function inside your functional, which is useful for instance when you don't want the function to take up space in your environment. This can be done a little more elegantly with `purrr` functions using a shortcut denoted by `~`. Let's say that we want to standardise all variables in `airquality`. The `map` function is the `purrr` equivalent of `lapply`. We can use it with or with the shortcut, and with or without pipes (we mention the use of pipes now because it will be important in what comes next):

```
# Base solution:
lapply(airquality, function(x) { (x-mean(x))/sd(x) })

# Base solution with pipe:
library(magrittr)
airquality %>% lapply(function(x) { (x-mean(x))/sd(x) })

# purrr solution:
library(purrr)
map(airquality, function(x) { (x-mean(x))/sd(x) })
```

```
# We can make the purrr solution less verbose using a shortcut:
map(airquality, ~(.-mean(.))/sd(.))

# purr solution with pipe and shortcut:
airquality %>% map(~(.-mean(.))/sd(.))
```

Where this shortcut really shines is if you need to use multiple functionals. Let's say that we want to standardise the `airquality` variables, compute a `summary` and then extract columns `2` and `5` from the summary (which contains the 1st and 3rd quartile of the data):

```
# Impenetrable base solution:
lapply(lapply(lapply(airquality,
                     function(x) { (x-mean(x))/sd(x) }),
              summary),
      function(x) { x[c(2, 5)] })

# Base solution with pipe:
airquality %>%
      lapply(function(x) { (x-mean(x))/sd(x) }) %>%
      lapply(summary) %>%
      lapply(function(x) { x[c(2, 5)] })

# purrr solution:
airquality %>%
      map(~(.-mean(.))/sd(.)) %>%
      map(summary) %>%
      map(~.[c(2, 5)])
```

Once you know the meaning of `~`, the `purrr` solution is a lot cleaner than the base solutions.

6.5.4 Specialised functions

So far, it may seem like `map` is just like `lapply` but with a shortcut for defining functions. Which is more or less true. But `purrr` contains a lot more functionals that you can use, each tailored to specific problems.

For instance, if you need to specify that the output should be a vector of a specific type, you can use:

- `map_dbl(data, function)` instead of `vapply(data, function, vector("numeric", length))`,
- `map_int(data, function)` instead of `vapply(data, function, vector("integer", length))`,

- `map_chr(data, function)` instead of `vapply(data, function, vector("character", length))`,
- `map_lgl(data, function)` instead of `vapply(data, function, vector("logical", length))`.

If you need to specify that the output should be a data frame, you can use:

- `map_dfr(data, function)` instead of `sapply(data, function)`.

The `~` shortcut for functions is available for all these `map_*` functions. In case you need to pass additional arguments to the function inside the functional, just add them at the end of the functional call:

```
airquality %>% map_dbl(max)
airquality %>% map_dbl(max, na.rm = TRUE)
```

Another specialised function is the `walk` function. It works just like `map`, but doesn't return anything. This is useful if you want to apply a function with no output, such as `cat` or `read.csv`:

```
# Returns a list of NULL values:
airquality %>% map(~cat("Maximum:", max(.), "\n"))

# Returns nothing:
airquality %>% walk(~cat("Maximum:", max(.), "\n"))
```

~

Exercise 6.18. Use a `map_*` function to simultaneously compute the monthly maximum and minimum temperature in the `airquality` data frame, returning a vector.

6.5.5 Exploring data with functionals

Functionals are great for creating custom summaries of your data. For instance, if you want to check the data type and number of unique values of each variable in your dataset, you can do that with a functional:

```
library(ggplot2)
diamonds %>% map_dfr(~(data.frame(unique_values = length(unique(.)),
                                  class = class(.))))
```

You can of course combine `purrr` functionals with functions from other packages, e.g. to replace `length(unique(.))` with a function from your favourite data manipulation package:

```
# Using uniqueN from data.table:
library(data.table)
```

```
dia <- as.data.table(diamonds)
dia %>% map_dfr(~(data.frame(unique_values = uniqueN(.),
                             class = class(.))))

# Using n_distinct from dplyr:
library(dplyr)
diamonds %>% map_dfr(~(data.frame(unique_values = n_distinct(.),
                                  class = class(.))))
```

When creating summaries it can often be useful to be able to loop over both the elements of a vector and their indices. In `purrr`, this is done using the usual `map*` functions, but with an `i` (for index) in the beginning of their names, e.g. `imap` and `iwalk`:

```
# Returns a list of NULL values:
imap(airquality, ~ cat(.y, ": ", median(.x), "\n", sep = ""))

# Returns nothing:
iwalk(airquality, ~ cat(.y, ": ", median(.x), "\n", sep = ""))
```

Note that `.x` is used to denote the variable, and that `.y` is used to denote the *name* of the variable. If `i*` functions are used on vectors without element names, indices are used instead. The names of elements of vectors can be set using `set_names`:

```
# Without element names:
x <- 1:5
iwalk(x, ~ cat(.y, ": ", exp(.x), "\n", sep = ""))

# Set element names:
x <- set_names(x, c("exp(1)", "exp(2)", "exp(3)", "exp(4)", "exp(5)"))
iwalk(x, ~ cat(.y, ": ", exp(.x), "\n", sep = ""))
```

∼

Exercise 6.19. Write a function that takes a data frame as input and returns the following information about each variable in the data frame: variable name, number of unique values, data type and number of missing values. The function should, as you will have guessed, use a functional.

Exercise 6.20. In Exercise 6.11 you wrote a function that printed the names and variables for all `.csv` files in a folder given by `folder_path`. Use `purrr` functionals to do the same thing.

6.5.6 Keep calm and carry on

Another neat feature of `purrr` is the `safely` function, which can be used to wrap a function that will be used inside a functional, and makes sure that the functional returns a result even if there is an error. For instance, let's say that we want to compute the logarithm of all variables in the `msleep` data:

```
library(ggplot2)
msleep
```

Note that some columns are `character` vectors, which will cause `log` to throw an error:

```
log(msleep$name)
log(msleep)
lapply(msleep, log)
map(msleep, log)
```

Note that the error messages we get from `lapply` and `map` here don't give any information about which variable caused the error, making it more difficult to figure out what's gone wrong.

If first we wrap `log` with `safely`, we get a list containing the correct output for the numeric variables, and error messages for the non-numeric variables:

```
safe_log <- safely(log)
lapply(msleep, safe_log)
map(msleep, safe_log)
```

Not only does this tell us where the errors occur, but it also returns the logarithms for all variables that `log` actually could be applied to.

If you'd like your functional to return some default value, e.g. `NA`, instead of an error message, you can use `possibly` instead of `safely`:

```
pos_log <- possibly(log, otherwise = NA)
map(msleep, pos_log)
```

6.5.7 Iterating over multiple variables

A final important case is when you want to iterate over more than one variable. This is often the case when fitting statistical models that should be used for prediction, as you'll see in Section 8.1.11. Another example is when you wish to create plots for several subsets in your data. For instance, we could create a plot of `carat` versus `price` for each combination of `color` and `cut` in the `diamonds` data. To do this for a single combination, we'd use something like this:

```
library(ggplot2)
library(dplyr)

diamonds %>% filter(cut == "Fair",
                    color == "D") %>%
             ggplot(aes(carat, price)) +
                 geom_point() +
                 ggtitle("Fair, D")
```

To create such a plot for all combinations of `color` and `cut`, we must first create a data frame containing all unique combinations, which can be done using the `distinct` function from `dplyr`:

```
combos <- diamonds %>% distinct(cut, color)
cuts <- combos$cut
colours <- combos$color
```

`map2` and `walk2` from `purrr` loop over the elements of two vectors, `x` and `y`, say. They combine the first element of `x` with the first element of `y`, the second element of `x` with the second element of `y`, and so on - meaning that they won't automatically loop over all combinations of elements. That is the reason why we use `distinct` above to create two vectors where each pair (`x[i]`, `y[i]`) correspond to a combination. Apart from the fact that we add a second vector to the call, `map2` and `walk2` work just like `map` and `walk`:

```
# Print all pairs:
walk2(cuts, colours, ~cat(.x, .y, "\n"))

# Create a plot for each pair:
combos_plots <- map2(cuts, colours, ~{
                     diamonds %>% filter(cut == .x,
                                         color == .y) %>%
                      ggplot(aes(carat, price)) +
                             geom_point() +
                             ggtitle(paste(.x, .y, sep =", "))})

# View some plots:
combos_plots[[1]]
combos_plots[[30]]

# Save all plots in a pdf file, with one plot per page:
pdf("all_combos_plots.pdf", width = 8, height = 8)
combos_plots
dev.off()
```

The base function `mapply` could also have been used here. If you need to iterate over more than two vectors, you can use `pmap` or `pwalk`, which work analogously to `map2` and `walk2`.

~

Exercise 6.21. Using the `gapminder` data from the `gapminder` package, create scatterplots of `pop` and `lifeExp` for each combination of `continent` and `year`. Save each plot as a separate `.png` file.

6.6 Measuring code performance

There are probably as many ideas about what good code is as there are programmers. Some prefer readable code; others prefer concise code. Some prefer to work with separate functions for each task, while others would rather continue to combine a few basic functions in new ways. Regardless of what you consider to be *good code*, there are a few objective measures that can be used to assess the quality of your code. In addition to writing code that works and is bug-free, you'd like your code to be:

- *Fast*: meaning that it runs quickly. Some tasks can take anything from second to weeks, depending on what code you write for them. Speed is particularly important if you're going to run your code many times.
- *Memory efficient*: meaning that it uses as little of your computer's memory as possible. Software running on your computer uses its memory - its RAM - to store data. If you're not careful with RAM, you may end up with a full memory and a sluggish or frozen computer. Memory efficiency is critical if you're working with big datasets, that take up a lot of RAM to begin with.

In this section we'll have a look at how you can measure the speed and memory efficiency of R functions. A caveat is that while speed and memory efficiency are important, the most important thing is to come up with a solution that works in the first place. You should almost always start by solving a problem, and then worry about speed and memory efficiency, not the other way around. The reasons for this is that efficient code often is more difficult to write, read, and debug, which can slow down the process of writing it considerably.

Note also, that speed and memory usage is system dependent - the clock frequency and architecture of your processor and speed and size of your RAM will affect how your code performs, as will what operating system you use and what other programs you are running at the same time. That means that if you wish to compare how two functions perform, you need to compare them on the same system under the same conditions.

As a side note, a great way to speed up functions that use either loops or functionals is parallelisation. We cover that topic in Section 10.2.

6.6.1 Timing functions

To measure how long a piece of code takes to run, we can use `system.time` as follows:

```
rtime <- system.time({
      x <- rnorm(1e6)
      mean(x)
      sd(x)
})

# elapsed is the total time it took to execute the code:
rtime
```

This isn't the best way of measuring computational time though, and doesn't allow us to compare different functions easily. Instead, we'll use the `bench` package, which contains a function called `mark` that is very useful for measuring the execution time of functions and blocks of code. Let's start by installing it:

```
install.packages("bench")
```

In Section 6.1.1 we wrote a function for computing the mean of a vector:

```
average <- function(x)
{
      return(sum(x)/length(x))
}
```

Is this faster or slower than `mean`? We can use `mark` to apply both functions to a vector multiple times, and measure how long each execution takes:

```
library(bench)
x <- 1:100
bm <- mark(mean(x), average(x))
bm # Or use View(bm) if you don't want to print the results in the
   # Console panel.
```

`mark` has executed both function `n_itr` times each, and measured how long each execution took to perform. The execution time varies - in the output you can see the shortest (`min`) and median (`median`) execution times, as well as the number of iterations per second (`itr/sec`). Be a little wary of the units for the execution times so that you don't get them confused - a millisecond (ms, 10^{-3} seconds) is 1,000 microseconds (µs, 1 µs is 10^{-6} seconds), and 1 microsecond is 1,000 nanoseconds (ns, 1 ns is 10^{-9} seconds).

The result here may surprise you - it appears that `average` is faster than `mean`! The reason is that `mean` does a lot of things that `average` does not: it checks the data type and gives error messages if the data is of the wrong type (e.g. `character`), and then traverses the vector twice to lower the risk of errors due to floating point arithmetics.

All of this takes time, and makes the function slower (but safer to use).

We can plot the results using the `ggbeeswarm` package:

```
install.packages("ggbeeswarm")

plot(bm)
```

It is also possible to place blocks of code inside curly brackets, `{ }`, in `mark`. Here is an example comparing a vectorised solution for computing the squares of a vector with a solution using a loop:

```
x <- 1:100
bm <- mark(x^2,
    {
        y <- x
        for(i in seq_along(x))
        {
            y[i] <- x[i]*x[i]
        }
        y
    })
bm
plot(bm)
```

Although the above code works, it isn't the prettiest, and the `bm` table looks a bit confusing because of the long expression for the code block. I prefer to put the code block inside a function instead:

```
squares <- function(x)
{
        y <- x
        for(i in seq_along(x))
        {
            y[i] <- x[i]*x[i]
        }
        return(y)
}

x <- 1:100
bm <- mark(x^2, squares(x))
bm
plot(bm)
```

Note that `squares(x)` is faster than the original code block:

```
bm <- mark(squares(x),
    {
        y <- x
        for(i in seq_along(x))
        {
            y[i] <- x[i]*x[i]
        }
        y
    })
bm
```

Functions in R are *compiled* the first time they are run, which often makes them run faster than the same code would have outside of the function. We'll discuss this further next.

6.6.2 Measuring memory usage - and a note on compilation

`mark` also shows us how much memory is allocated when running different code blocks, in the `mem_alloc` column of the output[7].

Unfortunately, measuring memory usage is a little tricky. To see why, restart R (yes, really - this is important!), and then run the following code to benchmark `x^2` versus `squares(x)`:

```
library(bench)

squares <- function(x)
{
        y <- x
        for(i in seq_along(x))
        {
            y[i] <- x[i]*x[i]
        }
        return(y)
}

x <- 1:100
bm <- mark(x^2, squares(x))
bm
```

Judging from the `mem_alloc` column, it appears that the `squares(x)` loop not only is slower, but also uses more memory. But wait! Let's run the code again, just to be

[7]But only if your version of R has been *compiled with memory profiling*. If you are using a standard build of R, i.e. have downloaded the base R binary from R-project.org, you should be good to go. You can check that memory profiling is enabled by checking that `capabilities("profmem")` returns `TRUE`. If not, you may need to reinstall R if you wish to enable memory profiling.

sure of the result:

```
bm <- mark(x^2, squares(x))
bm
```

This time out, both functions use less memory, and `squares` now uses *less* memory than `x^2`. What's going on?

Computers can't read code written in R or most other programming languages directly. Instead, the code must be translated to *machine code* that the computer's processor uses, in a process known as *compilation*. R uses *just-in-time compilation* of functions and loops[8], meaning that it translates the R code for new functions and loops to machine code *during execution*. Other languages, such as C, use ahead-of-time compilation, translating the code *prior to execution*. The latter can make the execution much faster, but some flexibility is lost, and the code needs to be run through a compiler ahead of execution, which also takes time. When doing the just-in-time compilation, R needs to use some of the computer's memory, which causes the memory usage to be greater the first time the function is run. However, if an R function is run again, it has already been compiled, meaning R doesn't have to allocate memory for compilation.

