teacheR

Teach Yourself or Others R

Adam Rawles

Contents

Chapter 1

teacheR

1.1 Overview

This book is a collection of training materials for an introduction to the R statistical computing programming language. Broken down into chapters, I've aimed to cover most of the basics. Alongside each chapter is a xaringan presentation. This can help for those looking to learn, but can also function as a first step for those looking to begin teaching R but who don't have the time to fully develop their own training modules. Hopefully, all the topics covered in the text version will be covered in the presentation and vice versa, so if you learn visually then you can rest easy knowing that you're not missing out! You can find a link to each presentation on the first page of each chapter.

The book is largely split into two sections. One section ("For Students") that's aimed at those that are entirely new to R. I explain the basics in a way that doesn't require a background in data analysis or computing and you should have a decent understanding of the fundamentals of R if you manage to make it through.

In the second section ("For Teachers"), we look at some of the topics in greater detail, looking at the theory and specifics that underpin what we've learnt in the previous section. For those interested in teaching R to others, this section provides an introduction to the underlying workings of R that can be extremely helpful when questions from your students begin to arise. For example, we look at functions in both sections, however we cover the basics of what a function is and how to use one in the "For Students" section, and how to create functions in the "For Teachers" section.

This is a work in progress, and so I would greatly appreciate any feedback. Anything from typos to content suggestions, feel free to raise a GitHub issue if you feel something should be changed.

1.1.1 Acknowledgements

This book was made possible with the help of those who raised issues and proposed pull requests. With thanks to: [\\@Swapnil-2001](https://github.com/Swapnil-2001)

1.2 About Me

I began using R in my second year of university, during an internship looking at publication bias correction methods. I was under the tutorship of a member of staff who helped me immensely, but I must confess that I have never taken an official course in R, online or in person. I like to think, however, that this is not always a bad thing. Learning from the bottom up and struggling along the way is a fantastic way to acquire knowledge and instills a very important lesson:

You're not going to know everything there is to know about R. Ever. But that's okay.

I'm now 4 years into my R career and I use R every day. With that in mind, I don't think there has ever been a day when I haven't referred to a tutorial, or Stack Overflow, or even just Googled the name of a function that I've used 1000 times before. There is a great repository of knowledge for R and it's one of the things I love most about the R community. So please never feel as though you're an impostor in a world of R gurus. In reality, everyone else is just as lost as you. But if you keep ticking along and never feel that learning something new in R isn't worth your time, you'll end up doing some great things.

And in a roundabout way, that is part of the reason I decided to develop these materials. I don't pretend to be the ultimate R programmer, because I still know what it's like to learn something from the start. And everyone has to start somewhere. So I hope that I can help impart some of the lessons that I've learnt over the 4 years to anyone who's looking to learn R in a way that won't leave you feeling lost.

The only final note I have before we start learning how to use R is another bit of advice:

Don't believe everything you read

Whilst this is probably a good thing to keep in mind for any type of training, I feel it's particularly relevant with R for two reasons. Firstly, when it comes to programming languages, lots of people have opinions. Some are true, most are not. Most things you read are a mix between fact and opinion, so take everything with a pinch of salt. For example, the developers of the `ggplot2` package are fervently against arbitrary second axes and so support for them in `ggplot2` is limited. I also share this view, but that doesn't mean that I'm right - read, learn, but question and make your own mind up.

Secondly, R and particularly all of its packages are prone to change. For this reason, people may make statements relative to one version of R that aren't necessarily true in the future. Things have changed over the years, and so answers from a 10 year-old Stack Overflow question may not still be true when you come across them. A microcosmic version of this are some recent changes in the `tidyr` package. Historically, converting data from/to long and short formats was done using the `spread()` and `gather()` functions. However, in newer releases, these functions are deprecated in favour of `pivot_wider()` and `pivot_longer()`, which provide the same functionality but also some extra bits. The practical implication of this suggestion is don't always read one tutorial on a subject before you dive in.

Chapter 2

Introduction

This chapter gives a brief intro into what R is and whether you should use it. For many of you, you may already be aware of the vast majority of this information, but for those who are brand new to R or even to data analysis and programming, this is a good place to start.

