# Exploratory Data Analysis with R

Dylan Childs

2021-02-09

# Contents

# What you will learn

This book provides a self-contained introduction to how to use R for exploratory data analysis. Think of it as a resource to be referred to when needed. There is no need to memorise everything in this book. Instead, aim to understand the key concepts and familiarise yourself with the content, so that you know where to look for information when you need it. The details will get easier with practise.

## Aims

This book has three related aims:

1. Introduce the R ecosystem. R is widely used by biologists and environmental scientists to manipulate and clean data, produce high quality figures, and carry out statistical analyses. We will teach you some basic R programming so that you are in a position to address these needs in future if you need to. You don't have to become an expert programmer to have a successful career in biology but knowing a little bit of programming has almost become a prerequisite for doing research in the 21$^{st}$ century.

2. Demonstrate how to use R to carry out data manipulation and visualisation. Designing good experiments, collecting data, and analysis are hard, and these activities often take a great deal time and money. If you want to effectively communicate your hard-won results, it is difficult to beat a good figure or diagram; conversely, if you want to be ignored, put everything into a boring table. R is really good at producing figures, so even if you end up just using it as a platform for visualising data, your time hasn't been wasted.

3. Provides a foundation for learning statistics later on. If you want to be a biologist, particularly one involved in research, there is really no way to avoid using statistics. You might be able to dodge it by becoming a theoretician, but if that really is your primary interest you should probably being studying for a mathematics degree. For the rest of us who collect and analyse data knowing about statistics is essential: it allows us to

distinguish between real patterns (the "signal") and chance variation (the "noise").

## Topics

The topics we will cover are divided into three 'blocks':

The **Getting Started with R** block introduces the R language and the RStudio environment. The aim is to quickly run through what you need to know to start using R productively. This includes some basic terminology, how to use R packages, and how to access help. We are not trying to turn you into an expert programmer—though you may be surprised to discover that you do enjoy it. However, by the end of this block you will know enough about R to begin learning the practical material that follows.

The **Data Wrangling** block aims to show you how to manipulate data with R. If you regularly work with data a large amount of time will inevitably be spent getting data into the format you need. The informal name for this is 'data wrangling'. This topic that is often not taught to beginners, which is a shame because mastering the art of data wrangling saves time in the long run. We'll learn how to get data into and out of R, makes subsets of important variables, create new variables, summarise your data, and so on.

The **Exploratory Data Analysis** block is all about using R to help you understand and describe your data. The first step in any analysis after you have managed to wrangle the data into shape should involve some kind of visualisation or numerical summary. You will learn how to do this using one of the best plotting systems in R: **ggplot2**. We will review the different kinds of 'variables' you might have to analyse, discuss the different ways you can describe them, and then learn how to explore relationships between variables.

## Technologies

### What is R?

The answer to this question very much depends on who you ask. We could go on and on about the various features that R possesses. R is a functional programming language, it supports object orientation, etc etc… but these kinds of explanations are only helpful to someone who already knows about computer languages. Here's what you need to know… When a typical R user talks about "R" they are often referring to two things at once, the GNU R language and the ecosystem that exists around the language:

- R is all about **data analysis**. We can carry out any standard statistical analysis in R, as well as access a huge array of more sophisticated tools with impressive names like "structural equation model", "random forests" and "penalized regression". These days, when statisticians and computer scientists develop a new analysis tool, they often implement it in R first. This means a competent R user can always access the latest, cutting edge analysis tools. R also has the best graphics and plotting facilities of any platform. With sufficient expertise, we can make pretty much any type of figure we need (e.g. scatter plots, phylogenetic trees, spatial maps, or even volcanoes). In short, R is a very productive environment for doing data analysis.

- Because R is such a good environment for data analysis, a very large **community of users** has grown up around it. The size of this community has increased steadily since R was created, but this growth has really increased up in the last 5-10 years or so. In the early 2000s there were very few books about R and the main way to access help online was through the widely-feared R mailing lists. Now, there are probably hundreds of books about different aspects of R, online tutorials written by enthusiasts, and many websites that exist solely to help people learn R. The resulting ecosystem is vast, and though it can be difficult to navigate at times, when we run into an R-related problem the chances are that the answer is already written down somewhere[1].

R is not just about data analysis. R is a fully-fledged programming language, meaning that once you become proficient with it you can do things such as construct numerical simulation models, solve equations, query websites, send emails or carry out many other tasks we don't have time to write down. We won't do any of this year or next but it is worth noting that R can do much more than just analyse data if we need it to.

## What is RStudio?

R is essentially just a computer program that sits there and waits for instructions in the form of text. Those instructions can be typed in by a user or they can be sent to it from another program. R also runs in a variety of different environments. The job of RStudio is to provide an environment that makes R a more pleasant and productive tool.

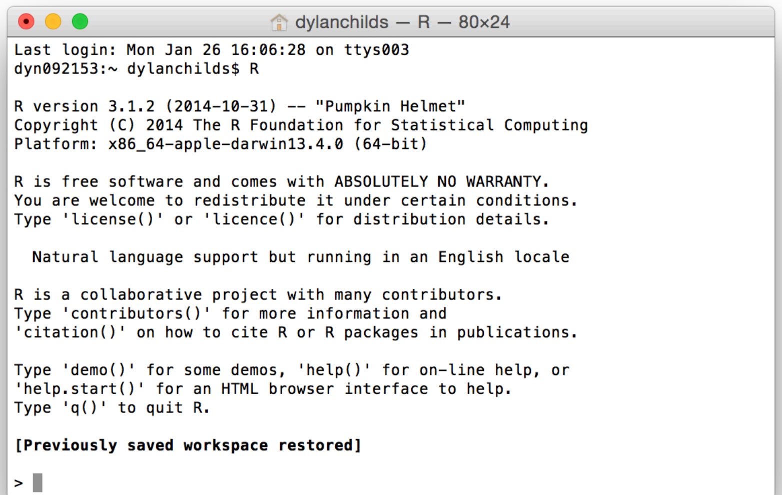
**R and RStudio are not the same thing.**

RStudio is a different program from R—it is installed separately and occupies its own place in the Programs menu (Windows PC) or Applications folder (Mac). We can run R without RStudio if we need to, but we cannot run RStudio without R. Remember that!

One way to get a sense of why RStudio is a Very Good Thing is to look at what running R without it is like. The simplest way to run it on a Linux or

[1]The other big change is that R is finally starting to become part of the commercial landscape—learning how to use it can only improve your job prospects.

Unix-based machine (like a Mac) is to use something called the Terminal. It's well beyond the scope of this book to get into what this is, but in a nutshell, the Terminal provides a low-level, text-based way to interact with a computer. Here is what R looks like running inside a Terminal on a Mac:




We can run R in much the same way on Windows using the "Command Prompt" if we need to. The key thing you need to take away from that screenshot is that running R like this is very "bare bones". We typed the letter "R" in the Terminal and hit Enter to start R. It printed a little information as it started up and then presented us with "the prompt" (`>`), waiting for input. This is where we type or paste in instructions telling R what to do. There is no other way to interact with it when we run R like this – no menus or buttons, just a lonely prompt waiting for instructions.

So what is RStudio? In one sense RStudio is just another Graphical User Interface for R which improves on the "bare bones" experience. However, it is a GUI on steroids. It is more accurate to describe it as an Integrated Development Environment (IDE). There is no all-encompassing definition of an IDE, but they all exist to make programmer's lives easier by integrating various useful tools into a single piece of software. From the perspective of this book, there are four key features that we care about:

- The R interpreter—the thing that was running in the Terminal above—runs inside RStudio. It's accessed via a window labelled Console. This is where we type in instructions we want to execute when we are working directly with R. The Console also shows us any output that R prints in response to these instructions. So if we just want the “bare bones” experience, we can still have it.

- RStudio provides facilities for working with R programs using something called a Source Code Editor. An R program ( also called a “script”)” is just is a collection of instructions in the R language that have been saved to a text file. Nothing more! However, it is much easier to work with a script using a proper Source Code Editor than an ordinary text editor like Notepad.
- An good IDE like RStudio also gives you a visual, point-and-click means of accessing various language-specific features. This is a bit difficult to explain until we have have actually used some of these, but trust us, being able to do things like manage packages, set working directories, or inspect objects we’ve made simplifies day-to-day use of R. This especially true for new users.
- RStudio is cross-platform—it will run on a Windows PC, a Linux PC or a Mac. In terms of the appearance and the functionality it provides, RStudio is exactly the same on each of these platforms. If we learn to work with R via RStudio on a Windows PC, it’s no problem migrating to a Mac or Linux PC later on if we need to. This is a big advantage for those of us who work on multiple platforms.

We’ll only scratch the surface of what RStudio can do. The reason for introducing a powerful tool like RStudio is because one day you may need to access sophisticated features like debugging facilities, package build tools, and repository management. RStudio makes it easy to use these advanced tools.

# How to use this book

We have adopted a number of formatting conventions in this book to distinguish between normal text, R code, file names, and so on. You need to be aware to make best use of the book.

## Text, instructions, and explanations

Normal text—instructions, explanations and so on—are written in the same type as this document. We will tend to use bold for emphasis and italics to highlight specific technical terms when they are first introduced. In addition:

- `This typeface` is used to distinguish R code within a sentence of text: e.g. "We use the `mutate` function to change or add new variables."
- A sequence of selections from an RStudio menu is indicated as follows: e.g. **File** **New File** **R Script**
- File names referred to in general text are given in upper case in the normal typeface: e.g. MYFILE.CSV.

At various points in the text you will come across text in different coloured boxes. These are designed to highlight stand-alone exercises or little pieces of supplementary information that might otherwise break the flow. There are three different kinds of boxes:

**Action!**

This is an **action** box. We use these when we want you to do something. Do not ignore these boxes.

**Information!**

This is an **information** box. These aim to offer a discussion of why something works the way it does.

**Warning!**

This is a **warning** box. These usually highlight a common 'gotcha' that might trip up new users.

## R code and output

We try to illustrate ideas using snippets of real R code where possible. It's a good idea to run these when working through a topic. The best way to learn something is to use it. Of course, in order to do that we need to know what we're looking at… Stand alone snippets will be formatted like this:

```
tmp <- 1
print(tmp)
```

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

At this point it does not matter what the above actually means. You just need to understand how the formatting of R code in this book works. The lines that start with `##` show us what R prints to the screen after it evaluates an instruction and does whatever was asked of it, that is, they show the output. The lines that **do not** start with `##` show us the instructions, that is, they show us the input. So remember, the absence of `##` shows us what we are asking R to do, otherwise we are looking at something R prints in response to these instructions.