In conclusion, if you want to benchmark the memory usage of functions, make sure to run them once before benchmarking. Alternatively, if your function takes a long time to run, you can compile it without running it using the `cmpfun` function from the `compiler` package:

```
library(compiler)
squares <- cmpfun(squares)
squares(1:10)
```

~

Exercise 6.22. Write a function for computing the mean of a vector using a `for` loop. How much slower than `mean` is it? Which function uses more memory?

Exercise 6.23. We have seen three different ways of filtering a data frame to only keep rows that fulfil a condition: using base R, `data.table` and `dplyr`. Suppose that we want to extract all flights from 1 January from the `flights` data in the `nycflights13` package:

```
library(data.table)
library(dplyr)
library(nycflights13)
# Read about the data:
```

[8]Since R 3.4.

```
?flights

# Make a data.table copy of the data:
flights.dt <- as.data.table(flights)

# Filtering using base R:
flights0101 <- flights[flights$month == 1 & flights$day == 1,]
# Filtering using data.table:
flights0101 <- flights.dt[month == 1 & day == 1,]
# Filtering using dplyr:
flights0101 <- flights %>% filter(month ==1, day == 1)
```

Compare the speed and memory usage of these three approaches. Which has the best performance?

Chapter 7

Modern classical statistics

"Modern classical" may sound like a contradiction, but it is in fact anything but. Classical statistics covers topics like estimation, quantification of uncertainty, and hypothesis testing - all of which are at the heart of data analysis. Since the advent of modern computers, much has happened in this field that has yet to make it to the standard textbooks of introductory courses in statistics. This chapter attempts to bridge part of that gap by dealing with those classical topics, but with a modern approach that uses more recent advances in statistical theory and computational methods. Particular focus is put on how simulation can be used for analyses and for evaluating the properties of statistical procedures.

Whenever it is feasible, our aim in this chapter and the next is to:

- Use hypothesis tests that are based on permutations or the bootstrap rather than tests based on strict assumptions about the distribution of the data or asymptotic distributions,
- To complement estimates and hypothesis tests with computing confidence intervals based on sound methods (including the bootstrap),
- Offer easy-to-use Bayesian methods as an alternative to frequentist tools.

After reading this chapter, you will be able to use R to:

- Generate random numbers,
- Perform simulations to assess the performance of statistical methods,
- Perform hypothesis tests,
- Compute confidence intervals,
- Make sample size computations,
- Report statistical results.

7.1 Simulation and distributions

A *random variable* is a variable whose value describes the outcome of a random phenomenon. A (probability) *distribution* is a mathematical function that describes the probability of different outcomes for a random variable. Random variables and distributions are at the heart of probability theory and most, if not all, statistical models.

As we shall soon see, they are also invaluable tools when evaluating statistical methods. A key component of modern statistical work is *simulation*, in which we generate artificial data that can be used both in the analysis of real data (e.g. in permutation tests and bootstrap confidence intervals, topics that we'll explore in this chapter) and for assessing different methods. Simulation is possible only because we can generate random numbers, so let's begin by having a look at how we can generate random numbers in R.

7.1.1 Generating random numbers

The function `sample` can be used to randomly draw a number of elements from a vector. For instance, we can use it to draw 2 random numbers from the first ten integers: $1, 2, \ldots, 9, 10$:

```
sample(1:10, 2)
```

Try running the above code multiple times. You'll get different results each time, because each time it runs the random number generator is in a different *state*. In most cases, this is desirable (if the results were the same each time we used `sample`, it wouldn't be random), but not if we want to replicate a result at some later stage.

When we are concerned about reproducibility, we can use `set.seed` to fix the state of the random number generator:

```
# Each run generates different results:
sample(1:10, 2); sample(1:10, 2)

# To get the same result each time, set the seed to a
# number of your choice:
set.seed(314); sample(1:10, 2)
set.seed(314); sample(1:10, 2)
```

We often want to use simulated data from a probability distribution, such as the normal distribution. The normal distribution is defined by its mean μ and its variance σ^2 (or, equivalently, its standard deviation σ). There are special functions for generating data from different distributions - for the normal distribution it is called `rnorm`. We specify the number of observations that we want to generate (`n`) and the parameters of the distribution (the mean `mu` and the standard deviation `sigma`):

```
rnorm(n = 10, mu = 2, sigma = 1)

# A shorter version:
rnorm(10, 2, 1)
```

Similarly, there are functions that can be used compute the quantile function, density function, and cumulative distribution function (CDF) of the normal distribution. Here are some examples for a normal distribution with mean 2 and standard deviation 1:

```
qnorm(0.9, 2, 1)     # Upper 90 % quantile of distribution
dnorm(2.5, 2, 1)     # Density function f(2.5)
pnorm(2.5, 2, 1)     # Cumulative distribution function F(2.5)
```

~

Exercise 7.1. Sampling can be done with or without *replacement*. If replacement is used, an observation can be drawn more than once. Check the documentation for `sample`. How can you change the settings to sample with replacement? Draw 5 random numbers from the first ten integers, with replacement.

7.1.2 Some common distributions

Next, we provide the syntax for random number generation, quantile functions, density/probability functions and cumulative distribution functions for some of the most commonly used distributions. This section is mainly intended as a reference, for you to look up when you need to use one of these distributions - so there is no need to run all the code chunks below right now.

Normal distribution $N(\mu, \sigma^2)$ with mean μ and variance σ^2:

```
rnorm(n, mu, sigma)     # Generate n random numbers
qnorm(0.95, mu, sigma) # Upper 95 % quantile of distribution
dnorm(x, mu, sigma)     # Density function f(x)
pnorm(x, mu, sigma)     # Cumulative distribution function F(X)
```

Continuous uniform distribution $U(a, b)$ on the interval (a, b), with mean $\frac{a+b}{2}$ and variance $\frac{(b-a)^2}{12}$:

```
runif(n, a, b)     # Generate n random numbers
qunif(0.95, a, b) # Upper 95 % quantile of distribution
dunif(x, a, b)     # Density function f(x)
punif(x, a, b)     # Cumulative distribution function F(X)
```

Exponential distribution $Exp(m)$ with mean m and variance m^2:

```
rexp(n, 1/m)     # Generate n random numbers
qexp(0.95, 1/m) # Upper 95 % quantile of distribution
dexp(x, 1/m)     # Density function f(x)
pexp(x, 1/m)     # Cumulative distribution function F(X)
```

Gamma distribution $\Gamma(\alpha, \beta)$ with mean $\frac{\alpha}{\beta}$ and variance $\frac{\alpha}{\beta^2}$:

```
rgamma(n, alpha, beta)     # Generate n random numbers
qgamma(0.95, alpha, beta) # Upper 95 % quantile of distribution
dgamma(x, alpha, beta)     # Density function f(x)
pgamma(x, alpha, beta)     # Cumulative distribution function F(X)
```

Lognormal distribution $LN(\mu, \sigma^2)$ with mean $\exp(\mu + \sigma^2/2)$ and variance $(\exp(\sigma^2) - 1)\exp(2\mu + \sigma^2)$:

```
rlnorm(n, mu, sigma)     # Generate n random numbers
qlnorm(0.95, mu, sigma) # Upper 95 % quantile of distribution
dlnorm(x, mu, sigma)     # Density function f(x)
plnorm(x, mu, sigma)     # Cumulative distribution function F(X)
```

t-distribution $t(\nu)$ with mean 0 (for $\nu > 1$) and variance $\frac{\nu}{\nu-2}$ (for $\nu > 2$):

```
rt(n, nu)     # Generate n random numbers
qt(0.95, nu) # Upper 95 % quantile of distribution
dt(x, nu)     # Density function f(x)
pt(x, nu)     # Cumulative distribution function F(X)
```

Chi-squared distribution $\chi^2(k)$ with mean k and variance $2k$:

```
rchisq(n, k)     # Generate n random numbers
qchisq(0.95, k) # Upper 95 % quantile of distribution
dchisq(x, k)     # Density function f(x)
pchisq(x, k)     # Cumulative distribution function F(X)
```

F-distribution $F(d_1, d_2)$ with mean $\frac{d_2}{d_2-2}$ (for $d_2 > 2$) and variance $\frac{2d_2^2(d_1+d_2-2)}{d_1(d_2-2)^2(d_2-4)}$ (for $d_2 > 4$):

```
rf(n, d1, d2)     # Generate n random numbers
qf(0.95, d1, d2) # Upper 95 % quantile of distribution
df(x, d1, d2)     # Density function f(x)
pf(x, d1, d2)     # Cumulative distribution function F(X)
```

Beta distribution $Beta(\alpha, \beta)$ with mean $\frac{\alpha}{\alpha+\beta}$ and variance $\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$:

```
rbeta(n, alpha, beta)     # Generate n random numbers
qbeta(0.95, alpha, beta) # Upper 95 % quantile of distribution
dbeta(x, alpha, beta)     # Density function f(x)
pbeta(x, alpha, beta)     # Cumulative distribution function F(X)
```

Binomial distribution $Bin(n, p)$ with mean np and variance $np(1-p)$:

```
rbinom(n, n, p)     # Generate n random numbers
qbinom(0.95, n, p) # Upper 95 % quantile of distribution
dbinom(x, n, p)     # Probability function f(x)
pbinom(x, n, p)     # Cumulative distribution function F(X)
```

Poisson distribution $Po(\lambda)$ with mean λ and variance λ:

```
rpois(n, lambda)     # Generate n random numbers
qpois(0.95, lambda) # Upper 95 % quantile of distribution
dpois(x, lambda)     # Probability function f(x)
ppois(x, lambda)     # Cumulative distribution function F(X)
```

Negative binomial distribution $NegBin(r, p)$ with mean $\frac{rp}{1-p}$ and variance $\frac{rp}{(1-p)^2}$:

```
rnbinom(n, r, p)     # Generate n random numbers
qnbinom(0.95, r, p) # Upper 95 % quantile of distribution
dnbinom(x, r, p)     # Probability function f(x)
pnbinom(x, r, p)     # Cumulative distribution function F(X)
```

Multivariate normal distribution with mean vector μ and covariance matrix Σ:

```
library(MASS)
mvrnorm(n, mu, Sigma) # Generate n random numbers
```

~

Exercise 7.2. Use `runif` and (at least) one of `round`, `ceiling` and `floor` to generate observations from a discrete random variable on the integers $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$.

7.1.3 Assessing distributional assumptions

So how can we know that the functions for generating random observations from distributions work? And when working with real data, how can we know what distribution fits the data? One answer is that we can visually compare the distribution of the generated (or real) data to the target distribution. This can for instance be done by comparing a histogram of the data to the target distribution's density function.

To do so, we must add `aes(y = ..density..))` to the call to `geom_histogram`, which rescales the histogram to have area 1 (just like a density function has). We can then add the density function using `geom_function`:

```
# Generate data from a normal distribution with mean 10 and
# standard deviation 1
```

```
generated_data <- data.frame(normal_data = rnorm(1000, 10, 1))

library(ggplot2)
# Compare to histogram:
ggplot(generated_data, aes(x = normal_data)) +
      geom_histogram(colour = "black", aes(y = ..density..)) +
      geom_function(fun = dnorm, colour = "red", size = 2,
                  args = list(mean = mean(generated_data$normal_data),
                              sd = sd(generated_data$normal_data)))
```

Try increasing the number of observations generated. As the number of observations increase, the histogram should start to look more and more like the density function.

We could also add a density estimate for the generated data, to further aid the eye here - we'd expect this to be close to the theoretical density function:

```
# Compare to density estimate:
ggplot(generated_data, aes(x = normal_data)) +
      geom_histogram(colour = "black", aes(y = ..density..)) +
      geom_density(colour = "blue", size = 2) +
      geom_function(fun = dnorm, colour = "red", size = 2,
                 args = list(mean = mean(generated_data$normal_data),
                             sd = sd(generated_data$normal_data)))
```

If instead we wished to compare the distribution of the data to a χ^2 distribution, we would change the value of `fun` and `args` in `geom_function` accordingly:

```
# Compare to density estimate:
ggplot(generated_data, aes(x = normal_data)) +
      geom_histogram(colour = "black", aes(y = ..density..)) +
      geom_density(colour = "blue", size = 2) +
      geom_function(fun = dchisq, colour = "red", size = 2,
                  args = list(df = mean(generated_data$normal_data)))
```

Note that the values of `args` have changed. `args` should always be a list containing values for the parameters of the distribution: `mu` and `sigma` for the normal distribution and `df` for the χ^2 distribution (the same as in Section 7.1.2).

Another option is to draw a quantile-quantile plot, or Q-Q plot for short, which compares the theoretical quantiles of a distribution to the empirical quantiles of the data, showing each observation as a point. If the data follows the theorised distribution, then the points should lie more or less along a straight line.

To draw a Q-Q plot for a normal distribution, we use the geoms `geom_qq` and `geom_qq_line`:

```
# Q-Q plot for normality:
ggplot(generated_data, aes(sample = normal_data)) +
        geom_qq() + geom_qq_line()
```

For all other distributions, we must provide the quantile function of the distribution (many of which can be found in Section 7.1.2):

```
# Q-Q plot for the lognormal distribution:
ggplot(generated_data, aes(sample = normal_data)) +
        geom_qq(distribution = qlnorm) +
        geom_qq_line(distribution = qlnorm)
```

Q-Q-plots can be a little difficult to read. There will always be points deviating from the line - in fact, that's expected. So how much must they deviate before we rule out a distributional assumption? Particularly when working with real data, I like to compare the Q-Q-plot of my data to Q-Q-plots of simulated samples from the assumed distribution, to get a feel for what kind of deviations can appear if the distributional assumption holds. Here's an example of how to do this, for the normal distribution:

```
# Look at solar radiation data for May from the airquality
# dataset:
May <- airquality[airquality$Month == 5,]

# Create a Q-Q-plot for the solar radiation data, and store
# it in a list:
qqplots <- list(ggplot(May, aes(sample = Solar.R)) +
  geom_qq() + geom_qq_line() + ggtitle("Actual data"))

# Compute the sample size n:
n <- sum(!is.na(May$Temp))

# Generate 8 new datasets of size n from a normal distribution.
# Then draw Q-Q-plots for these and store them in the list:
for(i in 2:9)
{
    generated_data <- data.frame(normal_data = rnorm(n, 10, 1))
    qqplots[[i]] <- ggplot(generated_data, aes(sample = normal_data)) +
      geom_qq() + geom_qq_line() + ggtitle("Simulated data")
}

# Plot the resulting Q-Q-plots side-by-side:
library(patchwork)
(qqplots[[1]] + qqplots[[2]] + qqplots[[3]]) /
  (qqplots[[4]] + qqplots[[5]] + qqplots[[6]]) /
```

```
(qqplots[[7]] + qqplots[[8]] + qqplots[[9]])
```

You can run the code several times, to get more examples of what Q-Q-plots can look like when the distributional assumption holds. In this case, the tail points in the Q-Q-plot for the solar radiation data deviate from the line more than the tail points in most simulated examples do, and personally, I'd be reluctant to assume that the data comes from a normal distribution.