2.1 What is R?

R is a public licence programming language. More specifically, it's a statistical programming language meaning that it's often used for statistical analysis rather than software development. R is also a functional programming, rather than an object-oriented programming language like Python. This means that operation in R are primarily performed by functions (input, do something, output), but more about that later.

Strictly speaking, R is not just a functional programming language. In reality, a language is never purely one type and R is no exception. There are object-oriented systems in R (three main ones), meaning that object-oriented programming is possible and relatively straightforward in R.

So basically, R is a functional programming language with some object-oriented systems. If that means very little to you, don't worry. For the vast majority of users, this is a purely academic definition.

One important attribute about R however that may affect you, is that R is an interpreted language. This essentially means that when you send someone some R code, they need R installed to be able to run it. This means that making full programs is difficult. In the opeRate book that follows this one, we'll look at the `shiny` package, which can be used to quickly make web apps based on R code. These apps are no different in the sense that they also need R installed to

be able to run, but because they are web-based, they are significantly easier to share.

For the most part then, if you want to share R code with colleagues, they'll need to have R installed as well.

2.2 Should I use R?

People will forever argue about which is better, R or Python or Java or C or writing down mathematical equations on a piece of paper and handing it to a monkey to solve. I imagine you're reading this because you heard that R was good for data analysis, and it absolutely is. And so is Python. They're just... different. Personally, I prefer to use R but I understand that other people don't.

Importantly though, never feel as though you've missed a trick by picking a particular language. Programming is not just a practice, it's a way of thinking, and experience is almost always transferable across languages.

For reference however, here are a few of the things that you can use R for:

- data analysis
- statistics and machine learning
- reporting and writing technical documents
- web apps
- text analysis

If you're interested in any of these, then you're in the right place.

2.3 Using R

R is very simple. There is a console where you type commands and get responses. Like the classic command-line interfaces you see when the stereotypical nerd has to hack into the FBI database, you type commands, one at a time into the console, R processes it, and then produces a response if appropriate. For example, if you type `2 + 2` into the R console and hit enter, you'll get `4`.

Writing commands out one at a time can be quite time-consuming if you want to make changes however. So we use scripts to store multiple lines of code that can then be run altogether. When you execute a script, each line gets passed one by one to the console and executed. For example, I might make a script with this code:

```
variable1 <- 2 + 2
variable1 / 10
```

```
## [1] 0.4
```

So when I run the script, it will run the first line, then the second without me having to type anything else in.

2.4 RStudio

RStudio is separate from R. R is a programming language and RStudio is an integrated development environment or *IDE*. This means that RStudio doesn't actually run any code, it just passes it to R for you, meaning that you'll need R to really use RStudio.

RStudio is a massive part of how you interact with R however. For example, with the exception of a few days when I was waiting for RStudio to be installed, I can't ever remember using R without RStudio.

In the previous section, we talked very briefly of the R console and scripts. RStudio helps with this workflow. It makes it easier to create scripts, providing extra tools to help write code quicker, and then acts as a window to R when you want to execute the script.

2.4.1 What is an IDE?

At its simplest definition, an IDE helps you get work done in your programming language of choice. It can help you save blocks of code, organise projects, save plots and everything in between. R comes with a basic user interface when you install it, but RStudio provides lot more functionality to help you interact with the R console.

2.4.2 RStudio Panes

TO DO

2.5 Packages

As we know, R is a functional programming language, meaning that we rely on functions to do our work for us. And when you install R, you'll have access to thousands of functions that come bundled with it. However, these functions have been chosen because they're more generalisable and basic. Including functions for everything that could be done in R with the base version would result in it being unnecessarily large.

So instead of them being available from the start, people create sets of functions that usually are used for a particularly tasks and then distribute them as a

package. You can install this package and have access to all these great functions that someone has written for you.

Some great examples of R packages are:

- ggplot2 for creating plots
- dplyr for data manipulation
- shiny for creating web apps
- BMRSr for extracting energy data
  - Truth be told, this isn't a *great* example of a package, it's just the one I've made so that's why it's here.