# Get up and running

## Different ways to run RStudio

We can run RStudio in a variety of different ways—

1. Most people use the version of RStudio called **RStudio Desktop**, either in its free-to-use guise (Open Source Edition) or the commercial version (RStudio Desktop Pro). The desktop version of RStudio are stand-alone applications that run locally on a computer and have to be installed like any other piece of software. This is generally easy, but you can run into problems if you have an old computer or a Chromebook.
2. The second way to use RStudio is by accessing a version called **RStudio Server** through a web browser. RStudio Server is usually administered by professional IT people. Life is easy if you belong to an organisation that has set up RStudio Server, because all you need to get going is a user account, a semi-modern web browser and an internet connection.
3. Finally, the company that makes RStudio also runs a commercial cloud-based solution called RStudio Cloud. This allows anyone web browser and an internet connection to use R and RStudio. Although there is a free version, this is fairly limited meaning you end up paying a monthly fee to do 'real work'. However, RStudio Cloud can be a good backup option when all else fails.

**Do you need to install R and RStudio?**

If you're lucky enough to have access to RStudio Server or an RStudio Cloud account you don't need to install R and RStudio on your own computer. Just access those cloud service through a decent web browser. That said, it can be useful to have a local copy on your own computer, e.g. because you don't have a reliable internet connection. Obviously, if you can't access those cloud services you'll have to install R and RStudio to use them!

# Installing R and RStudio locally

It does not need to cost a penny to use R and RStudio. The source code for R is open source, meaning anyone with the time, energy and expertise is free to download it and alter it as they please. Open source does not necessarily mean free, as in it costs £0 to use, but luckily R **is** free in this sense. On the other hand, RStudio is developed and maintained by a for-profit company (called… RStudio). Luckily, because they make their money selling professional software and services, the open source desktop version of RStudio is also free to use. This section will show you how to download and install R and RStudio.

### Installing R

In order to install R you need to download the appropriate installer from the Comprehensive R Archive Network (CRAN). We are going to use the "base distribution" as this contains everything you need to use R under normal circumstances. There is a single installer for Windows. On a Mac, it's important to match the installer to the version of OS X. In either case, R uses a the standard install mechanism that should be familiar to anyone who has installed an application on their machine. There is no need to change the default settings—doing so will probably lead to problems later on.

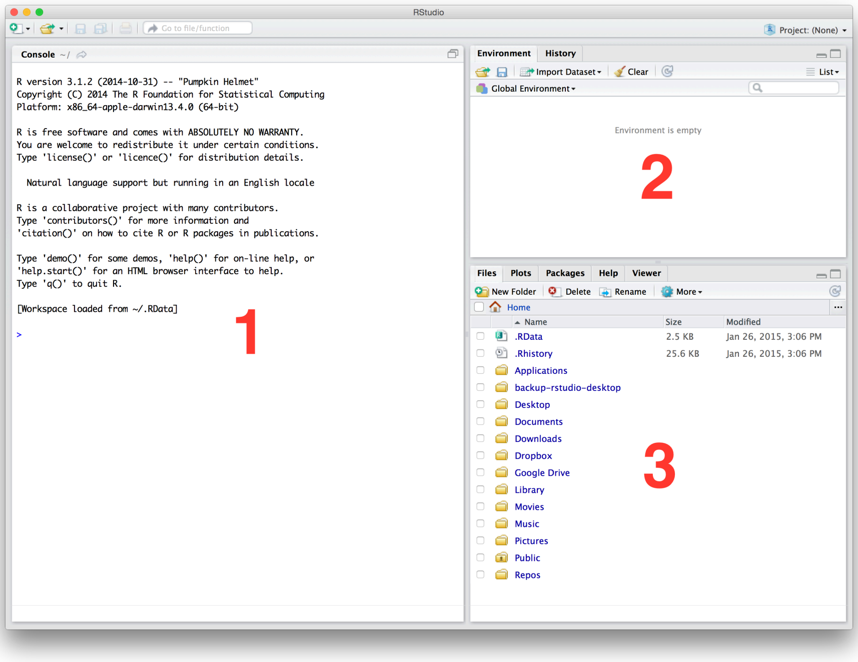
After installing R it should be visible in the Programs menu on a Windows computer or in the Applications folder on a Mac. In fact, that thing labelled 'R' is very simple a Graphical User Interface (GUI) for R. When we launch the R GUI we're presented with something called the Console, which is where we can interact directly with R by typing things at the so-called prompt, `>`, and a few buttons and menus for managing common tasks. We will not study the GUIs in any detail because we recommend using RStudio, but it's important to be aware they exist so that you don't accidentally use them instead of RStudio.

## Installing RStudio

RStudio can be downloaded from the RStudio download page. The one to go for is the Open Source Edition of RStudio Desktop, **not** the commercial version of RStudio Desktop called RStudio Desktop Pro. RStudio installs like any other piece of software—just run the installer and follow the instructions. There's no need to configure after after installation.

# A quick look at RStudio

Once installed RStudio runs like any other stand-alone application via the Programs menu or the Applications folder on a Windows PC or Mac, respectively (though it will only work properly if R is also installed). Here is how RStudio appears the first time it runs on a Mac:




There are three panes inside a single window, which we have labelled with red numbers. Each of these has a well-defined purpose. Let's take a quick look at these:

1. The large window on the left is the Console. This is basically where R lives inside RStudio. The Console lets you know what R is doing and provides a mechanism to interact with R by typing instructions. All this happens at the prompt, `>`. You will be working in the Console in the next chapter so we won't say any more about this here.

2. The window at the top right contains two or more tabs. One of these, labelled **Environment**, allows us to see all the R 'objects' we can currently access. Another, labelled **History**, allows us to see a list of instructions we've previously sent to R. The buttons in this tab allow us to reuse or save these instructions.

3. The window at the bottom right contains five tabs. The first, labelled **Files**, gives us a way to interact with the files and folders. The next tab, labelled **Plots**, is where any figures we produce are displayed. This tab also allows you to save your figures to file. The **Packages** tab is where we view, install and update packages used to extend the functionality of R. The **Help** tab is where you can access and display various different help pages. Finally, **Viewer** is an embedded web browser.

**My RStudio looks different!**

Don't be alarmed if RStudio looks different on your computer. RStudio saves its state between different sessions, so if you've have already messed about with it you will see these changes when you restart it. For example, there is a fourth pane that is often be visible in RStudio—the source code Editor we mentioned above.

## Working at the Console in RStudio

R was designed to be used interactively—it is what is known as an **interpreted language**, which we can interact with via something called a Command Line Interface (CLI). This is just a fancy way of saying that we can type instructions to "do something" into the Console and those instructions will then be interpreted when we hit the Enter key. If our R instructions do not contain any errors, R will then do something like read in some data, perform a calculation, make a figure, and so on. What actually happens obviously depends on what we ask it to do.

Let's briefly see what all this means by doing something very simple with R. Type `1 + 3` at the Console and hit the Enter key:

```
1+3
```

```
## [1] 4
```

The first line above just reminds us what we typed into the Console. The line after that beginning with `##` shows us what R printed to the Console after reading and evaluating our instructions.

What just happened? We can ignore the `[1]` bit for now (the meaning of this will become clear later in the course). What are we left with – the number 2. The instruction we gave R was in effect "evaluate the expression `1 + 3`". R read this in, decided it was a valid R expression, evaluated the expression, and then printed the result to the Console for us. Unsurprisingly, the expression `1 + 3` is a request to add the numbers 1 and 3, and so R prints the number 4 to the Console.

OK… that was not very exciting. In the next chapter we will start learning to use R to carry out more useful calculations. The important take-away from this is that this sequence of events—reading instructions, evaluating those instructions and printing their output (if there is any output)—happens every time we type or paste something into the Console and hit Enter.

**What does that word 'expression' mean?**

Why do we keep using that word *expression*? Here is what the Wikipedia page says:

> An expression in a programming language is a combination of explicit values, constants, variables, operators, and functions that are interpreted according to the particular rules of precedence and of association for a particular programming language, which computes and then produces another value.

That probably doesn't make much sense! In simple terms, an R expression is a small set of instructions that tell R to do something. That's it. We could write 'instructions' instead of 'expressions' throughout this book but we may as well use the correct word.

# Part I

# Introduction to R

# Chapter 1

# A quick introduction to R

## 1.1 Using R as a big calculator

### 1.1.1 Basic arithmetic

The Get up and running chapter showed that R could handle familiar arithmetic operations: division, multiplication, addition and subtraction. If we want to add or subtract two numbers, we place the `+` or `-` symbol in between two numbers and hit Enter. R will read the arithmetic expression, evaluate it, and print the result to the Console. This works as you'd expect:

*Addition–*

```
3 + 2
```

```
## [1] 5
```

*Subtraction–*

```
5 - 1
```

```
## [1] 4
```

Multiplication and division are no different. However, we can't use `x` or `÷` symbols for these operations. Instead, use `*` and `/` to multiply and divide:

*Multiplication–*

```
7 * 2
```

```
## [1] 14
```

*Division–*

```
3 / 2
```

```
## [1] 1.5
```

We can also exponentiate numbers, i.e. raise one number to the power of another. Use the ^ operator to do this:

*Exponentiation–*

```
4^2
```

```
## [1] 16
```

This raises 4 to the power of 2 (i.e. we squared it). In general, we can raise a number `x` to the power of `y` using `x^y`. Neither `x` nor `y` need to be whole numbers either.

**Operators?**

What does 'operator' mean? An operator is simply a symbol (or sequence of symbols) that does something specific with one or more inputs. For example, operators like `/`, `*`, `+` and `-` carry out arithmetic calculations with pairs of numbers. Operators are one of the basic building blocks of a programming language like R.

### 1.1.2 Combining arithmetic operations

We can also combine arithmetic operations. Assume we want to subtract 6 from $2^3$. The expression to perform this calculation is:

```
2^3 - 6
```

```
## [1] 2
```

Simple enough, but what if we had wanted to carry out a slightly longer calculation that required the last answer to then be divided by 2? This is the **wrong** way to do this:

```
2^3 - 6 / 2
```

```
## [1] 5
```

The answer we expect here is 1. So what happened? R evaluated 6/2 first and then subtracted this answer from $2^3$.