~

Exercise 7.3. Investigate the sleeping times in the `msleep` data from the `ggplot2` package. Do they appear to follow a normal distribution? A lognormal distribution?

Exercise 7.4. Another approach to assessing distributional assumptions for real data is to use formal hypothesis tests. One example is the Shapiro-Wilk test for normality, available in `shapiro.test`. The null hypothesis is that the data comes from a normal distribution, and the alternative is that it doesn't (meaning that a low p-value is supposed to imply non-normality).

1. Apply `shapiro.test` to the sleeping times in the `msleep` dataset. According to the Shapiro-Wilk test, is the data normally distributed?
2. Generate 2,000 observations from a $\chi^2(100)$ distribution. Compare the histogram of the generated data to the density function of a normal distribution. Are they similar? What are the results when you apply the Shapiro-Wilk test to the data?

7.1.4 Monte Carlo integration

In this chapter, we will use simulation to compute p-values and confidence intervals, to compare different statistical methods, and to perform sample size computations. Another important use of simulation is in *Monte Carlo integration*, in which random numbers are used for numerical integration. It plays an important role in for instance statistical physics, computational biology, computational linguistics, and Bayesian statistics; fields that require the computation of complicated integrals.

To create an example of Monte Carlo integration, let's start by writing a function, `circle`, that defines a quarter-circle on the unit square. We will then plot it using the geom `geom_function`:

```
circle <- function(x)
{
      return(sqrt(1-x^2))
}
```

```
ggplot(data.frame(x = c(0, 1)), aes(x)) +
      geom_function(fun = circle)
```

Let's say that we are interest in computing the area under quarter-circle. We can highlight the area in our plot using `geom_area`:

```
ggplot(data.frame(x = seq(0, 1, 1e-4)), aes(x)) +
      geom_area(aes(x = x,
                    y = ifelse(x^2 + circle(x)^2 <= 1, circle(x), 0)),
                    fill = "pink") +
      geom_function(fun = circle)
```

To find the area, we will generate a large number of random points uniformly in the unit square. By the law of large numbers, the proportion of points that end up under the quarter-circle should be close to the area under the quarter-circle[1]. To do this, we generate 10,000 random values for the x and y coordinates of each point using the $U(0, 1)$ distribution, that is, using `runif`:

```
B <- 1e4
unif_points <- data.frame(x = runif(B), y = runif(B))
```

Next, we add the points to our plot:

```
ggplot(unif_points, aes(x, y)) +
      geom_area(aes(x = x,
                    y = ifelse(x^2 + circle(x)^2 <= 1, circle(x), 0)),
                    fill = "pink") +
      geom_point(size = 0.5, alpha = 0.25,
               colour = ifelse(unif_points$x^2 + unif_points$y^2 <= 1,
                               "red", "black")) +
      geom_function(fun = circle)
```

Note the order in which we placed the geoms - we plot the points after the area so that the pink colour won't cover the points, and the function after the points so that the points won't cover the curve.

To estimate the area, we compute the proportion of points that are below the curve:

```
mean(unif_points$x^2 + unif_points$y^2 <= 1)
```

In this case, we can also compute the area exactly: $\int_0^1 \sqrt{1-x^2}dx = \pi/4 = 0.7853\ldots$. For more complicated integrals, however, numerical integration methods like Monte Carlo integration may be required. That being said, there are better numerical integration methods for low-dimensional integrals like this one. Monte Carlo integration

[1] In general, the proportion of points that fall below the curve will be proportional to the area under the curve *relative* to the area of the sample space. In this case the sample space is the unit square, which has area 1, meaning that the relative area is the same as the absolute area.

is primarily used for higher-dimensional integrals, where other techniques fail.

7.2 Student's t-test revisited

For decades teachers all over the world have been telling the story of William Sealy Gosset: the head brewer at Guinness who derived the formulas used for the t-test and, following company policy, published the results under the pseudonym "Student".

Gosset's work was hugely important, but the passing of time has rendered at least parts of it largely obsolete. His distributional formulas were derived out of necessity: lacking the computer power that we have available to us today, he was forced to impose the assumption of normality on the data, in order to derive the formulas he needed to be able to carry out his analyses. Today we can use simulation to carry out analyses with fewer assumptions. As an added bonus, these simulation techniques often happen to result in statistical methods with better performance than Student's t-test and other similar methods.

7.2.1 The old-school t-test

The *really* old-school way of performing a t-test - the way statistical pioneers like Gosset and Fisher would have done it - is to look up p-values using tables covering several pages. There haven't really been any excuses for doing that since the advent of the personal computer though, so let's not go further into that. The "modern" version of the old-school t-test uses numerical evaluation of the formulas for Student's t-distribution to compute p-values and confidence intervals. Before we delve into more modern approaches, let's look at how we can run an old-school t-test in R.

In Section 3.6 we used `t.test` to run a t-test to see if there is a difference in how long carnivores and herbivores sleep, using the `msleep` data from `ggplot2`[2]. First, we subtracted a subset of the data corresponding to carnivores and herbivores, and then we ran the test. There are in fact several different ways of doing this, and it is probably a good idea to have a look at them.

In the approach used in Section 3.6 we created two vectors, using bracket notation, and then used those as arguments for `t.test`:

```
library(ggplot2)
carnivores <- msleep[msleep$vore == "carni",]
herbivores <- msleep[msleep$vore == "herbi",]
t.test(carnivores$sleep_total, herbivores$sleep_total)
```

Alternatively, we could have used formula notation, as we e.g. did for the linear model in Section 3.7. We'd then have to use the `data` argument in `t.test` to supply the

[2]Note that this is not a random sample of mammals, and so one of the fundamental assumptions behind the t-test isn't valid in this case. For the purpose of showing how to use the t-test, the data is good enough though.

data. By using `subset`, we can do the subsetting simultaneously:

```
t.test(sleep_total ~ vore, data =
         subset(msleep, vore == "carni" | vore == "herbi"))
```

Unless we are interested in keeping the vectors `carnivores` and `herbivores` for other purposes, this latter approach is arguably more elegant.

Speaking of elegance, the `data` argument also makes it easy to run a t-test using pipes. Here is an example, where we use `filter` from `dplyr` to do the subsetting:

```
library(dplyr)
msleep %>% filter(vore == "carni" | vore == "herbi") %>%
        t.test(sleep_total ~ vore, data = .)
```

We could also use the `magrittr` pipe `%$%` from Section 6.2 to pass the variables from the filtered subset of `msleep`, avoiding the `data` argument:

```
library(magrittr)
msleep %>% filter(vore == "carni" | vore == "herbi") %$%
        t.test(sleep_total ~ vore)
```

There are even more options than this - the point that I'm trying to make here is that like most functions in R, you can use functions for classical statistics in many different ways. In what follows, I will show you one or two of these, but don't hesitate to try out other approaches if they seem better to you.

What we just did above was a two-sided t-test, where the null hypothesis was that there was no difference in means between the groups, and the alternative hypothesis that there was a difference. We can also perform one-sided tests using the `alternative` argument. `alternative = "greater"` means that the alternative is that the first group has a greater mean, and `alternative = "less"` means that the first group has a smaller mean. Here is an example with the former:

```
t.test(sleep_total ~ vore,
       data = subset(msleep, vore == "carni" | vore == "herbi"),
       alternative = "greater")
```

By default, R uses the Welch two-sample t-test, meaning that it is *not* assumed that the groups have equal variances. If you don't want to make that assumption, you can add `var.equal = TRUE`:

```
t.test(sleep_total ~ vore,
       data = subset(msleep, vore == "carni" | vore == "herbi"),
       var.equal = TRUE)
```

In addition to two-sample t-tests, `t.test` can also be used for one-sample tests and paired t-tests. To perform a one-sample t-test, all we need to do is to supply a single vector with observations, along with the value of the mean μ under the null

hypothesis. I usually sleep for about 7 hours each night, and so if I want to test whether that is true for an average mammal, I'd use the following:

```
t.test(msleep$sleep_total, mu = 7)
```

As we can see from the output, your average mammal sleeps for 10.4 hours per day. Moreover, the p-value is quite low - apparently, I sleep unusually little for a mammal!

As for paired t-tests, we can perform them by supplying two vectors (where element 1 of the first vector corresponds to element 1 of the second vector, and so on) and the argument `paired = TRUE`. For instance, using the `diamonds` data from `ggplot2`, we could run a test to see if the length `x` of diamonds with a fair quality of the cut on average equals the width `y`:

```
fair_diamonds <- subset(diamonds, cut == "Fair")
t.test(fair_diamonds$x, fair_diamonds$y, paired = TRUE)
```

~

Exercise 7.5. Load the VAS pain data `vas.csv` from Exercise 3.8. Perform a one-sided t-test to see test the null hypothesis that the average VAS among the patients during the time period is less than or equal to 6.

7.2.2 Permutation tests

Maybe it was a little harsh to say that Gosset's formulas have become obsolete. The formulas are mathematical approximations to the distribution of the test statistics under the null hypothesis. The truth is that they work very well as long as your data is (nearly) normally distributed. The two-sample test also works well for non-normal data as long as you have balanced sample sizes, that is, equally many observations in both groups. However, for one-sample tests, and two-sample tests with imbalanced sample sizes, there are better ways to compute p-values and confidence intervals than to use Gosset's traditional formulas.

The first option that we'll look at is permutation tests. Let's return to our mammal sleeping times example, where we wanted to investigate whether there are differences in how long carnivores and herbivores sleep on average:

```
t.test(sleep_total ~ vore, data =
         subset(msleep, vore == "carni" | vore == "herbi"))
```

There are 19 carnivores and 32 herbivores - 51 animals in total. If there are no differences between the two groups, the `vore` labels offer no information about how long the animals sleep each day. Under the null hypothesis, the assignment of `vore` labels to different animals is therefore for all intents and purposes random. To find the distribution of the test statistic under the null hypothesis, we could look at all

possible ways to assign 19 animals the label `carnivore` and 32 animals the label `herbivore`. That is, look at all permutations of the labels. The probability of a result at least as extreme as that obtained in our sample (in the direction of the alternative), i.e. the p-value, would then be the proportion of permutations that yield a result at least extreme as that in our sample. This is known as a permutation test.

Permutation tests were known to the likes of Gosset and Fisher (Fisher's exact test is a common example), but because the number of permutations of labels often tend to become quite large (76,000 billion, in our carnivore-herbivore example), they lacked the means actually to use them. 76,000 billion permutations may be too many even today, but we can obtain very good approximations of the p-values of permutation tests using simulation.

The idea is that we look at a large number of randomly selected permutations, and check for how many of them we obtain a test statistic that is more extreme than the sample test statistic. The law of large number guarantees that this proportion will converge to the permutation test p-value as the number of randomly selected permutations increases.

Let's have a go!

```
# Filter the data, to get carnivores and herbivores:
data <- subset(msleep, vore == "carni" | vore == "herbi")

# Compute the sample test statistic:
sample_t <- t.test(sleep_total ~ vore, data = data)$statistic

# Set the number of random permutations and create a vector to
# store the result in:
B <- 9999
permutation_t <- vector("numeric", B)

# Start progress bar:
pbar <- txtProgressBar(min = 0, max = B, style = 3)

# Compute the test statistic for B randomly selected permutations
for(i in 1:B)
{
      # Draw a permutation of the labels:
      data$vore <- sample(data$vore, length(data$vore),
                          replace = FALSE)

      # Compute statistic for permuted sample:
      permutation_t[i] <- t.test(sleep_total ~ vore,
                                 data = data)$statistic
```

```
      # Update progress bar
      setTxtProgressBar(pbar, i)
}
close(pbar)

# In this case, with a two-sided alternative hypothesis, a
# "more extreme" test statistic is one that has a larger
# absolute value than the sample test statistic.

# Compute approximate permutation test p-value:
mean(abs(permutation_t) > abs(sample_t))
```

In this particular example, the resulting p-value is pretty close to that from the old-school t-test. However, we will soon see examples where the two versions of the t-test differ more.

You may ask why we used 9,999 permutations and not 10,000. The reason is that we avoid p-values that are equal to traditional significance levels like 0.05 and 0.01 this way. If we'd used 10,000 permutations, 500 of which yielded a statistics that had a larger absolute value than the sample statistic, then the p-value would have been exactly 0.05, which would cause some difficulties in trying to determine whether or not the result was significant at the 5 % level. This cannot happen when we use 9,999 permutations instead (500 statistics with a large absolute value yields the p-value $0.050005 > 0.05$, and 499 yields the p-value $0.0499 < 0.05$).

Having to write a `for` loop every time we want to run a t-test seems unnecessarily complicated. Fortunately, others have tread this path before us. The `MKinfer` package contains a function to perform (approximate) permutation t-tests, which also happens to be faster than our implementation above. Let's install it:

```
install.packages("MKinfer")
```

The function for the permutation t-test, `perm.t.test`, works exactly like `t.test`. In all the examples from Section 7.2.1 we can replace `t.test` with `perm.t.test` to run a permutation t-test instead. Like so:

```
library(MKinfer)
perm.t.test(sleep_total ~ vore, data =
              subset(msleep, vore == "carni" | vore == "herbi"))
```

Note that two p-values and confidence intervals are presented: one set from the permutations and one from the old-school approach - so make sure that you look at the right ones!

You may ask how many randomly selected permutations we need to get an accurate approximation of the permutation test p-value. By default, `perm.t.test` uses 9,999

permutations (you can change that number using the argument `R`), which is widely considered to be a reasonable number. If you are running a permutation test with a much more complex (and computationally intensive) statistic, you may have to use a lower number, but avoid that if you can.

7.2.3 The bootstrap

A popular method for computing p-values and confidence intervals that resembles the permutation approach is the bootstrap. Instead of drawing permuted samples, new observations are drawn with replacement from the original sample, and then labels are randomly allocated to them. That means that each randomly drawn sample will differ not only in the permutation of labels, but also in what observations are included - some may appear more than once and some not at all.

We will have a closer look at the bootstrap in Section 7.7, where we will learn how to use it for creating confidence intervals and computing p-values for any test statistic. For now, we'll just note that `MKinfer` offers a bootstrap version of the t-test, `boot.t.test` :

```
library(MKinfer)
boot.t.test(sleep_total ~ vore, data =
              subset(msleep, vore == "carni" | vore == "herbi"))
```

Both `perm.test` and `boot.test` have a useful argument called `symmetric`, the details of which are discussed in depth in Section 12.3.

7.2.4 Saving the output

When we run a t-test, the results are printed in the Console. But we can also store the results in a variable, which allows us to access e.g. the p-value of the test:

```
library(ggplot2)
carnivores <- msleep[msleep$vore == "carni",]
herbivores <- msleep[msleep$vore == "herbi",]
test_result <- t.test(sleep_total ~ vore, data =
                  subset(msleep, vore == "carni" | vore == "herbi"))

test_result
```

What does the resulting object look like?

```
str(test_result)
```

As you can see, `test_result` is a `list` containing different parameters and vectors for the test. To get the p-value, we can run the following:

```
test_result$p.value
```

7.2.5 Multiple testing

Some programming tools from Section 6.4 can be of use if we wish to perform multiple t-tests. For example, maybe we want to make pairwise comparisons of the sleeping times of all the different feeding behaviours in `msleep`: carnivores, herbivores, insectivores and omnivores. To find all possible pairs, we can use a nested `for` loop (Section 6.4.2). Note how the indices `i` and `j` that we loop over are set so that we only run the test for each combination once:

```
library(MKinfer)

# List the different feeding behaviours (ignoring NA's):
vores <- na.omit(unique(msleep$vore))
B <- length(vores)

# Compute the number of pairs, and create an appropriately
# sized data frame to store the p-values in:
n_comb <- choose(B, 2)
p_values <- data.frame(group1 = vector("character", n_comb),
                       group2 = vector("character", n_comb),
                       p = vector("numeric", n_comb))

# Loop over all pairs:
k <- 1 # Counter variable
for(i in 1:(B-1))
{
      for(j in (i+1):B)
      {
            # Run a t-test for the current pair:
            test_res <- perm.t.test(sleep_total ~ vore,
                   data = subset(msleep,
                                 vore == vores[i] | vore == vores[j]))
            # Store the p-value:
            p_values[k, ] <- c(vores[i], vores[j], test_res$p.value)
            # Increase the counter variable:
            k <- k + 1
      }
}
```

To view the p-values for each pairwise test, we can now run:

```
p_values
```

When we run multiple tests, the risk for a type I error increases, to the point where we're virtually guaranteed to get a significant result. We can reduce the risk of false positive results and adjust the p-values for multiplicity using for instance Bonferroni correction, Holm's method (an improved version of the standard Bonferroni approach), or the Benjamini-Hochberg approach (which controls the *false discovery rate* and is useful if you for instance are screening a lot of variables for differences), using `p.adjust`:

```
p.adjust(p_values$p, method = "bonferroni")
p.adjust(p_values$p, method = "holm")
p.adjust(p_values$p, method = "BH")
```

7.2.6 Multivariate testing with Hotelling's T^2

If you are interested in comparing the means of several variables for two groups, using a multivariate test is sometimes a better option than running multiple univariate t-tests. The multivariate generalisation of the t-test, Hotelling's T^2, is available through the `Hotelling` package:

```
install.packages("Hotelling")
```

As an example, consider the `airquality` data. Let's say that we want to test whether the mean ozone, solar radiation, wind speed, and temperature differ between June and July. We could use four separate t-tests to test this, but we could also use Hotelling's T^2 to test the null hypothesis that the mean vector, i.e. the vector containing the four means, is the same for both months. The function used for this is `hotelling.test`:

```
# Subset the data:
airquality_t2 <- subset(airquality, Month == 6 | Month == 7)

# Run the test under the assumption of normality:
library(Hotelling)
t2 <- hotelling.test(Ozone + Solar.R + Wind + Temp ~ Month,
                     data = airquality_t2)
t2

# Run a permutation test instead:
t2 <- hotelling.test(Ozone + Solar.R + Wind + Temp ~ Month,
                     data = airquality_t2, perm = TRUE)
t2
```

7.2.7 Sample size computations for the t-test

In any study, it is important to collect enough data for the inference that we wish to make. If we want to use a t-test for a test about a mean or the difference of two

means, what constitutes "enough data" is usually measured by the power of the test. The sample is large enough when the test achieves high enough power. If we are comfortable assuming normality (and we may well be, especially as the main goal with sample size computations is to get a ballpark figure), we can use `power.t.test` to compute what power our test would achieve under different settings. For a two-sample test with unequal variances, we can use `power.welch.t.test` from `MKpower` instead. Both functions can be used to either find the sample size required for a certain power, or to find out what power will be obtained from a given sample size.

First of all, let's install `MKpower`:

```
install.packages("MKpower")
```

`power.t.test` and `power.welch.t.test` both use `delta` to denote the mean difference under the alternative hypothesis. In addition, we must supply the standard deviation`sd` of the distribution. Here are some examples:

```
library(MKpower)

# A one-sided one-sample test with 80 % power:
power.t.test(power = 0.8, delta = 1, sd = 1, sig.level = 0.05,
             type = "one.sample", alternative = "one.sided")

# A two-sided two-sample test with sample size n = 25 and equal
# variances:
power.t.test(n = 25, delta = 1, sd = 1, sig.level = 0.05,
             type = "two.sample", alternative = "two.sided")

# A one-sided two-sample test with 90 % power and equal variances:
power.t.test(power = 0.9, delta = 1, sd = 0.5, sig.level = 0.01,
             type = "two.sample", alternative = "one.sided")

# A one-sided two-sample test with 90 % power and unequal variances:
power.welch.t.test(power = 0.9, delta = 1, sd1 = 0.5, sd2 = 1,
                   sig.level = 0.01,
                   type = "two.sample", alternative = "one.sided")
```

You may wonder how to choose `delta` and `sd`. If possible, it is good to base these numbers on a pilot study or related previous work. If no such data is available, your guess is as good as mine. For `delta`, some useful terminology comes from medical statistics, where the concept of *clinical significance* is used increasingly often. Make sure that `delta` is large enough to be clinically significant, that is, large enough to actually matter in practice.

If we have reason to believe that the data follows a non-normal distribution, another option is to use simulation to compute the sample size that will be required. We'll

do just that in Section 7.6.

Exercise 7.6. Return to the one-sided t-test that you performed in Exercise 7.5. Assume that `delta` is 0.5 (i.e. that the true mean is 6.5) and that the standard deviation is 2. How large does the sample size n have to be for the power of the test to be 95 % at a 5 % significance level? What is the power of the test when the sample size is $n = 2,351$?

7.2.8 A Bayesian approach

The Bayesian paradigm differs in many ways from the frequentist approach that we use in the rest of this chapter. In Bayesian statistics, we first define a *prior distribution* for the parameters that we are interested in, representing our beliefs about them (for instance based on previous studies). Bayes' theorem is then used to derive the *posterior distribution*, i.e. the distribution of the coefficients given the prior distribution and the data. Philosophically, this is very different from frequentist estimation, in which we don't incorporate prior beliefs into our models (except for through which variables we include).

In many situations, we don't have access to data that can be used to create an *informative* prior distribution. In such cases, we can use a so-called weakly informative prior instead. These act as a sort of "default priors", representing large uncertainty about the values of the coefficients.

The `rstanarm` package contains methods for using Bayesian estimation to fit some common statistical models. It takes a while to install, but it is well worth it:

```
install.packages("rstanarm")
```

To use a Bayesian model with a weakly informative prior to analyse the difference in sleeping time between herbivores and carnivores, we load `rstanarm` and use `stan_glm` in complete analogue with how we use `t.test`:

```
library(rstanarm)
library(ggplot2)
m <- stan_glm(sleep_total ~ vore, data =
         subset(msleep, vore == "carni" | vore == "herbi"))

# Print the estimates:
m
```

There are two estimates here: an "intercept" (the average sleeping time for carnivores) and `voreherbi` (the difference between carnivores and herbivores). To plot the posterior distribution of the difference, we can use `plot`:

```
plot(m, "dens", pars = c("voreherbi"))
```

To get a 95 % credible interval (the Bayesian equivalent of a confidence interval) for the difference, we can use `posterior_interval` as follows:

```
posterior_interval(m,
        pars = c("voreherbi"),
        prob = 0.95)
```

p-values are not a part of Bayesian statistics, so don't expect any. It is however possible to perform a kind of Bayesian test of whether there is a difference by checking whether the credible interval for the difference contains 0. If not, there is evidence that there is a difference (Thulin, 2014c). In this case, 0 is contained in the interval, and there is no evidence of a difference.

In most cases, Bayesian estimation is done using Monte Carlo integration (specifically, a class of methods known as Markov Chain Monte Carlo, MCMC). To check that the model fitting has converged, we can use a measure called $\hat{R}$. It should be less than 1.1 if the fitting has converged:

```
plot(m, "rhat")
```

If the model fitting hasn't converged, you may need to increase the number of iterations of the MCMC algorithm. You can increase the number of iterations by adding the argument `iter` to `stan_glm` (the default is 2,000).

If you want to use a custom prior for your analysis, that is of course possible too. See `?priors` and `?stan_glm` for details about this, and about the default weakly informative prior.

7.3 Other common hypothesis tests and confidence intervals

There are thousands of statistical tests in addition to the t-test, and equally many methods for computing confidence intervals for different parameters. In this section we will have a look at some useful tools: the nonparametric Wilcoxon-Mann-Whitney test for location, tests for correlation, χ^2-tests for contingency tables, and confidence intervals for proportions.

7.3.1 Nonparametric tests of location

The Wilcoxon-Mann-Whitney test, `wilcox.test` in R, is a nonparametric alternative to the t-test that is based on ranks. `wilcox.test` can be used in complete analogue to `t.test`.

We can use two vectors as input:

```
library(ggplot2)
carnivores <- msleep[msleep$vore == "carni",]
herbivores <- msleep[msleep$vore == "herbi",]
wilcox.test(carnivores$sleep_total, herbivores$sleep_total)
```

Or use a formula:

```
wilcox.test(sleep_total ~ vore, data =
              subset(msleep, vore == "carni" | vore == "herbi"))
```

7.3.2 Tests for correlation

To test the null hypothesis that two numerical variables are correlated, we can use `cor.test`. Let's try it with sleeping times and brain weight, using the `msleep` data again:

```
library(ggplot2)
cor.test(msleep$sleep_total, msleep$brainwt,
         use = "pairwise.complete")
```

The setting `use = "pairwise.complete"` means that `NA` values are ignored.

`cor.test` doesn't have a `data` argument, so if you want to use it in a pipeline I recommend using the `%$%` pipe (Section 6.2) to pass on the vectors from your data frame:

```
library(magrittr)
msleep %$% cor.test(sleep_total, brainwt, use = "pairwise.complete")
```

The test we just performed uses the Pearson correlation coefficient as its test statistic. If you prefer, you can use the nonparametric Spearman and Kendall correlation coefficients in the test instead, by changing the value of `method`:

```
# Spearman test of correlation:
cor.test(msleep$sleep_total, msleep$brainwt,
         use = "pairwise.complete",
         method = "spearman")
```

These tests are all based on asymptotic approximations, which among other things causes the Pearson correlation test perform poorly for non-normal data. In Section 7.7 we will create a bootstrap version of the correlation test, which has better performance.

7.3.3 χ^2-tests

χ^2 (chi-squared) tests are most commonly used to test whether two categorical variables are independent. To use it, we must first construct a contingency table, i.e. a

table showing the counts for different combinations of categories, typically using `table`. Here is an example with the `diamonds` data from `ggplot2`:

```
library(ggplot2)
table(diamonds$cut, diamonds$color)
```

The null hypothesis of our test is that the quality of the cut (`cut`) and the colour of the diamond (`color`) are independent, with the alternative being that they are dependent. We use `chisq.test` with the contingency table as input to run the χ^2 test of independence:

```
chisq.test(table(diamonds$cut, diamonds$color))
```

By default, `chisq.test` uses an asymptotic approximation of the p-value. For small sample sizes, it is almost often better to use permutation p-values by setting `simulate.p.value = TRUE` (but here the sample is not small, and so the computation of the permutation test will take a while):

```
chisq.test(table(diamonds$cut, diamonds$color),
           simulate.p.value = TRUE)
```

As with `t.test`, we can use pipes to perform the test if we like:

```
library(magrittr)
diamonds %$% table(cut, color) %>%
      chisq.test()
```

If both of the variables are binary, i.e. only take two values, the power of the test can be approximated using `power.prop.test`. Let's say that we have two variables, X and Y, taking the values 0 and 1. Assume that we collect n observations with $X = 0$ and n with $X = 1$. Furthermore, let `p1` be the probability that $Y = 1$ if $X = 0$ and `p2` be the probability that $Y = 1$ if $X = 1$. We can then use `power.prop.test` as follows:

```
# Assume that n = 50, p1 = 0.4 and p2 = 0.5 and compute the power:
power.prop.test(n = 50, p1 = 0.4, p2 = 0.5, sig.level = 0.05)

# Assume that p1 = 0.4 and p2 = 0.5 and that we want 85 % power.
# To compute the sample size required:
power.prop.test(power = 0.85, p1 = 0.4, p2 = 0.5, sig.level = 0.05)
```

7.3.4 Confidence intervals for proportions

The different t-test functions provide confidence intervals for means and differences of means. But what about proportions? The `binomCI` function in the `MKinfer` package allows us to compute confidence intervals for proportions from binomial experiments using a number of methods. The input is the number of "successes" `x`, the sample

size `n`, and the `method` to be used.

Let's say that we want to compute a confidence interval for the proportion of herbivore mammals that sleep for more than 7 hours a day.

```
library(ggplot2)
herbivores <- msleep[msleep$vore == "herbi",]

# Compute the number of animals for which we know the sleeping time:
n <- sum(!is.na(herbivores$sleep_total))

# Compute the number of "successes", i.e. the number of animals
# that sleep for more than 7 hours:
x <- sum(herbivores$sleep_total > 7, na.rm = TRUE)
```

The estimated proportion is `x/n`, which in this case is 0.625. We'd like to quantify the uncertainty in this estimate by computing a confidence interval. The standard Wald method, taught in most introductory courses, can be computed using:

```
library(MKinfer)
binomCI(x, n, conf.level = 0.95, method = "wald")
```

Don't do that though! The Wald interval is known to be severely flawed (Brown et al., 2001), and much better options are available. If the proportion can be expected to be close to 0 or 1, the Clopper-Pearson interval is recommended, and otherwise the Wilson interval is the best choice (Thulin, 2014a):

```
binomCI(x, n, conf.level = 0.95, method = "clopper-pearson")
binomCI(x, n, conf.level = 0.95, method = "wilson")
```

An excellent Bayesian credible interval is the Jeffreys interval, which uses the weakly informative Jeffreys prior:

```
binomCI(x, n, conf.level = 0.95, method = "jeffreys")
```

The `ssize.propCI` function in `MKpower` can be used to compute the sample size needed to obtain a confidence interval with a given width[3]. It relies on asymptotic formulas that are highly accurate, as you later on will verify in Exercise 7.17.

```
library(MKpower)
# Compute the sample size required to obtain an interval with
# width 0.1 if the true proportion is 0.4:
ssize.propCI(prop = 0.4, width = 0.1, method = "wilson")
ssize.propCI(prop = 0.4, width = 0.1, method = "clopper-pearson")
```

[3]Or rather, a given *expected*, or average, width. The width of the interval is a function of a random variable, and is therefore also random.