If that's obvious, great. If not, it's time to learn about **order of precedence**. R uses a standard set of rules to decide the order in which calculations feed into one another to unambiguously evaluate any expression. It uses the same order as every other computer language, which thankfully is the same one we all learn in mathematics classes at school. The order of precedence is:

1. exponents and roots (also, 'powers' or 'orders')
2. division and then multiplication
3. additional and then subtraction

**BODMAS and friends**

If you find it difficult to remember the standard order of precedence there are a load of mnemonics that can to help.

We need to control the order of evaluation to arrive at the answer we were looking for in the above example. Do this by grouping together calculations inside parentheses, i.e. 'round brackets' `(` and `)`. Here's the expression we should have used:

```
(2^3 - 6) / 2
```

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

We can use more than one pair of parentheses to control the order of evaluation in more complex calculations. The order of evaluation then happens 'inside-out'. For example, if we want to find the cube root of 2 (i.e. $2^{1/3}$) rather than $2^3$ in that last calculation we would instead write:

```
(2^(1/3) - 6) / 2
```

```
## [1] -2.370039
```

The parentheses around the `1/3` are needed to ensure this is evaluated prior to being used as the exponent.

**Working efficiently at the Console**

Working at the Console soon gets tedious if we have to retype similar things over and over again. There is no need to do this, though. Place the cursor at the prompt and hit the up arrow. What happens? This brings back the last expression sent to R's interpreter. Hit the up arrow again to see the last-but-one expression, and so on. We go back down the list using the down arrow. Once we're at the line we need, we use the left and right arrows to move around the expression and the delete key to remove the parts we want to change. Once an expression has been edited we can hit Enter to send it to R again. Try it.

## 1.2 Problematic calculations

Now is a good time to highlight how R handles certain kinds of awkward numerical calculations. One of these involves division of a non-zero by 0. Mathematically, division of a finite number by `0` equals A Very Large Number: infinity. Some programming languages will respond to an attempt to do this with an error. R is a bit more forgiving:

```
1 / 0
```

```
## [1] Inf
```

R has a special built-in value that allows it to handle this kind of result. This is `Inf`, which stands for 'infinity'.

The other special kind of value we sometimes run into is generated by calculations that don't have a well-defined numerical result. For example, look what happens when we try to divide 0 by 0:

```
0 / 0
```

```
## [1] NaN
```

The `NaN` in this result stands for 'Not a Number'. R produces `NaN` because 0/0 is not defined mathematically: it produces something that is Not a Number.

The reason we are pointing out `Inf` and `NaN` is not that we expect to use them. It's important to know what they represent because they often arise due to a mistake somewhere in our code. It's hard to track down such mistakes if we don't know how `Inf` and `NaN` arise.

**R as a fancy calculator**

What we've seen so far is that we can interact with R via the so-called REPL: the read-evaluate-print loop. R takes user input (e.g. `1 / 0`), evaluates it (`1 / 0 = Inf`), prints the results (`## [1] Inf`), and then waits for the next input (e.g. `0 / 0`). This facility is handy because it means we can use R interactively, working through a set of calculations line-by-line.

## 1.3 Storing and reusing results

We've not yet tried to do anything remotely complicated or interesting beyond using parentheses to construct longer calculations. This approach is acceptable when a calculation is straightforward, but it quickly becomes unwieldy for dealing with anything more complicated.

The best way to see what we mean is by working through a simple example—solving a quadratic equation. You probably remember these from school. A quadratic equation looks like this:

$$a + bx + cx^2 = 0$$

If we know the values of $a$, $b$ and $c$ then we can solve this equation to find the values of $x$ that satisfy this equation. Here's the well-known formula for these solutions:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

We can use R to calculate these solutions for us. Say that we want to find the solutions to the quadratic equation when $a = 1$, $b = 6$ and $c = 5$. We have to turn the above equation into a pair of R expressions:

*Solution 1–*

```
(-6 + (6^2 -4 * 1 * 5)^(1/2)) / (2 * 1)
```

```
## [1] -1
```

*Solution 2–*

```
(-6 - (6^2 -4 * 1 * 5)^(1/2)) / (2 * 1)
```

```
## [1] -5
```

The output tells us that the two values of $x$ that satisfy this particular quadratic equation are -1 and -5.

But what should we do if we now need to solve a different quadratic equation? Working at the Console, we could bring up the expressions we typed (using the up arrow) and change the numbers to match the new values of $a$, $b$ and $c$. However, editing expressions like this is tedious, and more importantly, it's error-prone because we have to make sure we substitute the new numbers into precisely the right positions.

A partial solution to this problem is to store the values of $a$, $b$ and $c$ in some way so that we only have to change them one. We'll see why this is useful in a moment.

First, we need to learn how to store results in R. The key to this is to use the **assignment operator**, written as an arrow pointing to the left, `<-`. Sticking with the current example, we need to store the numbers 1, 6 and 5. We do this by typing out three expressions, one after the another, each time hitting enter to get R to evaluate it:

```
a <- 1
```

```
b <- 6
```

```
c <- 5
```

The exact sequence `<-` defines the assignment operator. R won't recognise it as assignment if we try to include a space between the `<` and `-` symbols.

Notice that R didn't print anything to screen. So what actually happened? We asked R to first evaluate the expression on the right hand side of each `<-` (just a number in this case) and then **assigns the result** of that evaluation instead of printing it. Each result has a name associated with it, which appears on the left hand-side of the `<-`.

The net result of all this is that we have stored the numbers 1, 6 and 5 somewhere in R and associated them with the letters `a`, `b` and `c`, respectively. We can check whether this assignment business has worked by looking at the **Environment** tab in the top right RStudio window. There should be three 'names' listed in that tab now (`a`, `b` and `c`) along with the associated numbers 1, 6 and 5.

What does this mean in practical terms? Look at what happens if we now type the letter `a` into the Console and hit Enter:

```
a
```

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

It looks the same as if we had typed the number `1` directly into the Console. We stored the output from three separate R expressions, associating each a name so that we can access it again[1]. Whenever we use the assignment operator `<-` we are telling R to keep whatever kind of value results from the calculation on the right-hand side of `<-`, giving it the name on the left-hand side so that we can access it later.

Why is this useful? Let's imagine we want to do more than one thing with our three numbers. If we want to know their sum or their product we can now use:

*Sum–*

```
a + b + c
```

```
## [1] 12
```

*Product–*

[1]Technically, this is called **binding** the name to a value. You don't need to remember this.

```
a * b * c
```

```
## [1] 30
```

So… once we've stored a result and associated it with a name we can reuse it whenever needed. Returning to our example, we can now calculate the solutions to the quadratic equation by typing these two expressions into the Console:

*Solution 1–*

```
(-b + (b^2 -4 * a * c)^(1/2)) / (2 * a)
```

```
## [1] -1
```

*Solution 2–*

```
(-b - (b^2 -4 * a * c)^(1/2)) / (2 * a)
```

```
## [1] -5
```

Imagine we'd now like to find the solutions to a different quadratic equation where $a = 1$, $b = 5$ and $c = 5$. We only changed the value of $b$ here. To find the new solutions we have to do two things. First we change the value of the number associated with `b`…

```
b <- 5
```

…then we bring up those lines that calculate the solutions to the quadratic equation and run them, one after the other:

```
(-b + (b^2 -4 * a * c)^(1/2)) / (2 * a)
```

```
## [1] -1.381966
```

```
(-b - (b^2 -4 * a * c)^(1/2)) / (2 * a)
```

```
## [1] -3.618034
```

We don't have to retype those expressions. We can use the up arrow to bring each one back to the prompt and hit Enter. This is much simpler than editing the expressions.

More importantly, we are beginning to see the benefits of using something like R—we can break down complex calculations into a series of steps, storing and reusing intermediate results as required.

**RStudio shortcut**

We use the assignment operator `<-` all the time when working with R. Because it's inefficient to type the `<` and `-` characters repeatedly, RStudio has a built-in shortcut for typing the assignment operator.

The shortcut is 'Alt + `-`' . Try it now. Move the cursor to the Console, hold down the Alt key ('Option' on a Mac), and press the `-` sign key. RStudio will auto-magically add insert `<-`. **If you only learn one RStudio shortcut, learn this one!** It will save you a lot of time in the long run.

## 1.4 How does assignment work?

When we use the assignment operator `<-` to associate names and values, we refer to this as creating or modifying **a variable**. This is much less tedious than using words like 'associate', 'value', and 'name' all the time. Why is it called a variable? What happens when we run these lines:

*Create* `myvar` *and print out its value*–

```
myvar <- 1
myvar
```

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

*Modify* `myvar` and *and print out its new value* –

```
myvar <- 7
myvar
```

```
## [1] 7
```

The first time we used `<-` with `myvar` on the left-hand side, we **created** a variable `myvar` associated with the value 1. We then printed out the value associated with `myvar`. The second line `myvar <- 7` **modified** the value of `myvar` to be 7 (and printed this out again). This is why we refer to `myvar` as a variable: we can change its value as we please.

What happened to the old value associated with `myvar`? In short, it is gone, kaput, lost… forever. The moment we assign a new value to `myvar` the old one is destroyed and can no longer be accessed. Remember this.

Keep in mind that the expression on the right-hand side of `<-` can be any kind of calculation and the variable can have any (valid) name we like. For example, if we want to perform the calculation `(1 + 2^3) / (2 + 7)` and associate the result with the word `answer`, we would do this:

```r
answer <- (1 + 2^3) / (2 + 7)
answer
```

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

Any expression can be used on the right-hand side of the assignment operator as along as it generates an output of some kind. For example, we can create new variables from others:

```r
newvar <- 2 * answer
```

What happened here? Start at the right-hand side of `<-`. The expression on this side contained the variable `answer` so R went to see if `answer` actually exists. It does, so it then substituted the value associated with `answer` into the calculation and assigned the resulting value of 2 to `newvar`.

Finally, look at what happens if we copy a variable using the assignment operator:

```r
myvar <- 7
mycopy <- myvar
```

We now have a pair of variables, `myvar` and `mycopy`, associated with the number 7. Each of these is associated with a **different copy** of this number. If we change the value associated with one of these variables it does not change the value of the other, as this shows:

```r
myvar <- 10
```

```r
myvar
```

```
## [1] 10
```

```r
mycopy
```

```
## [1] 7
```

R always behaves like this unless we work hard to alter this behaviour. Remember that—every time we assign one variable to another, we actually make a completely new, independent copy of its associated value. That probably doesn't seem like an obvious or important point, but trust us, it is. It will be critical to remember this behaviour when we start learning how to manipulate data sets.