~

Exercise 7.7. The function `binomDiffCI` from `MKinfer` can be used to compute a confidence interval for the *difference* of two proportions. Using the `msleep` data, use it to compute a confidence interval for the difference between the proportion of herbivores that sleep for more than 7 hours a day and the proportion of carnivores that sleep for more than 7 hours a day.

7.4 Ethical issues in statistical inference

The use and misuse of statistical inference offer many ethical dilemmas. Some common issues related to ethics and good statistical practice are discussed below. As you read them and work with the associated exercises, consider consulting the ASA's ethical guidelines, presented in Section 3.11.

7.4.1 p-hacking and the file-drawer problem

Hypothesis tests are easy to misuse. If you run enough tests on your data, you are almost guaranteed to end up with significant results - either due to chance or because some of the null hypotheses you test are false. The process of trying lots of different tests (different methods, different hypotheses, different sub-groups) in search of significant results is known as *p-hacking* or *data dredging*. This greatly increases the risk of false findings, and can often produce misleading results.

Many practitioners inadvertently resort to p-hacking, by mixing exploratory data analysis and hypothesis testing, or by coming up with new hypotheses to test as they work with their data. This can be avoided by planning your analyses in advance, a practice that in fact is required in medical trials.

On the other end of the spectrum, there is the *file-drawer problem*, in which studies with negative (i.e. not statistically significant) results aren't published or reported, but instead are stored in the researcher's file-drawers. There are many reasons for this, one being that negative results usually are seen as less important and less worthy of spending time on. Simply put, negative results just aren't news. If your study shows that eating kale every day significantly reduces the risk of cancer, then that is news, something that people are interested in learning, and something that can be published in a prestigious journal. However, if your study shows that a daily serving of kale has no impact on the risk of cancer, that's not news, people aren't really interested in hearing it, and it may prove difficult to publish your findings.

But what if 100 different researchers carried out the same study? If eating kale doesn't affect the risk of cancer, then we can still expect 5 out of these researchers to get significant results (using a 5 % significance level). If only those researchers publish their results, that may give the impressions that there is strong evidence of

the cancer-preventing effect of kale backed up by several papers, even though the majority of studies actually indicated that there was no such effect.

~

Exercise 7.8. *Discuss the following.* You are helping a research team with statistical analysis of data that they have collected. You agree on five hypotheses to test. None of the tests turns out significant. Fearing that all their hard work won't lead anywhere, your collaborators then ask you to carry out five new tests. Neither turns out significant. Your collaborators closely inspect the data and then ask you to carry out ten more tests, two of which are significant. The team wants to publish these significant results in a scientific journal. Should you agree to publish them? If so, what results should be published? Should you have put your foot down and told them not to run more tests? Does your answer depend on how long it took the research team to collect the data? What if the team won't get funding for new projects unless they publish a paper soon? What if other research teams competing for the same grants do their analyses like this?

Exercise 7.9. *Discuss the following.* You are working for a company that is launching a new product, a hair-loss treatment. In a small study, the product worked for 19 out of 22 participants (86 %). You compute a 95 % Clopper-Pearson confidence interval (Section 7.3.4) for the proportion of successes and find that it is (0.65, 0.97). Based on this, the company wants to market the product as being 97 % effective. Is that acceptable to you? If not, how should it be marketed? Would your answer change if the product was something else (new running shoes that make you faster, a plastic film that protects smartphone screens from scratches, or contraceptives)? What if the company wanted to market it as being 86 % effective instead?

Exercise 7.10. *Discuss the following.* You have worked long and hard on a project. In the end, to see if the project was a success, you run a hypothesis test to check if two variables are correlated. You find that they are not ($p = 0.15$). However, if you remove three outliers, the two variables are significantly correlated ($p = 0.03$). What should you do? Does your answer change if you only have to remove one outlier to get a significant result? If you have to remove ten outliers? 100 outliers? What if the p-value is 0.051 before removing the outliers and 0.049 after removing the outliers?

Exercise 7.11. *Discuss the following.* You are analysing data from an experiment to see if there is a difference between two treatments. You estimate[4] that given the sample size and the expected difference in treatment effects, the power of the test that you'll be using, i.e. the probability of rejecting the null hypothesis if it is false, is about 15 %. Should you carry out such an analysis? If not, how high does the power need to be for the analysis to be meaningful?

[4] We'll discuss methods for producing such estimates in Section 7.5.3.

7.4.2 Reproducibility

An analysis is *reproducible* if it can be reproduced by someone else. By producing reproducible analyses, we make it easier for others to scrutinise our work. We also make all the steps in the data analysis transparent. This can act as a safeguard against data fabrication and data dredging.

In order to make an analysis reproducible, we need to provide at least two things. First, *the data* - all unedited data files in their original format. This also includes *metadata* with information required to understand the data (e.g. codebooks explaining variable names and codes used for categorical variables). Second, *the computer code* used to prepare and analyse the data. This includes any wrangling and preliminary testing performed on the data.

As long as we save our data files and code, data wrangling and analyses in R are inherently reproducible, in contrast to the same tasks carried out in menu-based software such as Excel. However, if reports are created using a word processor, there is always a risk that something will be lost along the way. Perhaps numbers are copied by hand (which may introduce errors), or maybe the wrong version of a figure is pasted into the document. R Markdown (Section 4.1) is a great tool for creating completely reproducible reports, as it allows you to integrate R code for data wrangling, analyses, and graphics in your report-writing. This reduces the risk of manually inserting errors, and allows you to share your work with others easily.

~

Exercise 7.12. *Discuss the following.* You are working on a study at a small-town hospital. The data involves biomarker measurements for a number of patients, and you show that patients with a sexually transmittable disease have elevated levels of some of the biomarkers. The data also includes information about the patients: their names, ages, ZIP codes, heights, and weights. The research team wants to publish your results and make the analysis reproducible. Is it ethically acceptable to share all your data? Can you make the analysis reproducible without violating patient confidentiality?

7.5 Evaluating statistical methods using simulation

An important use of simulation is in the evaluation of statistical methods. In this section, we will see how simulation can be used to compare the performance of two estimators, as well as the type I error rate and power of hypothesis tests.

7.5.1 Comparing estimators

Let's say that we want to estimate the mean μ of a normal distribution. We could come up with several different estimators for μ:

- The sample mean $\bar{x}$,
- The sample median $\tilde{x}$,
- The average of the largest and smallest value in the sample: $\frac{x_{max}+x_{min}}{2}$.

In this particular case (under normality), statistical theory tells us that the sample mean is the best estimator[5]. But how much better is it, really? And what if we didn't know statistical theory - could we use simulation to find out which estimator to use?

To begin with, let's write a function that computes the estimate $\frac{x_{max}+x_{min}}{2}$:

```
max_min_avg <- function(x)
{
      return((max(x)+min(x))/2)
}
```

Next, we'll generate some data from a $N(0,1)$ distribution and compute the three estimates:

```
x <- rnorm(25)

x_mean <- mean(x)
x_median <- median(x)
x_mma <- max_min_avg(x)
x_mean; x_median; x_mma
```

As you can see, the estimates given by the different approaches differ, so clearly the choice of estimator matters. We can't determine which to use based on a single sample though. Instead, we typically compare the long-run properties of estimators, such as their *bias* and *variance*. The bias is the difference between the mean of the estimator and the parameter it seeks to estimate. An estimator is *unbiased* if its bias is 0, which is considered desirable at least in this setting. Among unbiased estimators, we prefer the one that has the smallest variance. So how can we use simulation to compute the bias and variance of estimators?

The key to using simulation here is to realise that `x_mean` is an observation of the random variable $\bar{X} = \frac{1}{25}(X_1 + X_2 + \cdots + X_{25})$ where each X_i is $N(0,1)$-distributed. We can generate observations of X_i (using `rnorm`), and can therefore also generate observations of $\bar{X}$. That means that we can obtain an arbitrarily large sample of observations of $\bar{X}$, which we can use to estimate its mean and variance. Here is an example:

```
# Set the parameters for the normal distribution:
mu <- 0
sigma <- 1
```

[5] At least in terms of mean squared error.

```
# We will generate 10,000 observations of the estimators:
B <- 1e4
res <- data.frame(x_mean = vector("numeric", B),
                  x_median = vector("numeric", B),
                  x_mma = vector("numeric", B))

# Start progress bar:
pbar <- txtProgressBar(min = 0, max = B, style = 3)

for(i in seq_along(res$x_mean))
{
      x <- rnorm(25, mu, sigma)
      res$x_mean[i] <- mean(x)
      res$x_median[i] <- median(x)
      res$x_mma[i] <- max_min_avg(x)

      # Update progress bar
      setTxtProgressBar(pbar, i)
}
close(pbar)

# Compare the estimators:
colMeans(res-mu) # Bias
apply(res, 2, var) # Variances
```

All three estimators appear to be unbiased (even if the simulation results aren't exactly 0, they are very close). The sample mean has the smallest variance (and is therefore preferable!), followed by the median. The $\frac{x_{max}+x_{min}}{2}$ estimator has the worst performance, which is unsurprising as it ignores all information not contained in the extremes of the dataset.

In Section 7.5.5 we'll discuss how to choose the number of simulated samples to use in your simulations. For now, we'll just note that the estimate of the estimators' biases becomes more stable as the number of simulated samples increases, as can be seen from this plot, which utilises `cumsum`, described in Section 5.3.3:

```
# Compute estimates of the bias of the sample mean for each
# iteration:
res$iterations <- 1:B
res$x_mean_bias <- cumsum(res$x_mean)/1:B - mu

# Plot the results:
library(ggplot2)
ggplot(res, aes(iterations, x_mean_bias)) +
      geom_line() +
```

```
      xlab("Number of iterations") +
      ylab("Estimated bias")

# Cut the x-axis to better see the oscillations for smaller
# numbers of iterations:
ggplot(res, aes(iterations, x_mean_bias)) +
      geom_line() +
      xlab("Number of iterations") +
      ylab("Estimated bias") +
      xlim(0, 1000)
```

∼

Exercise 7.13. Repeat the above simulation for different samples sizes n between 10 and 100. Plot the resulting variances as a function of n.

Exercise 7.14. Repeat the simulation in 7.13, but with a $t(3)$ distribution instead of the normal distribution. Which estimator is better in this case?

7.5.2 Type I error rate of hypothesis tests

In the same vein that we just compared estimators, we can also compare hypothesis tests or confidence intervals. Let's have a look at the former, and evaluate how well the old-school two-sample t-test fares compared to a permutation t-test and the Wilcoxon-Mann-Whitney test.

For our first comparison, we will compare the type I error rate of the three tests, i.e. the risk of rejecting the null hypothesis if the null hypothesis is true. Nominally, this is the significance level α, which we set to be 0.05.

We write a function for such a simulation, to which we can pass the sizes `n1` and `n2` of the two samples, as well as a function `distr` to generate data:

```
# Load package used for permutation t-test:
library(MKinfer)

# Create a function for running the simulation:
simulate_type_I <- function(n1, n2, distr, level = 0.05, B = 999,
                            alternative = "two.sided", ...)
{
      # Create a data frame to store the results in:
      p_values <- data.frame(p_t_test = vector("numeric", B),
                             p_perm_t_test = vector("numeric", B),
                             p_wilcoxon = vector("numeric", B))
```

```
    # Start progress bar:
    pbar <- txtProgressBar(min = 0, max = B, style = 3)

    for(i in 1:B)
    {
        # Generate data:
        x <- distr(n1, ...)
        y <- distr(n2, ...)

        # Compute p-values:
        p_values[i, 1] <- t.test(x, y,
                                 alternative = alternative)$p.value
        p_values[i, 2] <- perm.t.test(x, y,
                                 alternative = alternative,
                                 R = 999)$perm.p.value
        p_values[i, 3] <- wilcox.test(x, y,
                                 alternative = alternative)$p.value

        # Update progress bar:
        setTxtProgressBar(pbar, i)
    }

    close(pbar)

    # Return the type I error rates:
    return(colMeans(p_values < level))
}
```

First, let's try it with normal data. The simulation takes a little while to run, primarily because of the permutation t-test, so you may want to take a short break while you wait.

```
simulate_type_I(20, 20, rnorm, B = 9999)
```

Next, let's try it with a lognormal distribution, both with balanced and imbalanced sample sizes. Increasing the parameter σ (`sdlog`) increases the skewness of the lognormal distribution (i.e. makes it *more* asymmetric and therefore less similar to the normal distribution), so let's try that to. In case you are in a rush, the results from my run of this code block can be found below it.

```
simulate_type_I(20, 20, rlnorm, B = 9999, sdlog = 1)
simulate_type_I(20, 20, rlnorm, B = 9999, sdlog = 3)
simulate_type_I(20, 30, rlnorm, B = 9999, sdlog = 1)
simulate_type_I(20, 30, rlnorm, B = 9999, sdlog = 3)
```

My results were:

```
# Normal distribution, n1 = n2 = 20:
     p_t_test p_perm_t_test    p_wilcoxon
   0.04760476    0.04780478    0.04680468

# Lognormal distribution, n1 = n2 = 20, sigma = 1:
     p_t_test p_perm_t_test    p_wilcoxon
   0.03320332    0.04620462    0.04910491

# Lognormal distribution, n1 = n2 = 20, sigma = 3:
     p_t_test p_perm_t_test    p_wilcoxon
   0.00830083    0.05240524    0.04590459

# Lognormal distribution, n1 = 20, n2 = 30, sigma = 1:
     p_t_test p_perm_t_test    p_wilcoxon
   0.04080408    0.04970497    0.05300530

# Lognormal distribution, n1 = 20, n2 = 30, sigma = 3:
     p_t_test p_perm_t_test    p_wilcoxon
   0.01180118    0.04850485    0.05240524
```

What's noticeable here is that the permutation t-test and the Wilcoxon-Mann-Whitney test have type I error rates that are close to the nominal 0.05 in all five scenarios, whereas the t-test has too low a type I error rate when the data comes from a lognormal distribution. This makes the test too conservative in this setting. Next, let's compare the power of the tests.

7.5.3 Power of hypothesis tests

The power of a test is the probability of rejecting the null hypothesis if it is false. To estimate that, we need to generate data under the alternative hypothesis. For two-sample tests of the mean, the code is similar to what we used for the type I error simulation above, but we now need two functions for generating data - one for each group, because the groups differ under the alternative hypothesis. Bear in mind that the alternative hypothesis for the two-sample test is that the two distributions differ in location, so the two functions for generating data should reflect that.

```
# Load package used for permutation t-test:
library(MKinfer)

# Create a function for running the simulation:
simulate_power <- function(n1, n2, distr1, distr2, level = 0.05,
                           B = 999, alternative = "two.sided")
{
      # Create a data frame to store the results in:
```

```
      p_values <- data.frame(p_t_test = vector("numeric", B),
                             p_perm_t_test = vector("numeric", B),
                             p_wilcoxon = vector("numeric", B))

      # Start progress bar:
      pbar <- txtProgressBar(min = 0, max = B, style = 3)

      for(i in 1:B)
      {
            # Generate data:
            x <- distr1(n1)
            y <- distr2(n2)

            # Compute p-values:
            p_values[i, 1] <- t.test(x, y,
                                     alternative = alternative)$p.value
            p_values[i, 2] <- perm.t.test(x, y,
                                     alternative = alternative,
                                     R = 999)$perm.p.value
            p_values[i, 3] <- wilcox.test(x, y,
                                     alternative = alternative)$p.value

            # Update progress bar:
            setTxtProgressBar(pbar, i)
      }

      close(pbar)

      # Return power:
      return(colMeans(p_values < level))
}
```

Let's try this out with lognormal data, where the difference in the log means is 1:

```
# Balanced sample sizes:
simulate_power(20, 20, function(n) { rlnorm(n,
                                            meanlog = 2, sdlog = 1) },
                       function(n) { rlnorm(n,
                                             meanlog = 1, sdlog = 1) },
                       B = 9999)

# Imbalanced sample sizes:
simulate_power(20, 30, function(n) { rlnorm(n,
                                            meanlog = 2, sdlog = 1) },
```

```
                          function(n) { rlnorm(n,
                                                meanlog = 1, sdlog = 1) },
                          B = 9999)
```

Here are the results from my runs:

```
# Balanced sample sizes:
     p_t_test p_perm_t_test    p_wilcoxon
    0.6708671     0.7596760     0.8508851

# Imbalanced sample sizes:
     p_t_test p_perm_t_test    p_wilcoxon
    0.6915692     0.7747775     0.9041904
```

Among the three, the Wilcoxon-Mann-Whitney test appears to be preferable for lognormal data, as it manages to obtain the correct type I error rate (unlike the old-school t-test) and has the highest power (although we would have to consider more scenarios, including different samples sizes, other differences of means, and different values of σ to say for sure!).

Remember that both our estimates of power and type I error rates are proportions, meaning that we can use binomial confidence intervals to quantify the uncertainty in the estimates from our simulation studies. Let's do that for the lognormal setting with balanced sample sizes, using the results from my runs. The number of simulated samples were 9,999. For the t-test, the estimated type I error rate was `0.03320332`, which corresponds to $0.03320332 \cdot 9,999 = 332$ "successes". Similarly, there were 6,708 "successes" in the power study. The confidence intervals become:

```
library(MKinfer)
binomCI(332, 9999, conf.level = 0.95, method = "clopper-pearson")
binomCI(6708, 9999, conf.level = 0.95, method = "wilson")
```

~

Exercise 7.15. Repeat the simulation study of type I error rate and power for the old school t-test, permutation t-test and the Wilcoxon-Mann-Whitney test with $t(3)$-distributed data. Which test has the best performance? How much lower is the type I error rate of the old-school t-test compared to the permutation t-test in the case of balanced sample sizes?

7.5.4 Power of some tests of location

The `MKpower` package contains functions for quickly performing power simulations for the old-school t-test and Wilcoxon-Mann-Whitney test in different settings. The

arguments `rx` and `ry` are used to pass functions used to generate the random numbers, in line with the `simulate_power` function that we created above.

For the t-test, we can use `sim.power.t.test`:

```
library(MKpower)
sim.power.t.test(nx = 25, rx = rnorm, rx.H0 = rnorm,
                 ny = 25, ry = function(x) { rnorm(x, mean = 0.8) },
                 ry.H0 = rnorm)
```

For the Wilcoxon-Mann-Whitney test, we can use `sim.power.wilcox.test` for power simulations:

```
library(MKpower)
sim.power.wilcox.test(nx = 10, rx = rnorm, rx.H0 = rnorm,
                      ny = 15,
                      ry = function(x) { rnorm(x, mean = 2) },
                      ry.H0 = rnorm)
```

7.5.5 Some advice on simulation studies

There are two things that you need to decide when performing a simulation study:

- How many *scenarios* to include, i.e. how many different settings for the model parameters to study, and
- How many *iterations* to use, i.e. how many simulated samples to create for each scenario.

The number of scenarios is typically determined by what the purpose of the study is. If you only are looking to compare two tests for a particular sample size and a particular difference in means, then maybe you only need that one scenario. On the other hand, if you want to know which of the two tests that is preferable in general, or for different sample sizes, or for different types of distributions, then you need to cover more scenarios. In that case, the number of scenarios may well be determined by how much time you have available or how many you can fit into your report.

As for the number of iterations to run, that also partially comes down to computational power. If each iteration takes a long while to run, it may not be feasible to run tens of thousands of iterations (some advice for speeding up simulations by using parallelisation can be found in Section 10.2). In the best of all possible worlds, you have enough computational power available, and can choose the number of iterations freely. In such cases, it is often a good idea to use confidence intervals to quantify the uncertainty in your estimate of power, bias, or whatever it is that you are studying. For instance, the power of a test is estimated as the proportion of simulations in which the null hypothesis was rejected. This is a binomial experiment, and a confidence interval for the power can be obtained using the methods described in Section 7.3.4. Moreover, the `ssize.propCI` function described in said section can be used to

determine the number of simulations that you need to obtain a confidence interval that is short enough for you to feel that you have a good idea about the actual power of the test.

As an example, if a small pilot simulation indicates that the power is about 0.8 and you want a confidence interval with width 0.01, the number of simulations needed can be computed as follows:

```
library(MKpower)
ssize.propCI(prop = 0.8, width = 0.01, method = "wilson")
```

In this case, you'd need 24,592 iterations to obtain the desired accuracy.

7.6 Sample size computations using simulation

Using simulation to compare statistical methods is a key tool in methodological statistical research and when assessing new methods. In applied statistics, a use of simulation that is just as important is sample size computations. In this section we'll have a look at how simulations can be useful in determining sample sizes.

7.6.1 Writing your own simulation

Suppose that we want to perform a correlation test and want to know how many observations we need to collect. As in the previous section, we can write a function to compute the power of the test:

```
simulate_power <- function(n, distr, level = 0.05, B = 999, ...)
{
      p_values <- vector("numeric", B)

      # Start progress bar:
      pbar <- txtProgressBar(min = 0, max = B, style = 3)

      for(i in 1:B)
      {
            # Generate bivariate data:
            x <- distr(n)

            # Compute p-values:
            p_values[i] <- cor.test(x[,1], x[,2], ...)$p.value

            # Update progress bar:
            setTxtProgressBar(pbar, i)
      }
```

```
    close(pbar)

    return(mean(p_values < level))
}
```

Under the null hypothesis of no correlation, the correlation coefficient is 0. We want to find a sample size that will give us 90 % power at the 5 % significance level, for different hypothesised correlations. We will generate data from a bivariate normal distribution, because it allows us to easily set the correlation of the generated data. Note that the mean and variance of the marginal normal distributions are nuisance variables, which can be set to 0 and 1, respectively, without loss of generality (because the correlation test is invariant under scaling and shifts in location).

First, let's try our power simulation function:

```
library(MASS) # Contains mvrnorm function for generating data
rho <- 0.5 # The correlation between the variables
mu <- c(0, 0)
Sigma <- matrix(c(1, rho, rho, 1), 2, 2)

simulate_power(50, function(n) { mvrnorm(n, mu, Sigma) }, B = 999)
```

To find the sample size we need, we will write a new function containing a `while` loop (see Section 6.4.5), that performs the simulation for increasing values of n until the test has achieved the desired power:

```
library(MASS)

power.cor.test <- function(n_start = 10, rho, n_incr = 5, power = 0.9,
                           B = 999, ...)
{
    # Set parameters for the multivariate normal distribution:
    mu <- c(0, 0)
    Sigma <- matrix(c(1, rho, rho, 1), 2, 2)

    # Set initial values
    n <- n_start
    power_cor <- 0

    # Check power for different sample sizes:
    while(power_cor < power)
    {
        power_cor <- simulate_power(n,
                                    function(n) { mvrnorm(n, mu, Sigma) },
                                    B = B, ...)
```

```
        cat("n =", n, " - Power:", power_cor, "\n")
        n <- n + n_incr
    }

    # Return the result:
    cat("\nWhen n =", n, "the power is", round(power_cor, 2), "\n")
    return(n)
}
```

Let's try it out with different settings:

```
power.cor.test(n_start = 10, rho = 0.5, power = 0.9)
power.cor.test(n_start = 10, rho = 0.2, power = 0.8)
```

As expected, larger sample sizes are required to detect smaller correlations.

7.6.2 The Wilcoxon-Mann-Whitney test

The `sim.ssize.wilcox.test` in `MKpower` can be used to quickly perform sample size computations for the Wilcoxon-Mann-Whitney test, analogously to how we used `sim.power.wilcox.test` in Section 7.5.4:

```
library(MKpower)
sim.ssize.wilcox.test(rx = rnorm, ry = function(x) rnorm(x, mean = 2),
                      power = 0.8, n.min = 3, n.max = 10,
                      step.size = 1)
```

~

Exercise 7.16. Modify the functions we used to compute the sample sizes for the Pearson correlation test to instead compute sample sizes for the Spearman correlation tests. For bivariate normal data, are the required sample sizes lower or higher than those of the Pearson correlation test?

Exercise 7.17. In Section 7.3.4 we had a look at some confidence intervals for proportions, and saw how `ssize.propCI` can be used to compute sample sizes for such intervals using asymptotic approximations. Write a function to compute the exact sample size needed for the Clopper-Pearson interval to achieve a desired expected (average) width. Compare your results to those from the asymptotic approximations. Are the approximations good enough to be useful?

7.7 Bootstrapping

The bootstrap can be used formany things, most notably for constructing confidence intervals and running hypothesis tests. These tend to perform better than traditional parametric methods, such as the old-school t-test and its associated confidence interval, when the distributional assumptions of the parametric methods aren't met.

Confidence intervals and hypothesis tests are always based on a *statistic*, i.e. a quantity that we compute from the samples. The statistic could be the sample mean, a proportion, the Pearson correlation coefficient, or something else. In traditional parametric methods, we start by assuming that our data follows some distribution. For different reasons, including mathematical tractability, a common assumption is that the data is normally distributed. Under that assumption, we can then derive the distribution of the statistic that we are interested in analytically, like Gosset did for the t-test. That distribution can then be used to compute confidence intervals and p-values.

When using a bootstrap method, we follow the same steps, but use the observed data and simulation instead. Rather than making assumptions about the distribution[6], we use the empirical distribution of the data. Instead of analytically deriving a formula that describes the statistic's distribution, we find a good approximation of the distribution of the statistic by using simulation. We can then use that distribution to obtain confidence intervals and p-values, just as in the parametric case.

The simulation step is important. We use a process known as *resampling*, where we repeatedly draw new observations *with replacement* from the original sample. We draw B samples this way, each with the same size n as the original sample. Each randomly drawn sample - called a *bootstrap sample* - will include different observations. Some observations from the original sample may appear more than once in a specific bootstrap sample, and some not at all. For each bootstrap sample, we compute the statistic in which we are interested. This gives us B observations of this statistic, which together form what is called the *bootstrap distribution* of the statistic. I recommend using $B = 9,999$ or greater, but we'll use smaller B in some examples, to speed up the computations.

7.7.1 A general approach

The Pearson correlation test is known to be sensitive to deviations from normality. We can construct a more robust version of it using the bootstrap. To illustrate the procedure, we will use the `sleep_total` and `brainwt` variables from the `msleep` data. Here is the result from the traditional parametric Pearson correlation test:

```
library(ggplot2)
```

[6]Well, sometimes we make assumptions about the distribution *and* use the bootstrap. This is known as the parametric bootstrap, and is discussed in Section 7.7.4.

```
msleep %$% cor.test(sleep_total, brainwt, use = "pairwise.complete")
```

To find the bootstrap distribution of the Pearson correlation coefficient, we can use resampling with a `for` loop (Section 6.4.1):

```
# Extract the data that we are interested in:
mydata <- na.omit(msleep[,c("sleep_total", "brainwt")])

# Resampling using a for loop:
B <- 999 # Number of bootstrap samples
statistic <- vector("numeric", B)
for(i in 1:B)
{
      # Draw row numbers for the bootstrap sample:
      row_numbers <- sample(1:nrow(mydata), nrow(mydata),
                            replace = TRUE)

      # Obtain the bootstrap sample:
      sample <- mydata[row_numbers,]

      # Compute the statistic for the bootstrap sample:
      statistic[i] <- cor(sample[, 1], sample[, 2])
}

# Plot the bootstrap distribution of the statistic:
ggplot(data.frame(statistic), aes(statistic)) +
         geom_histogram(colour = "black")
```

Because this is such a common procedure, there are R packages that let's us do resampling without having to write a `for` loop. In the remainder of the section, we will use the `boot` package to draw bootstrap samples. It also contains convenience functions that allows us to get confidence intervals from the bootstrap distribution quickly. Let's install it:

```
install.packages("boot")
```

The most important function in this package is `boot`, which does the resampling. As input, it takes the original data, the number B of bootstrap samples to draw (called `R` here), and a function that computes the statistic of interest. This function should take the original data (`mydata` in our example above) and the row numbers of the sampled observation for a particular bootstrap sample (`row_numbers` in our example) as input.

For the correlation coefficient, the function that we input can look like this:

```
cor_boot <- function(data, row_numbers, method = "pearson")
{
    # Obtain the bootstrap sample:
    sample <- data[row_numbers,]

    # Compute and return the statistic for the bootstrap sample:
    return(cor(sample[, 1], sample[, 2], method = method))
}
```

To get the bootstrap distribution of the Pearson correlation coefficient for our data, we can now use `boot` as follows:

```
library(boot)

# Base solution:
boot_res <- boot(na.omit(msleep[,c("sleep_total", "brainwt")]),
                 cor_boot,
                 999)
```

Next, we can plot the bootstrap distribution of the statistic computed in `cor_boot`:

```
plot(boot_res)
```

If you prefer, you can of course use a pipeline for the resampling instead:

```
library(boot)
library(dplyr)

# With pipes:
msleep %>% select(sleep_total, brainwt) %>%
      drop_na %>%
      boot(cor_boot, 999) -> boot_res
```

7.7.2 Bootstrap confidence intervals

The next step is to use `boot.ci` to compute bootstrap confidence intervals. This is as simple as running:

```
boot.ci(boot_res)
```

Four intervals are presented: normal, basic, percentile and BCa. The details concerning how these are computed based on the bootstrap distribution are presented in Section 12.1. It is generally agreed that the percentile and BCa intervals are preferable to the normal and basic intervals; see e.g. Davison & Hinkley (1997) and Hall (1992); but which performs the best varies.