## 1.5 Global environment

Whenever we associate a name with a value we create a copy of both these things somewhere in the computer's memory. In R the "somewhere" is called an environment. We aren't going to get into a discussion of R's many different kinds of environments—that's an advanced topic well beyond the scope of this book. The one environment we do need to be aware of is the **Global Environment**.

Whenever we perform an assignment in the Console the variable we create is placed into the Global Environment. The set of variables currently in existence are listed in the **Environment** tab in RStudio. Take a look. There are two columns in the **Environment** tab: the first shows the names of the variables, the second summarises their values.

> **The Global Environment is temporary**
>
> By default, R will try to save everything in the Global Environment when we close it down and restore everything when we start the next R session. It does this by writing a copy of the Global Environment to disk. In theory, this means we can close down R, reopen it, and pick things up from where we left off. **Don't rely on this behaviour!** It just increases the risk of making a mistake.

## 1.6 Naming rules and conventions

We don't have to use a single letter to name things in R. We could use the words `tom`, `dick` and `harry` in place of `a`, `b` and `c`. It might be confusing to use them, but `tom`, `dick` and `harry` are all legal names as far as to R is concerned:

- A legal name in R is any sequence of letters, numbers, `.`, or `_`, but the sequence of characters we use must begin with a letter. Both upper and lower case letters are allowed. For example, `num_1`, `num.1`, `num1`, `NUM1`, `myNum1` are all legal names, but `1num` and `_num1` are not because they begin with `1` and `_`.
- R is case sensitive—it treats upper and lower case letters as different characters. This means R treats `num` and `Num` as distinct names. Forgetting about case sensitivity is a good way to create errors when using R. Try to remember that.

> **Don't begin a name with .**
>
> We are allowed to begin a name with a `.`, but this usually is A Bad Idea. Why? Because variable names that begin with `.` are hidden from view in the Global Environment—the value it refers to exists but it's invisible.

This behaviour exists to allow R to create invisible variables that control how it behaves. This is useful, but it isn't really meant to be used by the average user.

# Chapter 2

# Using functions

## 2.1 Introduction

Functions are an essential building block of any programming language. The job of a function is to carry out a calculation or computation that would typically require many lines code to do 'from scratch'. Functions allow us to reuse common computations while offering some control over the precise details of what happens. To use R effectively—even if our needs are very simple—we need to understand how to use functions. This chapter aims to explain what functions are for, how to use them, and how to avoid mistakes when doing so, without getting lost in the detail of how they work.

## 2.2 Functions and arguments

Functions allow us to reuse a calculation. The best way to see what we mean by this is to see one in action. The `round` function rounds numbers to a significant number of digits (no surprises there). To use it, we could type this into the Console and hit Enter:

```
round(x = 3.141593, digits = 2)
```

We have suppressed the output so that we can unpack things a bit first. We rely on the same basic construct every time we use a function. This starts with the name of the function as the prefix. In the example above, the function name is `round`. After the function name, we need a pair of opening and closing parentheses. This combination of name and parentheses alerts R to fact that we are using a function.

What about the bits inside the parentheses? These are called the **arguments** of the function. That's a horrible name, but it is the one that everyone uses, so we have to get used to it. Depending on how it was defined, a function can take zero, one, or more arguments. We will discuss this idea in more detail later in this section. In simple terms, the arguments control the behaviour of a function.

We used the `round` function with two arguments. We supplied each one as a name-value pair separated by a comma. When working with arguments, name-value pairs occur either side of the equals sign (`=`), with the argument **name** on the left-hand side and its **value** on the right-hand side. The name serves to identify which argument we are working with, and the value is the thing that controls what that argument does.

We call the process of associating argument names and values 'setting the arguments' of the function (or 'supplying the arguments'). Notice the similarity between supplying function arguments and the assignment operation discussed in the last topic. The difference is that name-value pairs are associated with the `=` symbol when involved in arguments.

**Use `=` to assign arguments**

**Do not** use the assignment operator `<-` inside the parentheses when working with functions. This is a "trust us" situation—you will end up in all kinds of difficulty if you do this!

The arguments control the behaviour of a function. Our job as users is to set the values of these to get the behaviour we want. However, The function determines arguments we are allowed to use, i.e. we are not free to choose whatever name we like[1].

```
round(x = 3.141593, digits = 2)
```

```
## [1] 3.14
```

The `round` function rounds one or more numeric inputs and rounds these to a particular number of significant digits. The argument that specifies the number(s) to round is `x`; the second argument, `digits`, specifies the number of decimal places we require. Based on the supplied values of these arguments, `3.141593` and `2`, respectively, the `round` function spits out a value of `3.14`, which is then printed to the Console.

What if we had wanted to the answer to 3 significant digits? We would set the `digits` argument to 3:

```
round(x = 3.141593, digits = 3)
```

[1] We say "typically", because R is a very flexible language and so there are certain exceptions to this simple rule of thumb. For now it is simpler to think of the names as constrained by the particular function we're using. Let's return to th example to see how all this works:

```
## [1] 3.142
```

This illustrates what we mean when we say arguments control the behaviour of the function—`digits` sets the number of significant digits calculated by `round`.

## 2.3 Evaluating arguments and returning results

Whenever R evaluates a function, we refer to this action as 'calling the function'. In our simple example, we called the `round` function with arguments `x` and `digits`. That said, we often just say 'use the function' because that is more natural to most users.

Several things happen when we call functions: first they **evaluate** their arguments, then they perform some action, and finally (optionally) **return** a value to us. Let's work through what all that means…

What do we mean by the word 'evaluate'? When we call a function, what typically happens is:

1. the R expression on the right-hand side of an `=` is evaluated,
2. the result is associated with the corresponding argument name, and
3. the function does its calculations using the resulting name-value pairs.

To see how the evaluation step works, take a look at a new example using `round`:

```
round(x = 2.3 + 1.4, digits = 0)
```

```
## [1] 4
```

What happened above is that R evaluated `2.3 + 1.4`, resulting in the number `3.7`, which was then associated with the argument `x`. We set `digits` to `0` this time so that `round` returns a whole number, `4`.

The important thing to realise is that the expression(s) on the right-hand side of the `=` can be anything we like. This third example essentially equivalent to the last one:

```
y <- 2.3 + 1.4
round(x = y, digits = 0)
```

```
## [1] 4
```

This time we created a new variable called `y` and supplied this as the value of the `x` argument. When we use the `round` function like this, the R interpreter spots that something on the right-hand side of an `=` is a variable and associates the value of this variable with `x` argument. As long as we have defined the numeric variable `y` at some point we can use it as the value of an argument.

At the beginning of this section, we said that a function may optionally **return** a value to us when it completes its task. That word 'return' refers to the process by which a function outputs a value. If we use a function at the Console the returned value is printed out. However, we can use this value in other ways. For example, there is nothing to stop us combining function calls with the arithmetic operations:

```
2 * round(x = 2.64, digits = 0)
```

```
## [1] 6
```

Here the R interpreter evaluates the function call and then multiplies the value it returns by 2. If we want to reuse this value, we have to assign the result of function call, for example:

```
roundnum <- 2 * round(x = 2.64, digits = 0)
```

Using a function with `<-` is no different from the examples using multiple arithmetic operations in the last topic. The R interpreter starts on the right-hand side of the `<-`, evaluates the function call there, and only then assigns the value to `roundnum`.

**Argument names vs variable names**

Keeping in mind what we've just learned, take a careful look at this example:

```
x <- 0
round(x = 3.7, digits = x)
```

```
## [1] 4
```

What is going on here? The key to understanding this is to realise that the symbol `x` is used in two different ways here. When it appears on the left-hand side of the `=` it represents an argument name. When it appears on the right-hand side, it is treated as a variable name, which must have a value associated with it for the above to be valid. That is a confusing way to use the `round` function, but it is perfectly valid.

The message here is that what matters is where things appear relative to the `=`, not the symbols used to represent them.

## 2.4 Specifying function arguments

So far, we have been concentrating on functions that carry out mathematical calculations with numbers. Functions can do all kinds of things. For example, some functions are designed to extract information about other functions. Take a look at the `args` function:

```
args(name = round)
```

```
## function (x, digits = 0)
## NULL
```

`args` prints a summary of the main arguments of a function. What can we learn from the summary of the arguments of `round`? Notice that the first one, `x`, is shown without an associated value, whereas the `digits` part of the summary is printed as `digits = 0`.

**The significance of this is that `digits` has a default value** (0 in this case). This means that we can leave out `digits` when using the round function:

```
round(x = 3.7)
```

```
## [1] 4
```

This is the same result as we would get using `round(x = 3.7, digits = 0)`. This 'default value' behaviour is useful because it allows us keep our R code concise. Some functions take a large number of arguments, many of which are defined with sensible defaults. Unless we need to change these default arguments, we can ignore them when we call such functions.

Notice that the `x` argument of `round` does not have a default, which means we have to supply a value. This is sensible, as the whole purpose of `round` is to round any number we give it.

There is another way to simplify our use of functions. Take a look at this example:

```
round(3.72, digits = 1)
```

```
## [1] 3.7
```

What does this demonstrate? **We do not have to specify argument names**. In the absence of a name R uses the position of the supplied argument to work out which name to associate it with. In this example we left out the name of the argument at position 1. This is where `x` belongs, so we end up rounding 3.71 to 1 decimal place.

R is even more flexible than this. **We don't necessarily have to use the full name of an argument,** because R can use partial matching on argument names:

```
round(3.72, dig = 1)
```

```
## [1] 3.7
```

This also works because R can unambiguously match the argument we named `dig` to `digits`.

**Be careful with your arguments**

Here is some advice. Do not rely on partial matching of function names. It just leads to confusion and the odd error. If you use it a lot, you end up forgetting the true name of arguments, and if you abbreviate too much, you create name matching conflicts. For example, if a function has arguments `arg1` and `arg2` and you use the partial name `a` for an argument, there is no way to know which argument you meant. We are pointing out partial matching so that you are aware of the behaviour. It is not worth the hassle of getting it wrong to save on a little typing, so do not use it.

What about position matching? This can also cause problems if we're not paying attention. For example, if you forget the order of the arguments to a function and then place your arguments in the wrong place, you will either generate an error or produce a nonsensical result. It is nice not to have to type out the `name = value` construct all the time though, so our advice is to rely on positional matching only for the first argument. This is a common convention in R that makes sense because it is often obvious what kind of information the first argument should carry.

## 2.5 Combining functions

Using R to do 'real work' usually involves linked steps, each facilitated by a different function. There is more than one way to achieve this. Here's a simple example that uses an approach we already know about—storing intermediate results:

```
y <- sqrt(10)
round(y, digits = 1)
```

```
## [1] 3.2
```

These two lines calculate the square root of the number 10 and assigned the result to `y`, then round this to one decimal place and print the result. We linked the two calculations by assigning a name to the first result and then used this as the input to a function in the second step.

Here is another way to replicate the same calculation:

```
round(sqrt(10), digits = 1)
```

```
## [1] 3.2
```

The technical name for this is **function composition** or **function nesting**: the `sqrt` function is 'nested inside' the `round` function. The way we have to read these constructs is **from the inside out**. The `sqrt(10)` expression is on the right-hand side of an `=` symbol, so this is evaluated first. The result of `sqrt(10)` is then associated with the first argument of the `round` function, and only then does the `round` function do its job.

There aren't any new ideas here. We have already seen that R evaluates whatever is on the right-hand side of the `=` symbol first before associating the resulting value with the appropriate argument name.

We can extend this nesting idea to do more complicated calculations, i.e. there's nothing to stop us using multiple levels of nesting either. Take a look at this example:

```
round(sqrt(abs(-10)), digits = 1)
```

```
## [1] 3.2
```

The `abs` function takes the absolute value of a number, i.e. removes the `-` sign if it is there. Remember, read nested calls from the inside out:

1. we find the absolute value of -10,
2. we calculate the square root of the resulting number (+10), and
3. then we rounded this to a whole number.

Nested function calls can be useful because they make R code less verbose (we write less), but this comes at a high cost of reduced readability. No reasonable person would say that `round(sqrt(abs(-10)), digits = 1)` is easy to read! For this reason, we aim to keep function nesting to a minimum. We will occasionally have to use the nesting construct, so it is important to understand how it works even if we don't like it.

The good news is that we'll see a much-easier-to-read method for applying a series of functions in the Data Wrangling block.

## 2.6 Functions do not have 'side effects'

We'll finish this chapter with an idea every R user needs to understand to avoid confusion. It relates to how functions modify their arguments, or more accurately, how they **do not** modify their arguments. Take a look at this example:

```r
y <- 3.7
round(y, digits = 0)
```

```
## [1] 4
```

```r
y
```

```
## [1] 3.7
```

We created a variable `y` with the value 3.7, rounded this to a whole number with `round`, printed out the result, and then printed the value of `y`. Notice that **the value of `y` has not changed** after using it as an argument to `round`.

This is important. R functions do not typically alter the values of their arguments (we say 'typically' because there are ways to alter this behaviour if we really want to). This behaviour is captured by the phrase 'functions do not have side effects'.

If we had intended to round the value of `y` so that we can use this new value later on, we have to assign the result of function evaluation, like this:

```r
y <- 3.7
y <- round(y, digits = 0)
```

The reason for pointing out this behaviour is because new R users sometimes assume a function will change its arguments. R functions do not typically do this. If we want to make use of changes, rather than print them to the Console, we need to assign the result a name, either by creating a new variable or overwriting the old one. Remember—functions do not have side effects! Forgetting this creates all kinds of headaches.

# Chapter 3

# Vectors

## 3.1 Introduction

This chapter has three goals. First, we want to learn how to work with a vector, one of the basic structures used to represent data in R-land. Second, we'll learn how to work with vectors by using **numeric** vectors to perform simple calculations. Finally, we'll introduce a couple of different kinds of vectors—**character** vectors and **logical** vectors. This material provides a foundation for working with real data sets in the next block.

> **Data structures?**
>
> The term 'data structure' is used to describe conventions or rules for organising and storing data on a computer. Computer languages use many different kinds of data structures. Fortunately, we only need to learn about a couple of relatively simple ones to use R for data analysis: 'vectors' and 'data frames'. This chapter will consider vectors. The next chapter will look at collections of vectors (a.k.a. data frames).

## 3.2 Atomic vectors

We'll start with a definition, even though it probably won't make much sense yet: a **vector** is a 1-dimensional data structure for storing a set of values, each accessible by its position in the vector. The simplest kind of vectors in R are called **atomic vectors**[1].

There are different kinds of atomic vector, but their defining, common feature

[1] The other common vector is called a "list". Lists are very useful but we won't cover them in this book.

is that it can only contain data of one 'type'. They might contain all integers (e.g. 2, 4, 6, …) or all characters (e.g. "A", "B", "C"), but they can't mix and match integers and characters (e.g. "A", 2, "C", 5).

The word 'atomic' in the name refers to the fact that an atomic vector can't be broken down into anything simpler—they are the simplest kind of data structure R knows. Even when working with a single number we're actually dealing with an atomic vector. Here's the very first expression we evaluated in the Introduction to R chapter:

```
1 + 1
```

```
## [1] 2
```

Look at the output. What is that `[1]` at the beginning? It's actually a clue that the output resulting from `1 + 1` is an atomic vector. We can verify this with the `is.atomic` functions. First, make a variable called `x` with the result of the `1 + 1` calculation:

```
x <- 1 + 1
x
```

```
## [1] 2
```

Then use `is.atomic` to check whether `x` really is an atomic vector:

```
is.atomic(x)
```

```
## [1] TRUE
```

Atomic vectors really are the simplest kind of data structure in R. Unlike many other languages, there is simply no way to represent just a number. Instead, a single number is always stored as a vector of length one[2].

## 3.3 Numeric vectors

A lot of work in R involves **numeric vectors**. After all, data analysis is all about numbers. Here's one simple way to construct a numeric vector (and print it out):

```
numvec <- numeric(length = 50)
numvec
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [39] 0 0 0 0 0 0 0 0 0 0 0 0
```

[2]The same is true for things like sets of characters (`"dog"`, `"cat"`, `"fish"`, …) and logical values (`TRUE` or `FALSE`) discussed in the next two chapters.

What happened? We made a numeric vector with 50 **elements**, each of which is the number 0. The word 'element' is used to refer to the values that reside in a vector.

When we create a vector but don't assign it to a name using `<-` R just prints it for us. Notice what happens when a vector is printed to the screen. Since a length-50 vector can't fit on one line, it was printed over two. At the beginning of each line there is a `[X]`: the number `X` gives the position of the elements printed at the beginning of each line.

If we need to check that we really have made a numeric vector, we can use the `is.numeric`[3] function to do this:

```
is.numeric(numvec)
```

```
## [1] TRUE
```

This confirms `numvec` is numeric by returning `TRUE` (a value of `FALSE` would mean that `numvec` is some other kind of object).

Keep in mind that R won't always print the exact values of the elements of a vector. For example, when R prints a numeric vector, it only prints the elements to 7 significant figures by default. We can see this by printing the built in constant `pi` to the Console:

```
pi
```

```
## [1] 3.141593
```

The actual value stored in `pi` is much more precise than this. We can see this by printing `pi` again using the `print` function:

```
print(pi, digits = 16)
```

```
## [1] 3.141592653589793
```

**Different kinds of numbers**

Roughly speaking, R stores numbers in two different ways depending of whether they are whole numbers ("integers") or numbers containing decimal points ("doubles" – don't ask). We're not going to worry about this difference. Most of the time, the distinction is invisible to users, so it is easier to think in terms of numeric vectors. We can mix and match integers and doubles in R without having to worry about how it is storing the numbers.

[3] This may not look like the most useful function in the world, but sometimes we need functions like `is.numeric` to understand what R is doing or root out mistakes in our scripts.

## 3.4 Constructing numeric vectors

We just saw to make a numeric vector of zeros using the `numeric` function. This is arguably not a particularly useful skill because we usually need to work vectors of particular values (not just 0). A useful function for creating custom vectors is the `c` function. Take a look at this example:

```
c(1.1, 2.3, 4.0, 5.7)
```

```
## [1] 1.1 2.3 4.0 5.7
```

The 'c' in the function name stands for 'combine'. The `c` function takes a variable number of arguments, each of which must be a vector of some kind, and combines these into a single vector. We supplied the `c` function with four arguments, each of which was a single number (i.e. a length-one vector). The `c` function combines these to generate a vector of length 4. Simple.

Now look at this example:

```
vec1 <- c(1.1, 2.3)
vec2 <- c(4.0, 5.7, 3.6)
c(vec1, vec2)
```

```
## [1] 1.1 2.3 4.0 5.7 3.6
```

This shows that we can use the `c` function to combine two or more vectors of any length. We combined a length-2 vector with a length-3 vector to produce a new length-5 vector.

**The `c` function is odd**

Notice that we did not have to name the arguments in those two examples—there were no `=` involved. The `c` function is an example of one of those flexible functions that breaks the simple rules of thumb for using arguments that we set out earlier: it can take a variable number of arguments that do not have predefined names. This behaviour is necessary for `c` to be of any use: to be useful it needs to be flexible enough to take any combination of arguments.

## 3.5 Named vectors

What happens if use named arguments with `c`? Take a look at this:

```
namedv <- c(a = 1, b = 2, c = 3)
namedv
```

```
## a b c
## 1 2 3
```

What happened? R used the argument names to set the names of each element in the vector. The resulting vector is still a 1-dimensional data structure. When printed to the Console, each element's value is printed along with its associated name above it. We can extract the names from a named vector using the `names` function:

```
names(namedv)
```

```
## [1] "a" "b" "c"
```

Being able to name the elements of a vector is useful because it makes it easier to identify and extract the bits we need.