We also receive a warning message:

```
Warning message:
In boot.ci(boot_res) : bootstrap variances needed for studentized
intervals
```

A fifth type of confidence interval, the studentised interval, requires bootstrap estimates of the standard error of the test statistic. These are obtained by running an *inner bootstrap*, i.e. by bootstrapping each bootstrap sample to get estimates of the variance of the test statistic. Let's create a new function that does this, and then compute the bootstrap confidence intervals:

```
cor_boot_student <- function(data, i, method = "pearson")
{
    sample <- data[i,]

    correlation <- cor(sample[, 1], sample[, 2], method = method)

    inner_boot <- boot(sample, cor_boot, 100)
    variance <- var(inner_boot$t)

    return(c(correlation, variance))
}

library(ggplot2)
library(boot)

boot_res <- boot(na.omit(msleep[,c("sleep_total", "brainwt")]),
                 cor_boot_student,
                 999)

# Show bootstrap distribution:
plot(boot_res)

# Compute confidence intervals - including studentised:
boot.ci(boot_res)
```

While theoretically appealing (Hall, 1992), studentised intervals can be a little erratic in practice. I prefer to use percentile and BCa intervals instead.

For two-sample problems, we need to make sure that the number of observations drawn from each sample is the same as in the original data. The `strata` argument in `boot` is used to achieve this. Let's return to the example studied in Section 7.2, concerning the difference in how long carnivores and herbivores sleep. Let's say that we want a confidence interval for the difference of two means, using the `msleep` data. The simplest approach is to create a Welch-type interval, where we allow the two populations to have different variances. We can then resample from each population

separately:

```
# Function that computes the mean for each group:
mean_diff_msleep <- function(data, i)
{
    sample1 <- subset(data[i, 1], data[i, 2] == "carni")
    sample2 <- subset(data[i, 1], data[i, 2] == "herbi")
    return(mean(sample1[[1]]) - mean(sample2[[1]]))
}

library(ggplot2) # Load the data
library(boot)    # Load bootstrap functions

# Create the data set to resample from:
boot_data <- na.omit(subset(msleep,
              vore == "carni" | vore == "herbi")[,c("sleep_total",
                                                    "vore")])
# Do the resampling - we specify that we want resampling from two
# populations by using strata:
boot_res <- boot(boot_data,
                 mean_diff_msleep,
                 999,
                 strata = factor(boot_data$vore))

# Compute confidence intervals:
boot.ci(boot_res, type = c("perc", "bca"))
```

~

Exercise 7.18. Let's continue the example with a confidence interval for the difference in how long carnivores and herbivores sleep. How can you create a confidence interval under the assumption that the two groups have equal variances?

7.7.3 Bootstrap hypothesis tests

Writing code for bootstrap hypothesis tests can be a little tricky, because the resampling must be done *under the null hypothesis*. The process is greatly simplified by computing p-values using *confidence interval inversion* instead. This approach exploits the equivalence between confidence intervals and hypothesis tests, detailed in Section 12.2. It relies on the fact that:

- The p-value of the test for the parameter θ is the smallest α such that θ is not contained in the corresponding $1 - \alpha$ confidence interval.

- For a test for the parameter θ with significance level α, the set of values of θ that aren't rejected by the test (when used as the null hypothesis) is a $1-\alpha$ confidence interval for θ.

Here is an example of how we can use a `while` loop (Section 6.4.5) for confidence interval inversion, in order to test the null hypothesis that the Pearson correlation between sleeping time and brain weight is $\rho = -0.2$. It uses the studentised confidence interval that we created in the previous section:

```
# Compute the studentised confidence interval:
cor_boot_student <- function(data, i, method = "pearson")
{
    sample <- data[i,]

    correlation <- cor(sample[, 1], sample[, 2], method = method)

    inner_boot <- boot(sample, cor_boot, 100)
    variance <- var(inner_boot$t)

    return(c(correlation, variance))
}

library(ggplot2)
library(boot)

boot_res <- boot(na.omit(msleep[,c("sleep_total", "brainwt")]),
                 cor_boot_student,
                 999)

# Now, a hypothesis test:
# The null hypothesis:
rho_null <- -0.2

# Set initial conditions:
in_interval <- TRUE
alpha <- 0

# Find the lowest alpha for which rho_null is in the interval:
while(in_interval)
{
    # Increase alpha a small step:
    alpha <- alpha + 0.001

    # Compute the 1-alpha confidence interval, and extract
    # its bounds:
```

```
    interval <- boot.ci(boot_res,
                        conf = 1 - alpha,
                        type = "stud")$student[4:5]

    # Check if the null value for rho is greater than the lower
    # interval bound and smaller than the upper interval bound,
    # i.e. if it is contained in the interval:
    in_interval <- rho_null > interval[1] & rho_null < interval[2]
}
# The loop will finish as soon as it reaches a value of alpha such
# that rho_null is not contained in the interval.

# Print the p-value:
alpha
```

The `boot.pval` package contains a function computing p-values through inversion of bootstrap confidence intervals. We can use it to obtain a bootstrap p-value without having to write a `while` loop. It works more or less analogously to `boot.ci`. The arguments to the `boot.pval` function is the `boot` object (`boot_res`), the type of interval to use (`"stud"`), and the value of the parameter under the null hypothesis (`-0.2`):

```
install.packages("boot.pval")
library(boot.pval)
boot.pval(boot_res, type = "stud", theta_null = -0.2)
```

Confidence interval inversion fails in spectacular ways for certain tests for parameters of discrete distributions (Thulin & Zwanzig, 2017), so be careful if you plan on using this approach with count data.

~

Exercise 7.19. With the data from Exercise 7.18, invert a percentile confidence interval to compute the p-value of the corresponding test of the null hypothesis that there is no difference in means. What are the results?

7.7.4 The parametric bootstrap

In some cases, we may be willing to make distributional assumptions about our data. We can then use the *parametric bootstrap*, in which the resampling is done not from the original sample, but the theorised distribution (with parameters estimated from the original sample). Here is an example for the bootstrap correlation test, where we assume a multivariate normal distribution for the data. Note that we no longer

include an index as an argument in the function `cor_boot`, because the bootstrap samples won't be drawn directly from the original data:

```
cor_boot <- function(data, method = "pearson")
{
    return(cor(data[, 1], data[, 2], method = method))
}

library(MASS)
generate_data <- function(data, mle)
{
    return(mvrnorm(nrow(data), mle[[1]], mle[[2]]))
}

library(ggplot2)
library(boot)

filtered_data <- na.omit(msleep[,c("sleep_total", "brainwt")])
boot_res <- boot(filtered_data,
                 cor_boot,
                 R = 999,
                 sim = "parametric",
                 ran.gen = generate_data,
                 mle = list(colMeans(filtered_data),
                            cov(filtered_data)))

# Show bootstrap distribution:
plot(boot_res)

# Compute bootstrap percentile confidence interval:
boot.ci(boot_res, type = "perc")
```

The BCa interval implemented in `boot.ci` is not valid for parametric bootstrap samples, so running `boot.ci(boot_res)` without specifying the interval `type` will render an error[7]. Percentile intervals work just fine, though.

7.8 Reporting statistical results

Carrying out a statistical analysis is only the first step. After that, you probably need to communicate your results to others: your boss, your colleagues, your clients, the public… This section contains some tips for how best to do that.

[7]If you really need a BCa interval for the parametric bootstrap, you can find the formulas for it in Davison & Hinkley (1997).

7.8.1 What should you include?

When reporting your results, it should always be clear:

- How the data was collected,
- If, how, and why any observations were removed from the data prior to the analysis,
- What method was used for the analysis (including a reference unless it is a routine method),
- If any other analyses were performed/attempted on the data, and if you don't report their results, why.

Let's say that you've estimate some parameter, for instance the mean sleeping time of mammals, and want to report the results. The first thing to think about is that you shouldn't include too many decimals: don't give the mean with 5 decimals if sleeping times only were measured with one decimal.

> **BAD:** The mean sleeping time of mammals was found to be 10.43373.
>
> **GOOD:** The mean sleeping time of mammals was found to be 10.4.

It is common to see estimates reported with standard errors or standard deviations:

> **BAD:** The mean sleeping time of mammals was found to be 10.3 ($\sigma = 4.5$).

or

> **BAD:** The mean sleeping time of mammals was found to be 10.3 (standard error 0.49).

or

> **BAD:** The mean sleeping time of mammals was found to be 10.3 ± 0.49.

Although common, this isn't a very good practice. Standard errors/deviations are included to give some indication of the uncertainty of the estimate, but are very difficult to interpret. In most cases, they will probably cause the reader to either overestimate or underestimate the uncertainty in your estimate. A much better option is to present the estimate with a confidence interval, which quantifies the uncertainty in the estimate in an interpretable manner:

> **GOOD:** The mean sleeping time of mammals was found to be 10.3 (95 % percentile bootstrap confidence interval: 9.5-11.4).

Similarly, it is common to include error bars representing standard deviations and standard errors e.g. in bar charts. This questionable practice becomes even more troublesome because a lot of people fail to indicate what the error bars represent. If you wish to include error bars in your figures, they should always represent confidence intervals, unless you have a very strong reason for them to represent something else. In the latter case, make sure that you clearly explain what the error bars represent.

If the purpose of your study is to describe differences between groups, you should present a confidence interval for the difference between the groups, rather than one confidence interval (or error bar) for each group. It is possible for the individual confidence intervals to overlap even if there is a significant difference between the two groups, so reporting group-wise confidence intervals will only lead to confusion. If you are interested in the difference, then of course *the difference* is what you should report a confidence interval for.

> **BAD:** There was no significant difference between the sleeping times of carnivores (mean 10.4, 95 % percentile bootstrap confidence interval: 8.4-12.5) and herbivores (mean 9.5, 95 % percentile bootstrap confidence interval: 8.1-12.6).
>
> **GOOD:** There was no significant difference between the sleeping times of carnivores (mean 10.4) and herbivores (mean 9.5), with the 95 % percentile bootstrap confidence interval for the difference being (-1.8, 3.5).

7.8.2 Citing R packages

In statistical reports, it is often a good idea to specify what version of a software or a package that you used, for the sake of reproducibility (indeed, this is a requirement in some scientific journals). To get the citation information for the version of R that you are running, simply type `citation()`. To get the version number, you can use `R.Version` as follows:

```
citation()
R.Version()$version.string
```

To get the citation and version information for a package, use `citation` and `packageVersion` as follows:

```
citation("ggplot2")
packageVersion("ggplot2")
```

Chapter 8

Regression models

Regression models, in which explanatory variables are used to model the behaviour of a response variable, is without a doubt the most commonly used class of models in the statistical toolbox. In this chapter, we will have a look at different types of regression models tailored to many different sorts of data and applications.

After reading this chapter, you will be able to use R to:

- Fit and evaluate linear and generalised linear models,
- Fit and evaluate mixed models,
- Fit survival analysis models,
- Analyse data with left-censored observations,
- Create matched samples.

8.1 Linear models

Being flexible enough to handle different types of data, yet simple enough to be useful and interpretable, linear models are among the most important tools in the statistics toolbox. In this section, we'll discuss how to fit and evaluate linear models in R.

8.1.1 Fitting linear models

We had a quick glance at linear models in Section 3.7. There we used the `mtcars` data:

```
?mtcars
View(mtcars)
```

First, we plotted fuel consumption (`mpg`) against gross horsepower (`hp`):

```
library(ggplot2)
ggplot(mtcars, aes(hp, mpg)) +
      geom_point()
```

Given n observations of p explanatory variables (also known as predictors, covariates, independent variables, and features), the linear model is:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \epsilon_i, \qquad i = 1, \ldots, n$$

where ϵ_i is a random error with mean 0, meaning that the model also can be written as:

$$E(y_i) = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip}, \qquad i = 1, \ldots, n$$

We fitted a linear model using `lm`, with `mpg` as the response variable and `hp` as the explanatory variable:

```
m <- lm(mpg ~ hp, data = mtcars)
summary(m)
```

We added the fitted line to the scatterplot by using `geom_abline`:

```
# Check model coefficients:
coef(m)

# Add regression line to plot:
ggplot(mtcars, aes(hp, mpg)) +
      geom_point() +
      geom_abline(aes(intercept = coef(m)[1], slope = coef(m)[2]),
                  colour = "red")
```

We had a look at some diagnostic plots given by applying `plot` to our fitted model `m`:

```
plot(m)
```

Finally, we added another variable, the car weight `wt`, to the model:

```
m <- lm(mpg ~ hp + wt, data = mtcars)
summary(m)
```

Next, we'll look at what more R has to offer when it comes to regression. Before that though, it's a good idea to do a quick exercise to make sure that you remember how to fit linear models.

~

Exercise 8.1. The `sales-weather.csv` data from Section 5.12 describes the weather in a region during the first quarter of 2020. Download the file from the book's web page. Fit a linear regression model with `TEMPERATURE` as the response variable and `SUN_HOURS` as an explanatory variable. Plot the results. Is there a connection?

You'll return to and expand this model in the next few exercises, so make sure to save your code.

Exercise 8.2. Fit a linear model to the `mtcars` data using the formula `mpg ~ .`. What happens? What is `~ .` a shorthand for?

8.1.2 Interactions and polynomial terms

It seems plausible that there could be an *interaction* between gross horsepower and weight. We can include an interaction term by adding `hp:wt` to the formula:

```
m <- lm(mpg ~ hp + wt + hp:wt, data = mtcars)
summary(m)
```

Alternatively, to include the main effects of `hp` and `wt` along with the interaction effect, we can use `hp*wt` as a shorthand for `hp + wt + hp:wt` to write the model formula more concisely:

```
m <- lm(mpg ~ hp*wt, data = mtcars)
summary(m)
```

It is often recommended to centre the explanatory variables in regression models, i.e. to shift them so that they all have mean 0. There are a number of benefits to this: for instance that the intercept then can be interpreted as the expected value of the response variable when all explanatory variables are equal to their means, i.e. in an average case[1]. It can also reduce any multicollinearity in the data, particularly when including interactions or polynomial terms in the model. Finally, it can reduce problems with numerical instability that may arise due to floating point arithmetics. Note however, that there is no need to centre the response variable[2].

Centring the explanatory variables can be done using `scale`:

```
# Create a new data frame, leaving the response variable mpg
# unchanged, while centring the explanatory variables:
mtcars_scaled <- data.frame(mpg = mtcars[,1],
                            scale(mtcars[,-1], center = TRUE,
                                  scale = FALSE))
```

[1] If the variables aren't centred, the intercept is the expected value of the response variable when all explanatory variables are 0. This isn't always realistic or meaningful.