## 3.6 Vectorised operations

All the simple arithmetic operators (e.g. `+` and `-`) and many mathematical functions are **vectorised** in R. When we use a vectorised function it operates on vectors on an element-by-element basis. We'll make a couple of simple vectors to illustrate what we mean:

```
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
x
```

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

```
y <- c(0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0)
y
```

```
##  [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

This constructed two length-10 numeric vectors, called `x` and `y`, where `x` is a sequence from 1 to 6 and `y` is a sequence from 0.1 to 0.6. `x` and `y` are the same length. Now look at what happens when we add these using `+`:

```
x + y
```

```
##  [1]  1.1  2.2  3.3  4.4  5.5  6.6  7.7  8.8  9.9 11.0
```

When R evaluates the expression `x + y` it does this by adding the first element of `x` to the first element of `y`, the second element of `x` to the second element of `y`, and so on, working through all ten elements of `x` and `y`. That's what is meant by a **vectorised** operation.

Vectorisation is implemented in all the standard mathematical functions. For example, the `round` function rounds each element of a numeric vector to the nearest integer by default:

```
round(y)
```

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

The same behaviour is seen with other mathematical functions like `sin`, `cos`, `exp`, and `log`—they apply the relevant function to each element of a numeric vector.

It is important to realise that not all functions are vectorised. For example, the `sum` function takes a vector of numbers and adds them up:

```
sum(y)
```

```
## [1] 5.5
```

Although `sum` obviously works on a numeric vector it is not 'vectorised' in the sense that it works element-by-element to return an output vector of the same length as its main argument. It just returns a single number—the sum total of the elements of its input.

**Vectorisation is not the norm**

R's vectorised behaviour may seem like the obvious thing to do, but most computer languages do not work like this. In other languages, we typically have to write a much more complicated expression to do something so simple. This is one reason R is such a good data analysis language: vectorisation allows us to express repetitious calculations in a simple, intuitive way. This behaviour can save a lot of time.

## 3.7 Other kinds of atomic vectors

The data we collect and analyse are often in the form of numbers. It comes as no surprise, therefore, that we work with numeric vectors a lot in R. Nonetheless, we also need to use other kinds of vectors, either to represent different types of data, or to help us manipulate our data. This section introduces two new types of atomic vector to help us do this: character vectors and logical vectors.

### 3.7.1 Character vectors

Each element of a **character vectors** is what is known as a "character string" (or "string" if we are feeling lazy). That term "character string" refers to a sequence of characters, such as "Treatment 1", "University of Sheffield", "Population Density". A character vector is an atomic vector that stores an ordered collection of one or more character strings.

If we want to construct a character vector in R, we have to place double (") or single (') quotation marks around the characters. For example, we can print the name "Dylan" to the Console like this:

```
"Dylan"
```

```
## [1] "Dylan"
```

Notice the [1]. This shows that what we just printed is an atomic vector of some kind. We know it's a character vector because the output is printed with double quotes around the value. We often need to make simple character vectors containing only one value—for example, to set the values of arguments to a function.

The quotation marks are not optional—they tell R we want to treat whatever is inside them as a literal value. The quoting is important. If we try to do the same thing as above without the quotes, we end up with an error:

```
Dylan
```

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

What happened? When the interpreter sees the word Dylan without quotes it assumes that this must be the name of a variable, so it goes in search of it in the global environment. We haven't made a variable called Dylan, so there is no way to evaluate the expression and R spits out an error to let us know there's a problem.

Character vectors are typically constructed to represent data of some kind. The c function is one starting point for this kind of thing:

```
# make a length-3 character vector
my_name <- c("Dylan", "Zachary", "Childs")
my_name
```

```
## [1] "Dylan"   "Zachary" "Childs"
```

Here we made a length-3 character vector, with elements corresponding to a first name, middle name, and last name. If we want to extract one or more elements from a character vector by their position

Take note, this is **not** equivalent to the above :

```
my_name <- c("Dylan Zachary Childs")
my_name
```

```
## [1] "Dylan Zachary Childs"
```

This length-1 character vector's only element is a single character string containing the first, middle and last name separated by spaces. We didn't even need to use the the `c` function here because we were only ever working with a length-1 character vector. i.e. we could have typed `"Dylan Zachary Childs"` and we would have ended up with exactly the same text printed at the Console.

### 3.7.2 Logical vectors

The elements of **logical vectors** only take two values: `TRUE` or `FALSE`. Don't let the simplicity of logical vectors fool you. They're very useful. As with other kinds of atomic vectors, the `c` function can be used to construct a logical vector:

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

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

So why are logical vectors useful? Their allow us to represent the results of questions such as, "is x greater than y" or "is x equal to y". The results of such comparisons may then be used to carry out various kinds of subsetting operations.

Before we can look at how to use logical vectors to evaluate comparisons, we need to introduce **relational operators**. These sound fancy, but they are very simple: we use relational operators to evaluate the relative value of vector elements. Six are available in R:

- `x < y`: is x less than y?
- `x > y`: is x greater than y?
- `x <= y`: is x less than or equal to y?
- `x >= y`: is x greater than or equal to y?
- `x == y`: is x equal to y?
- `x != y`: is x not equal to y?

The easiest way to understand how these work is by example. We need a couple of numeric variables first:

```
x <- c(11, 12, 13, 14, 15, 16, 17, 18, 19, 20)
y <- c(3, 6, 9, 12, 15, 18, 21, 24, 27, 30)
x
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
y
```

```
##  [1]  3  6  9 12 15 18 21 24 27 30
```

Now, if we need to evaluate and represent a question like, "is x greater than than y", we can use either `<` or `>`:

```
x > y
```

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

The `x > y` expression produces a logical vector, with `TRUE` values associated with elements in `x` are less than `y`, and `FALSE` otherwise. In this example, x is less than y until we reach the value of 15 in each sequence. Notice too that relational operators are vectorised: they work on an element by element basis.

What does the `==` operator do? It compares the elements of two vectors to determine if they are exactly equal:

```
x == y
```

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

The output of this comparison is true only for one element, the number 15, which is at the $5^{th}$ position in both `x` and `y`. The `!=` operator is essentially the opposite of `==`. It identifies cases where two elements are not exactly equal. We could step through each of the different relational operators, but hopefully, they are self-explanatory at this point (if not, experiment with them).

**`=` and `==` are not the same**

If we want to test for equivalence between the elements of two vectors we must use double equals (`==`), not single equals (`=`). Forgetting to use `==` instead of `=` is a very common source of mistakes. The `=` symbol already has a use in R—assigning name-value pairs—so it can't also be used to compare vectors because this would lead to ambiguity in our R scripts. Using `=` when you meant to use `==` is a very common mistake. If you make it, this will lead to all kinds of difficult-to-comprehend problems with your scripts. Try to remember the difference!

# Chapter 4

# Data frames

## 4.1 Introduction

The quick introduction to R chapter introduced the word 'variable' as a short-hand for a named object. For example, we can make a variable called `num_vec` that refers to a simple numeric vector using:

```
num_vec <- c(1.1, 2.3, 4.0, 5.7)
num_vec
```

```
## [1] 1.1 2.3 4.0 5.7
```

When a computer scientist talks about variables they're referring to these sorts of name-value associations.

However, the word 'variable' has a second, more abstract meaning in the world of statistics: it refers to anything we can control or measure. For example, data from an experiment will involve variables whose values describe the experimental conditions (e.g. low temperature vs. high temperature) and the quantities we chose to measure (e.g. enzyme activity). We refer to these kinds of variables as 'statistical variables'.

We'll discuss these statistical variables later on. We're pointing out the dual meaning of the word 'variable' now because we need to work with both interpretations. The duality can be confusing at times, but both meanings are in widespread use, so we just have to get used to them. We try to minimise confusion by using the phrase "statistical variable" when referring to data, rather than R objects.

We're introducing these ideas now because we're going to consider a new type of data object in this chapter—the **data frame**. Real-world data analysis involves collections of related statistical variables. How should we keep a large collection of variables organised? We could work with them individually, but this tends to be error prone. Instead, we need a way to keep related variables together. This is the problem that **data frames** are designed to manage.

## 4.2 Data frames

Data frames are one of R's features that mark it out as particularly good for data analysis. We can think of a data frame as a table-like object with rows and columns. A data frame collects together different statistical variables, storing each of them as a separate column. Related observations are all found in the same row.

We'll think about the columns first.

Each column of a data frame is a vector of some kind. These are usually simple atomic vectors containing numbers or character strings, though it is possible to include more complicated vectors. The critical constraint that a data frame applies is that each vector must have the same length. This is what gives a data frame it table-like structure.

The simplest way to get a feel for data frames is to make one. We can make one 'by hand' using some artificial data describing a made-up experiment. Imagine we had conducted a small experiment to examine biomass and community diversity in six field plots. Three plots were subjected to fertiliser enrichment. The other three plots act as experimental controls.

We could store the data describing this experiment in three vectors:

- `trt` (short for "treatment") shows which experimental manipulation was used in a given plot.
- `bms` (short for "biomass") shows the total biomass measured at the end of the experiment.
- `div` (short for "diversity") shows the number of species present at the end of the experiment.

Here's some R code to generate these three vectors (it doesn't matter what the actual values are, they're made up):

```
trt <- rep(c("Control","Fertilser"), each = 3)
bms <- c(284, 328, 291, 956, 954, 685)
div <- c(8, 12, 11, 8, 4, 5)
```

```
trt
```

```
## [1] "Control"   "Control"   "Control"   "Fertilser" "Fertilser" "Fertilser"
```

```
bms
```

```
## [1] 284 328 291 956 954 685
```

```
div
```

```
## [1]  8 12 11  8  4  5
```

Notice that the information about different observations are linked by their positions in these vectors. For example, the third control plot had a biomass of '291' and a species diversity '11'.

We use the `data.frame` function to construct a data frame from one or more vectors, i.e. to build a data frame from the three vectors we just created:

```
experim.data <- data.frame(trt, bms, div)
experim.data
```

```
##         trt bms div
## 1   Control 284   8
## 2   Control 328  12
## 3   Control 291  11
## 4 Fertilser 956   8
## 5 Fertilser 954   4
## 6 Fertilser 685   5
```

Notice what happens when we print the data frame: it is displayed as though it has rows and columns. That's what we meant when we said a data frame is a table-like structure.

The `data.frame` function takes a variable number of arguments. We used the `trt`, `bms` and `div` vectors, resulting in a data frame with three columns. Each of these vectors has 6 elements, so the resulting data frame has 6 rows. The names of the vectors were used to name its columns. The rows do not have names, but they are numbered to reflect their position.

The words `trt`, `bms` and `div` are not very informative. If we prefer to work with more meaningful column names—which is always a good idea—then we can name the `data.frame` arguments:

```
experim.data <- data.frame(Treatment = trt, Biomass = bms, Diversity = div)
experim.data
```

```
##   Treatment Biomass Diversity
## 1   Control     284         8
## 2   Control     328        12
## 3   Control     291        11
```

```
## 4 Fertilser     956         8
## 5 Fertilser     954         4
## 6 Fertilser     685         5
```

The new data frame contains the same data as the previous one but now the column names correspond to the human-readable words we chose.

**Don't bother with row names**

We can also name the rows of a data frame using the `row.names` argument of the `data.frame` function. We won't bother to show an example of this, though. Why? We can't easily work with the information in row names which means there's not much point adding it. If we need to include row-specific information in a data frame it's best to include an additional variable, i.e. an extra column.

## 4.3 Exploring data frames

The first things to do when presented with a new data set is to explore its structure to understand what we're dealing with. There are plenty of options for doing this when the data are stored in a data frame. For example, the `head` and `tail` functions extract the first and last few rows of a data set:

```
head(experim.data, n = 3)
```

```
##   Treatment Biomass Diversity
## 1   Control     284         8
## 2   Control     328        12
## 3   Control     291        11
```

```
tail(experim.data, n = 3)
```

```
##   Treatment Biomass Diversity
## 4 Fertilser     956         8
## 5 Fertilser     954         4
## 6 Fertilser     685         5
```

Notice that the `n` argument controls the number of rows printed. The `View` function can be used to open up the whole data set in a table- or spreadsheet-like like view:

```
View(experim.data)
```

Exactly what happens when we use `View` depends on how we're interacting with R. When we run it in RStudio a new tab opens up with the data shown inside it.

**`View` only displays the data**

The `View` function is only designed to display a data frame as a table of rows and columns. We can't change the data in any way with the `View` function. We can reorder the way the data are presented, but keep in mind that this won't alter the underlying data.

There are quite a few different R functions that will extract information about a data frame. The `nrow` and `ncol` functions return the number of rows and columns, respectively:

```
nrow(experim.data)
```

```
## [1] 6
```

```
ncol(experim.data)
```

```
## [1] 3
```

The `names` function can be used to extract the column names from a data frame:

```
colnames(experim.data)
```

```
## [1] "Treatment" "Biomass"   "Diversity"
```

The `experim.data` data frame has three columns, so `names` returns a character vector of length three, where each element corresponds to a column name.

## 4.4 Extracting and adding a single variable

Remember, each column of a data frame can be thought of as a variable. Data frames would not be much use if we could not extract and modify the variables they contain. In this section, we will briefly review how to extract a variable. We'll examine ways to manipulate the data within a data frame in later chapters.

One way of extracting a variable from a data frame uses a double square brackets construct, `[[`. For example, we extract the `Biomass` variable from our example data frame with the double square brackets like this:

```
experim.data[["Biomass"]]
```

```
## [1] 284 328 291 956 954 685
```

This prints whatever is in the `Biomass` column to the Console. What kind of object is this? It's a numeric vector:

```
is.numeric(experim.data[["Biomass"]])
```

```
## [1] TRUE
```

See? A data frame is a collection of vectors. Notice that all we did was print the resulting vector to the Console. If we want to do something with this numeric vector we need to assign the result:

```
bmass <- experim.data$Biomass
bmass^2
```

```
## [1]  80656 107584  84681 913936 910116 469225
```

Here, we extracted the `Biomass` variable, assigned it to `bmass`, and then squared this.

Notice that we used `"Biomass"` instead of `Biomass` inside the double square brackets, i.e. we quoted the name of the variable. This is because we want R to treat the word "Biomass" as a literal value. This little detail is important! If we don't quote the name, R will assume that `Biomass` is the name of an object and go in search of it in the global environment. Since we haven't created something called `Biomass`, leaving out the quotes would generate an error:

```
experim.data[[Biomass]]
```

```
## Error in (function(x, i, exact) if (is.matrix(i)) as.matrix(x)[[i]] else .subset2(x
```

The error message is telling us that R can't find a variable called `Biomass`.

The second method for extracting a variable uses the `$` operator. For example, to extract the `Biomass` column from `experim.data`, we use:

```
experim.data$Biomass
```

```
## [1] 284 328 291 956 954 685
```

We use the `$` operator by placing the name of the data frame we want to work with on the left-hand side and the name of the column (i.e. the variable) we want to extract on the right-hand side. Notice that we didn't have to put quotes around the variable name when using the `$` operator. We can do this if we want to—i.e. `experim.data$"Biomass"` also works—but `$` doesn't require it.

**Why is there more than one way to extract variables?**

There's no simple way to answer this question without getting into the details of how R represents data frames. The simple answer is that `$` and `[[` are not strictly equivalent, even though they appear to do much the same thing. The `$` method is a bit easier to read, and people tend to prefer it for interactive data analysis tasks, whereas the `[[` construct tends to be used when we need a bit more flexibility for programming.

# Chapter 5

# Packages

## 5.1 The R package system

The R package system is probably the most important single factor driving increased adoption of R. Packages are used to extend the basic capabilities of R. In his book about R packages Hadley Wickam says,

> Packages are the fundamental units of reproducible R code. They include reusable R functions, the documentation that describes how to use them, and sample data.

An R package is just a collection of folders and files in a standard, well-defined format that bundles together computer code, data, and documentation in a way that is easy to use and share with other users. The computer code might all be R code, but it can also include code written in other languages. Packages provide an R-friendly interface to use this "foreign" code without the need to understand how it works.

The base R distribution it comes with quite a few pre-installed packages. These base R packages represent a very small subset of all available R packages. The majority of these are hosted on a network of web servers around the world collectively know as CRAN: the Comprehensive R Archive Network, pronounced either "see-ran" or "kran".

CRAN is a fairly spartan web site, so it's easy to navigate. The landing page has about a dozen links on the right hand side. Under the *Software* section there is a link called Packages. Near the top of that packages page there is a link called Table of available packages, sorted by name that points to a very long list of all the packages on CRAN. The column on the left shows each package name, followed by a brief description of what the package does on the right. There are 1000s of packages listed there.

## 5.2 Task views

The huge list of packages on the available packages is pretty overwhelming. A more user-friendly view of many R packages can be found on the Task Views page (the link is on the left hand side, under the section labelled *CRAN*). A Task View is basically a curated guide to the packages and functions that are useful for certain disciplines. The Task Views page shows a list of these discipline-specific topics, along with a brief description. For example—

- The Environmentrics Task View contains information about using R to analyse ecological and environmental data.
- The Clinical Trials Task View contains information about using R for the design, monitoring and analysis of data from clinical trials.
- The Medical Image Analysis Task View contains information about packages for working with commercial medical image data.
- The Pharmacokinetic Task View contains information about packages for working with pharmacokinetic (PK) data.

Task views are often a good place to start looking for a new package to support a particular analysis in future projects.

## 5.3 Using packages

Two things need to happen to actually make use of a package. First, we need to ensure that a copy of the folders and files that make up the package are copied to an appropriate folder on our computer. This process of putting the package files into the correct location is called **installing** the package. Second, we need to **load and attach** the package for use in a particular R session. The word "session" refers to the time between when we start up R and close it down again.

It's worth unpacking these two ideas a bit, because packages are a very frequent source of confusion for new users:

- If we don't have a copy of a package's folders and files our computer we can't use it. The process of making this copy is called **installing** the package. It is possible to manually install packages by going to the CRAN website, downloading the package, and then using various tools to install it. We don't recommend using this approach though because it's inefficient and error prone. Instead, use built-in R functions to grab the package from CRAN and install it one step.

- Once we have a copy of the package on our hard drive it will remain there for us to use. We don't need to re-install a packages we plan to use every time we start a new R session. It is worth saying that again, **there is no need to install a package every time we start up R / RStudio**. The only exception to this rule is that a major update to R will sometimes require a complete re-install of the packages. These major updates are fairly infrequent though.

- Installing a package does nothing more than place a copy of the relevant files on our hard drive. If we actually want to use the functions or the data that comes with a package we need to make them available in our current R session. Unlike package installation this **load and attach** process as it's known has to be repeated every time we restart R. If we forget to load up the package we can't use it.

### 5.3.1 Viewing installed packages

We sometimes need to check whether a package is currently installed. RStudio provides a simple, intuitive way to see which packages are installed on our computer. The **Packages** tab in the bottom right pane shows the name of every installed package, a brief description (the same one seen on CRAN) and a version number.

There are also a few R functions that can be used to check whether a package is currently installed. For example, the `find.package` function can do this:

```
find.package("MASS")
```

```
## [1] "/Library/Frameworks/R.framework/Versions/4.0/Resources/library/MASS"
```

This either prints a "file path" showing us where the package is located, or returns an error if the package can't be found. Alternatively, the function called `installed.packages` returns a data frame containing a lot more information about the installed packages.

### 5.3.2 Installing packages

R packages can be installed from a number of different sources. For example, they can be installed from a local file on a computer, from the CRAN repository, or from a different kind of online repository called Github. Although various alternatives to CRAN are becoming more popular, we're only going to worry about installing packages that live on CRAN.

To install a package from an online repository like CRAN we have to first download the package files, possibly uncompress them (like we would a ZIP file), and move them to the correct location. All of this can be done using a single function: `install.packages`. For example, if we want to install a package called **fortunes**, we use:

```
install.packages("fortunes")
```

The quotes are necessary by the way. If everything is working—we have an active internet connection, the package name is valid, and so on—R will briefly pause while it communicates with the CRAN servers, we should see some red text reporting back what's happening, and then we're returned to the prompt.

The red text is just letting us know what R is up to. As long as this text does not include the word "error", there is usually no need to worry about it.

There are a couple of things to keep in mind. First, package names are case sensitive. For example, **fortunes** is not the same as **Fortunes**. Quite often package installations fail because we used the wrong case somewhere in the package name. The other aspect of packages we need to know about is related to **dependencies**: some packages rely on other packages in order to work properly. By default `install.packages` will install these dependencies, so we don't usually have to worry too much about them. Just don't be surprised if the `install.packages` function installs more than one package when only one was requested.

RStudio provides a way of interacting with `install.packages` via point-and-click. The **Packages** tab has an "Install"" button at the top right. Clicking on this brings up a small window with three main fields: "Install from", "Packages", and "Install to Library". We only need to work with the "Packages" field – the other two can be left at their defaults. When we start typing in the first few letters of a package name (e.g. **dplyr**) RStudio provides a list of available packages that match this. After we select the one we want and click the "Install" button, RStudio invokes `install.packages` with the appropriate arguments at the Console for us.

### 5.3.3 Loading and attaching packages

Once we've installed a package or two we'll probably want to actually use them. Two things have to happen to access a package's facilities: the package has to be loaded into memory, and then it has to attached to something called a search path so that R can find it. It is beyond the scope of this book to get in to "how" and "why" of these events. Fortunately, there's no need to worry about these details, as both loading and attaching can be done in a single step with a function called `library`. The `library` function works exactly as we might expect it to. If we want to start using the `fortunes` package—which was just installed above—all we need is:

```
library("fortunes")
```

Nothing much happens if everything is working as it should. R just returns us to the prompt without printing anything to the Console. The difference is that now we can use the functions that **fortunes** provides. As it turns out, there is only one, called fortune:

```
fortune()
```

```
##
## Friends don't let friends use Excel for statistics!
##    -- Jonathan D. Cryer (about problems with using Microsoft Excel for
```

```
##       statistics)
##       JSM 2001, Atlanta (August 2001)
```

The **fortunes** package is either very useful or utterly pointless, depending on ones perspective. It dispenses quotes from various R experts delivered to the venerable R mailing list.

### 5.3.4 Don't use RStudio for loading packages!

As usual, if we don't like working in the Console RStudio can help us out. There is a small button next to each package listed in the **Packages** tab. Packages that have been loaded and attached have a blue check box next to them, whereas this is absent from those that have not. Clicking on an empty check box will load up the package. We mention this because at some point most people realise they can use RStudio to load and attach packages. **We don't recommend using this route**. It's much better to put `library` statements into a script. Why? Because if we rely on RStudio to load packages, we have to do this every time we want to run a script, and if we forget one we need, the script won't work.

### 5.3.5 An analogy

The package system frequently confuses new users. The reason for this stems from the fact that they aren't clear about what the `install.packages` and `library` functions are doing. One way to think about these is by analogy with smartphone "Apps". Think of an R package as analogous to a smartphone App—a package effectively extends what R can do, just as an App extends what a phone can do.

When we want to try out a new App we have to first download it from an App store and install it on our phone. Once it has been downloaded, an App lives permanently on the phone and can be used whenever it's needed. Downloading and installing the App is something we only have to do once. Packages are no different. When we want to use an R package we first have to make sure it is installed on the computer (e.g. using `install.packages`). Installing a package is a 'do once' operation. Once installed, we don't need to install a package again each time we restart R.

In order to actually use an App on our phone we open it up by tapping on its icon. This obviously has to happen every time we want to use the App. The package equivalent of opening a smartphone App is the "load and attach" operation. This is what `library` does. It makes a package available for use in a particular session. We have to use `library` to load the package every time we start a new R session if we plan to access the functions in that package: loading and attaching a package via `library` is a "do every time" operation.

## 5.4 The tidyverse ecosystem of packages

[COMPLETE ME]

## 5.5 Package data

Remember what Hadley Wickam said about packages? "… include reusable R functions, the documentation that describes how to use them, **and sample data**." Many packages include sample data sets for use in examples and package vignettes. Use the `data` function to list the data sets hiding away in packages:

```
data(package = .packages(all.available = TRUE))
```

The mysterious `.packages(all.available = TRUE)` part of this generates a character vector with the names of all the installed packages in it. If we only use `data()` then R only lists the data sets found in a package called `datasets` and any additional packages we have loaded in the current R session.

The `datasets` package is part of the base R distribution. It exists for one reason—to store example data sets. The `datasets` package is automatically loaded when we start R, i.e. there's no need to use `library` to access it, meaning any data stored in this package can be accessed every time we start R.

From the perspective of learning to use R, working with package data is really useful because it allows us to work with 'well-behaved' data sets without having to worry about getting them into R. Importing data is certainly an important skill but it's not necessarily something a new user wants to worry about. We'll use a package data sets to learn about **dplyr** and **ggplot2** later.

# Part II

# Data Wrangling

# Chapter 6

# Data wrangling with dplyr

We describe how to use **readr** and **dplyr** to import and manipulate data in this section.

# Part III

# Exporing Data

# Chapter 7

# Exploratory data analysis with ggplot2

We describe how to use **ggplot2** to visually explore data in this section.

# Appendix A

# Getting help

## A.1 Introduction

R has a comprehensive built-in help system orientated around the base R functions, and every good external package also comes with its own set of **help files**. These provide information about individual package functions and summarise the included data sets. They also sometimes give descriptions of how different parts of the package should be used, and if we're lucky, one or more 'vignettes' that offer a practical demonstration of how to use the package.

We may as well get something out of the way early on. The word 'help' in the phrase 'help file' is a bit of a misnomer. It's more accurate to say R has an extensive **documentation** system. Help files are designed first and foremost to carefully document the different elements of a package, rather than explain how a particular function or the package as whole should be used to achieve a given end. They are aimed more at experienced users than novices. That said, help files often contain useful examples, and many package authors do try to make our life easier by providing functional demonstrations of their package.

It is important to get to grips with the built in help system. It contains a great deal of useful information which we need to really start using R effectively. The road to enlightenment is bumpy though.

## A.2 Browsing the help system

Help files are a little like mini web pages, which means we can navigate among them using hyperlinks. One way to begin browsing the help system uses the `help.start` function:

```
help.start()
```

When we run this function the **Package Index** page should open up in the **Help** tab of the bottom right pane in RStudio. This lists all the packages currently installed on a computer. We can view the help files associated with a package by clicking on the appropriate link. For example, all the functions that come with the base installation of R have a help file associated with them—we can click on the link to the R base package (`base`) to see these.

The packages that we install separately each have their own set of associated help files. We will see how to navigate these in a moment.

The help browser has Forward, Back, and Home buttons, just like a normal web browser. If we get lost in the mire of help pages we can always navigate backward until we get back to a familiar page. Clicking on the home button takes us to a page with three sections:

1. The **Manuals** section looks like it might be useful for novice users. Unfortunately, it's not. Even the "Introduction to R" manual is only helpful for someone who understands what terms like 'data structure' and 'data type' mean. The others are more or less impenetrable unless the reader already knows quite a bit about computing in general.
2. The **Reference** section is a little more helpful. The "Packages" link takes us to the same page opened by `help.start`. From here we can browse the help pages on a package-specific basis. The "Search Engine & Keywords" link takes us to a page we can use to search for specific help pages, either by supplying a search term or by navigating through the different keywords.
3. The **Miscellaneous Material** section has a couple of potentially useful links. The "User Manuals" link lists any user manuals supplied by package authors. The "Frequently Asked Questions" link is definitely worth reviewing at some point, tough again, most of the FAQs are a little difficult for novice users to fully understand.

## A.3 Searching for help files

After browsing help files via `help.start` for a bit it quickly becomes apparent that this way of searching for help is very inefficient. We often know the name of the function we need to use and all we want to do is open that particular help file. We can do this using the `help` function:

```
help(topic = Trig)
```

After we run this line RStudio opens up the help file for the trigonometry topic in the **Help** tab. This file provides information about the various trigonometric functions such as `sin` or `cos` (we'll see how to make sense of such help pages in a bit).

The `help` function needs a minimum of one argument: the name of the topic or function of interest. When we use it like this the help function searches across packages, looking for a help file whose name gives **an exact match** to the name we supplied. In this case, we opened the help file associated with the `Trig` topic.

Most of the time we use the `help` function to find the help page for a specific function, rather than a general topic. This is fine if we can remember the name of the topic associated with different functions. Most of us cannot. Luckily, the help function will also match help pages by the name of the function(s) they cover:

```
help(topic = sin)
```

Here we searched for help on the `sin` function. This is part of the `Trig` topic so `help(topic = sin)` brings up the same page as the `help(topic = Trig)`.

By default, the `help` function only searches for files associated with the base functions or with packages that we have loaded in the current session with the `library` function. If we want to search for help on the `mutate` function—part of the **dplyr** package—but we haven't run `library(dplyr)` in the current session this will fail:

```
help(mutate)
```

Instead, we need tell `help` where to look by setting the `package` argument:

```
help(mutate, package = "dplyr")
```

Even very experienced R users regularly forget how to use the odd function and have to dive into the help. It's for this reason that R has a built in shortcut for `help` accessed via `?`. For example, instead of typing `help(topic = sin)` into the Console we can bring up the help page for the `sin` function by using `?` like this:

```
?sin
```

This is just a convenient shortcut that does the same thing as `help`. The only difference is that `?` does not allow us to set arguments such as `package`.

## A.4 Navigating help files

Navigating help files is a little daunting at first. These files can be quite long and contain a lot of technical jargon. The help files associated with functions—

the most common type—do at least have a consistent structure with a number of distinct sections. Wrestling with a help file is much easier if we at least understand what each section is for. After the title, there are eight sections we need to know about:

1. **Description** gives us a short overview of what the function is meant to be used for. If the help page covers a family of related functions it gives a collective overview of all the functions. Always read this before diving into the rest of the help file.

2. **Usage** shows how the function(s) are meant to be used. It lists each member of the family as well as their common arguments. The argument names are listed on their own if they have no default, or in name-value pairs, where the value gives the default should we choose not to set it ourselves.

3. **Arguments** lists the allowed arguments along with a short description of what they do. This also tells us what what kind of data we're allowed to use with each argument, along with the allowable values (if relevant). Always read this section.

4. **Details** describes precisely how the function(s) behave, often in painful detail. This is often the hardest-to-comprehend section. We can sometimes get away with ignoring this section but when we really want to understand a function we need to wade through it.

5. **Value** explains what kind of object a function returns when it finishes doing whatever it does. We can often possible to guess what this will be from the type of function, but nonetheless, it is a good idea to check whether our reasoning is correct.

6. **References** just lists the key reference(s) for when if we really need to know the 'hows' and 'whys' of a function. We can usually skip this information. The one exception is if the function implements a particular analysis tool. It's best to know how such tools work before trying to use them.

7. **See Also** gives links to the help pages of related functions. These are usually functions that do something similar to the function of interest or are meant to be used in conjunction with it. We can often learn quite a bit about packages or related functions by following the links in this section.

8. **Examples** provides one or more examples of how to use the function. These are stand-alone examples, so there's nothing to stop us running them. This is often the most useful section of all. Seeing a function in action is a very good way to cut through the jargon and understand how it works.

## A.5 Vignettes and demos

The purpose of a package vignette is to provide a relatively brief, practical account of one or more of its features. Not all packages come with vignettes, though the best packages often do. We use the `vignette` function to view all the available vignettes in Rstudio. This will open up a tab that lists each vignette under their associated package name along with a brief description. A package will often have more than one vignette.

If we just want to see the vignettes associated with a particular package, we have to set the `package` argument. For example, to see the vignettes associated with **dplyr** we use:

```
vignette(package = "dplyr")
```

Each vignette has a name (the "topic") and is available either as a PDF or HTML file (or both). We can view a particular vignette by passing the `vignette` function the `package` and `topic` arguments. For example, to view the "grouping" vignette in the **dplyr** package we would use:

```
vignette(topic = "grouping", package = "dplyr")
```

The `vignette` function is fine, but it is more convenient to browse the list of vignettes inside a web browser. This allows us to open a particular vignette directly by clicking on its link, rather than working at the Console. We can use the `browseVignettes` function to do this:

```
browseVignettes()
```

This will open a page in our browser showing the vignettes we can view. As usual, we can narrow our options to a specific package by setting the `package` argument.