[2] On the contrary, doing so will usually only serve to make interpretation more difficult.

```
m <- lm(mpg ~ hp*wt, data = mtcars_scaled)
summary(m)
```

If we wish to add a polynomial term to the model, we can do so by wrapping the polynomial in `I()`. For instance, to add a quadratic effect in the form of the square weight of a vehicle to the model, we'd use:

```
m <- lm(mpg ~ hp*wt + I(wt^2), data = mtcars_scaled)
summary(m)
```

8.1.3 Dummy variables

Categorical variables can be included in regression models by using *dummy variables*. A dummy variable takes the values 0 and 1, indicating that an observation either belongs to a category (1) or not (0). If the original categorical variable has more than two categories, c categories, say, the number of dummy variables included in the regression model should be $c - 1$ (with the last category corresponding to all dummy variables being 0). R does this automatically for us if we include a `factor` variable in a regression model:

```
# Make cyl a categorical variable:
mtcars$cyl <- factor(mtcars$cyl)

m <- lm(mpg ~ hp*wt + cyl, data = mtcars)
summary(m)
```

Note how only two categories, 6 cylinders and 8 cylinders, are shown in the summary table. The third category, 4 cylinders, corresponds to both those dummy variables being 0. Therefore, the coefficient estimates for `cyl6` and `cyl8` are relative to the remaining *reference category* `cyl4`. For instance, compared to `cyl4` cars, `cyl6` cars have a higher fuel consumption, with their `mpg` being 1.26 lower.

We can control which category is used as the reference category by setting the order of the `factor` variable, as in Section 5.4. The first `factor` level is always used as the reference, so if for instance we want to use `cyl6` as our reference category, we'd do the following:

```
# Make cyl a categorical variable with cyl6 as
# reference variable:
mtcars$cyl <- factor(mtcars$cyl, levels =
                       c(6, 4, 8))

m <- lm(mpg ~ hp*wt + cyl, data = mtcars)
summary(m)
```

Dummy variables are frequently used for modelling differences between different

groups. Including only the dummy variable corresponds to using different intercepts for different groups. If we also include an interaction with the dummy variable, we can get different slopes for different groups. Consider the model

$$E(y_i) = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_{12} x_{i1} x_{i2}, \qquad i = 1, \dots, n$$

where x_1 is numeric and x_2 is a dummy variable. Then the intercept and slope changes depending on the value of x_2 as follows:

$$E(y_i) = \beta_0 + \beta_1 x_{i1}, \qquad \text{if } x_2 = 0,$$
$$E(y_i) = (\beta_0 + \beta_2) + (\beta_1 + \beta_{12}) x_{i1}, \qquad \text{if } x_2 = 1.$$

This yields a model where the intercept and slope differs between the two groups that x_2 represents.

~

Exercise 8.3. Return to the weather model from Exercise 8.1. Create a dummy variable for precipitation (zero precipitation or non-zero precipitation) and add it to your model. Also include an interaction term between the precipitation dummy and the number of sun hours. Are any of the coefficients significantly non-zero?

8.1.4 Model diagnostics

There are a few different ways in which we can plot the fitted model. First, we can of course make a scatterplot of the data and add a curve showing the fitted values corresponding to the different points. These can be obtained by running `predict(m)` with our fitted model `m`.

```
# Fit two models:
mtcars$cyl <- factor(mtcars$cyl)
m1 <- lm(mpg ~ hp + wt, data = mtcars) # Simple model
m2 <- lm(mpg ~ hp*wt + cyl, data = mtcars) # Complex model

# Create data frames with fitted values:
m1_pred <- data.frame(hp = mtcars$hp, mpg_pred = predict(m1))
m2_pred <- data.frame(hp = mtcars$hp, mpg_pred = predict(m2))

# Plot fitted values:
library(ggplot2)
ggplot(mtcars, aes(hp, mpg)) +
      geom_point() +
      geom_line(data = m1_pred, aes(x = hp, y = mpg_pred),
                colour = "red") +
```

```
      geom_line(data = m2_pred, aes(x = hp, y = mpg_pred),
                colour = "blue")
```

We could also plot the observed values against the fitted values:

```
n <- nrow(mtcars)
models <- data.frame(Observed = rep(mtcars$mpg, 2),
                     Fitted = c(predict(m1), predict(m2)),
                     Model = rep(c("Model 1", "Model 2"), c(n, n)))

ggplot(models, aes(Fitted, Observed)) +
      geom_point(colour = "blue") +
      facet_wrap(~ Model, nrow = 3) +
      geom_abline(intercept = 0, slope = 1) +
      xlab("Fitted values") + ylab("Observed values")
```

Linear models are fitted and analysed using a number of assumptions, most of which are assessed by looking at plots of the model residuals, $y_i - \hat{y}_i$, where $\hat{y}_i$ is the fitted value for observation i. Some important assumptions are:

- The model is *linear in the parameters*: we check this by looking for non-linear patterns in the residuals, or in the plot of observed against fitted values.
- *The observations are independent*: which can be difficult to assess visually. We'll look at models that are designed to handle correlated observations in Sections 8.4 and 9.6.
- *Homoscedasticity*: that the random errors all have the same variance. We check this by looking for non-constant variance in the residuals. The opposite of homoscedasticity is heteroscedasticity.
- *Normally distributed random errors*: this assumption is important if we want to use the traditional parametric p-values, confidence intervals and prediction intervals. If we use permutation p-values or bootstrap intervals (as we will later in this chapter), we no longer need this assumption.

Additionally, residual plots can be used to find influential points that (possibly) have a large impact on the model coefficients (influence is measured using *Cook's distance* and potential influence using *leverage*). We've already seen that we can use `plot(m)` to create some diagnostic plots. To get more and better-looking plots, we can use the `autoplot` function for `lm` objects from the `ggfortify` package:

```
library(ggfortify)
autoplot(m1, which = 1:6, ncol = 2, label.size = 3)
```

In each of the plots, we look for the following:

- Residuals versus fitted: look for patterns that can indicate non-linearity, e.g. that the residuals all are high in some areas and low in others. The blue line is there to aid the eye - it should ideally be relatively close to a

straight line (in this case, it isn't perfectly straight, which could indicate a mild non-linearity).
- Normal Q-Q: see if the points follow the line, which would indicate that the residuals (which we for this purpose can think of as estimates of the random errors) follow a normal distribution.
- Scale-Location: similar to the residuals versus fitted plot, this plot shows whether the residuals are evenly spread for different values of the fitted values. Look for patterns in how much the residuals vary - if they e.g. vary more for large fitted values, then that is a sign of heteroscedasticity. A horizontal blue line is a sign of homoscedasticity.
- Cook's distance: look for points with high values. A commonly-cited rule-of-thumb (Cook & Weisberg, 1982) says that values above 1 indicate points with a high influence.
- Residuals versus leverage: look for points with a high residual and high leverage. Observations with a high residual but low leverage deviate from the fitted model but don't affect it much. Observations with a high residual and a high leverage likely have a strong influence on the model fit, meaning that the fitted model could be quite different if these points were removed from the dataset.
- Cook's distance versus leverage: look for observations with a high Cook's distance and a high leverage, which are likely to have a strong influence on the model fit.

A formal test for heteroscedasticity, the Breusch-Pagan test, is available in the `car` package as a complement to graphical inspection. A low p-value indicates statistical evidence for heteroscedasticity. To run the test, we use `ncvTest` (where "ncv" stands for non-constant variance):

```
install.packages("car")
library(car)
ncvTest(m1)
```

A common problem in linear regression models is multicollinearity, i.e. explanatory variables that are strongly correlated. Multicollinearity can cause your β coefficients and p-values to change greatly if there are small changes in the data, rendering them unreliable. To check if you have multicollinearity in your data, you can create a scatterplot matrix of your explanatory variables, as in Section 4.8.1:

```
library(GGally)
ggpairs(mtcars[, -1])
```

In this case, there are some highly correlated pairs, `hp` and `disp` among them. As a numerical measure of collinearity, we can use the generalised variance inflation factor (GVIF), given by the `vif` function in the `car` package:

```
library(car)
m <- lm(mpg ~ ., data = mtcars)
```

```
vif(m)
```

A high GVIF indicates that a variable is highly correlated with other explanatory variables in the dataset. Recommendations for what a "high GVIF" is varies, from 2.5 to 10 or more.

You can mitigate problems related to multicollinearity by:

- Removing one or more of the correlated variables from the model (because they are strongly correlated, they measure almost the same thing anyway!),
- Centring your explanatory variables (particularly if you include polynomial terms),
- Using a regularised regression model (which we'll do in Section 9.4).

~

Exercise 8.4. Below are two simulated datasets. One exhibits a nonlinear dependence between the variables, and the other exhibits heteroscedasticity. Fit a model with `y` as the response variable and `x` as the explanatory variable for each dataset, and make some residual plots. Which dataset suffers from which problem?

```
exdata1 <- data.frame(
   x = c(2.99, 5.01, 8.84, 6.18, 8.57, 8.23, 8.48, 0.04, 6.80,
         7.62, 7.94, 6.30, 4.21, 3.61, 7.08, 3.50, 9.05, 1.06,
         0.65, 8.66, 0.08, 1.48, 2.96, 2.54, 4.45),
   y = c(5.25, -0.80, 4.38, -0.75, 9.93, 13.79, 19.75, 24.65,
         6.84, 11.95, 12.24, 7.97, -1.20, -1.76, 10.36, 1.17,
         15.41, 15.83, 18.78, 12.75, 24.17, 12.49, 4.58, 6.76,
         -2.92))

exdata2 <- data.frame(
   x = c(5.70, 8.03, 8.86, 0.82, 1.23, 2.96, 0.13, 8.53, 8.18,
         6.88, 4.02, 9.11, 0.19, 6.91, 0.34, 4.19, 0.25, 9.72,
         9.83, 6.77, 4.40, 4.70, 6.03, 5.87, 7.49),
   y = c(21.66, 26.23, 19.82, 2.46, 2.83, 8.86, 0.25, 16.08,
         17.67, 24.86, 8.19, 28.45, 0.52, 19.88, 0.71, 12.19,
         0.64, 25.29, 26.72, 18.06, 10.70, 8.27, 15.49, 15.58,
         19.17))
```

Exercise 8.5. We continue our investigation of the weather models from Exercises 8.1 and 8.3.

1. Plot the observed values against the fitted values for the two models that you've fitted. Does either model seem to have a better fit?

2. Create residual plots for the second model from Exercise 8.3. Are there any influential points? Any patterns? Any signs of heteroscedasticity?

8.1.5 Transformations

If your data displays signs of heteroscedasticity or non-normal residuals, you can sometimes use a Box-Cox transformation (Box & Cox, 1964) to mitigate those problems. The Box-Cox transformation is applied to your dependent variable y. What it looks like is determined by a parameter λ. The transformation is defined as $\frac{y_i^\lambda - 1}{\lambda}$ if $\lambda \neq 0$ and $\ln(y_i)$ if $\lambda = 0$. $\lambda = 1$ corresponds to no transformation at all. The `boxcox` function in `MASS` is useful for finding an appropriate choice of λ. Choose a λ that is close to the peak (inside the interval indicated by the outer dotted lines) of the curve plotted by `boxcox`:

```
m <- lm(mpg ~ hp + wt, data = mtcars)

library(MASS)
boxcox(m)
```

In this case, the curve indicates that $\lambda = 0$, which corresponds to a log-transformation, could be a good choice. Let's give it a go:

```
mtcars$logmpg <- log(mtcars$mpg)
m_bc <- lm(logmpg ~ hp + wt, data = mtcars)
summary(m_bc)

library(ggfortify)
autoplot(m_bc, which = 1:6, ncol = 2, label.size = 3)
```

The model fit seems to have improved after the transformation. The downside is that we now are modelling the log-mpg rather than mpg, which make the model coefficients a little difficult to interpret.

~

Exercise 8.6. Run `boxcox` with your model from Exercise 8.3. Does it indicate that a transformation can be useful for your model?

8.1.6 Alternatives to `lm`

Non-normal regression errors can sometimes be an indication that you need to transform your data, that your model is missing an important explanatory variable, that there are interaction effects that aren't accounted for, or that the relationship between the variables is non-linear. But sometimes, you get non-normal errors simply because the errors are non-normal.

The p-values reported by `summary` are computed under the assumption of normally distributed regression errors, and can be sensitive to deviations from normality. An alternative is to use the `lmp` function from the `lmPerm` package, which provides permutation test p-values instead. This doesn't affect the model fitting in any way - the only difference is how the p-values are computed. Moreover, the syntax for `lmp` is identical to that of `lm`:

```
# First, install lmPerm:
install.packages("lmPerm")

# Get summary table with permutation p-values:
library(lmPerm)
m <- lmp(mpg ~ hp + wt, data = mtcars)
summary(m)
```

In some cases, you need to change the arguments of `lmp` to get reliable p-values. We'll have a look at that in Exercise 8.12. Relatedly, in Section 8.1.7 we'll see how to construct bootstrap confidence intervals for the parameter estimates.

Another option that does affect the model fitting is to use a *robust* regression model based on M-estimators. Such models tend to be less sensitive to outliers, and can be useful if you are concerned about the influence of deviating points. The `rlm` function in `MASS` is used for this. As was the case for `lmp`, the syntax for `rlm` is identical to that of `lm`:

```
library(MASS)
m <- rlm(mpg ~ hp + wt, data = mtcars)
summary(m)
```

Another option is to use Bayesian estimation, which we'll discuss in Section 8.1.13.

~

Exercise 8.7. Refit your model from Exercise 8.3 using `lmp`. Are the two main effects still significant?

8.1.7 Bootstrap confidence intervals for regression coefficients

Assuming normality, we can obtain parametric confidence intervals for the model coefficients using `confint`:

```
m <- lm(mpg ~ hp + wt, data = mtcars)

confint(m)
```

I usually prefer to use bootstrap confidence intervals, which we can obtain using `boot`

and `boot.ci`, as we'll do next. Note that the only random part in the linear model

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \epsilon_i, \qquad i = 1, \ldots, n$$

is the error term ϵ_i. In most cases, it is therefore this term (and this term only) that we wish to resample. The explanatory variables should remain constant throughout the resampling process; the inference is conditioned on the values of the explanatory variables.

To achieve this, we'll resample from the model residuals, and add those to the values predicted by the fitted function, which creates new bootstrap values of the response variable. We'll then fit a linear model to these values, from which we obtain observations from the bootstrap distribution of the model coefficients.

It turns out that the bootstrap performs better if we resample not from the original residuals $e_1, \ldots, e_n$, but from scaled and centred residuals $r_i - \bar{r}$, where each r_i is a scaled version of residual e_i, scaled by the leverage h_i:

$$r_i = \frac{e_i}{\sqrt{1 - h_i}},$$

see Chapter 6 of Davison & Hinkley (1997) for details. The leverages can be computed using `lm.influence`.

We implement this procedure in the code below (and will then have a look at convenience functions that help us achieve the same thing more easily). It makes use of `formula`, which can be used to extract the model formula from regression models:

```
library(boot)

coefficients <- function(formula, data, i, predictions, residuals) {
      # Create the bootstrap value of response variable by
      # adding a randomly drawn scaled residual to the value of
      # the fitted function for each observation:
      data[,all.vars(formula)[1]] <- predictions + residuals[i]

      # Fit a new model with the bootstrap value of the response
      # variable and the original explanatory variables:
      m <- lm(formula, data = data)
      return(coef(m))
}

# Fit the linear model:
m <- lm(mpg ~ hp + wt, data = mtcars)

# Compute scaled and centred residuals:
res <- residuals(m)/sqrt(1 - lm.influence(m)$hat)
```