The thing to remember is that R has a fantastic open source community and if you need to do something in R, somebody has probably written a package to help you out.

2.5.1 Installing packages

You can think of installing a package as a bit like installing a program on your computer. You only need to install it once, but then you'll need to open it each time you use it.

Installing packages is really easy; you just use the `install.packages()` function:

```
install.packages("BMRSr")
```

You can choose where the package installs by supplying a path to the `lib` parameter (e.g. `lib = "C:/me/desktop"`), but by default it will install it into your default library folder. You can find the path to this default library folder with the `.libPaths()` function.

Once the package has been installed, you only need to reinstall it if there's a problem or you want to update it. Otherwise, you just need to load it every time you want to use it. The logic behind this is that you may have hundreds of packages installed and you don't need all of them for every project you do. So instead, we load specific packages we want each time.

To load a package, use the `library()` function. Place this somewhere near the start of your script so that it's obvious which packages someone will need if they're reading your code and want to do it for themselves. This will then load in all of the functions from that package for you to use.

But Adam, what happens if I load two packages that have functions with the same name? Ah, that's a great question. Later in the book ([#Environments]) we're going to look at exactly how R deals with this issues, but I'll give a simple explanation for now. When R loads your packages, it will do it in a specific order. The later on the package is loaded, the higher it's precedence. That means that when you try and use a function with a name that's in more than one of the packages you've loaded, it will default to the latest package you've loaded.

To help avoid these situations, there are some things in place. Firstly, when you load a package that includes function names that are already used, you'll get a conflict warning. Secondly, to avoid this confusion altogether, you can be explicit about which package your function came from. To do that, just prefix your function with the package name and `::` when you use it, like this:

```
dplyr::mutate()

mutate()
```

Finally, there is a package called `conflicted` that you can use to avoid these issues. Rather than giving a certain package precedence because it was loaded later, attempting to use a function that could come from more than one package will cause an error and you'll need to be explicit with which one you mean.

Personally, I use the prefix method. Yes it's a little bit more verbose, but it makes it 100 times easier for someone else to know exactly where your function has come from without the need for extra packages. This also tends to be the approach that I take because there's nothing worse than coming back to a script a year later and forgetting which packages you need because you didn't include the correct library calls.

Chapter 3

For Students

This section of the book is aimed at those who are either complete beginners to data analysis and statistical computing or who have some prior experience with another language or software package and they want to learn more about R.

Here, we'll take a look at the basics but without going into too much detail so as to be confusing. If you make it through this section, when you come out the other side you'll have more than enough knowledge to be able to complete your first real R project.

For those of you that feel you want to understand what underpins the concepts and code we're going to look at or those who want to eventually teach R to others, the following section ("For Teachers") will go into more detail. For example, you'll need to understand what a function is to to use R, but you won't need to know how to create one. In the "For Students" section, we look at what a function is, and then we learn how to create one in the "For Teachers" section.

3.1 Operators

Operators perform an action or represent something. For example, a great example of an operator is `+`. The `+` is a type of arithmetic operator that adds things together.

In this section, we're going to look at the more common arithmetic operators that are used for simple maths in R, and then at some logical operators that are used to evaluate whether a criteria has been fulfilled.

3.1.1 Arithmetic operators

At the base of lots of programming languages are the arithmetic operators. These are your symbols that perform things like addition, subtraction, multiplication, etc. Because these operations are so ubiquitous however, the symbols that are used are often very similar across languages, so if you've used Excel or Python or SPSS or anything similar before, then these should be fairly straightforward.

Here are the main operators in use:

```
2 + 2  # addition
```

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

```
10 - 5 # subtraction
```

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

```
5 * 4 # multiplication
```

```
## [1] 20
```

```
100 / 25 # division
```

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

3.1.2 Logical operators

Logical operators are slightly different to arithmetic operators - they are used to evaluate a particular criteria. For example, are two values equal. Or, are two values equal *and* two other values different.

To compare whether two things are equal, we use two equal signs (`==`) in R:

```
1 == 1 # equal
```

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

Why two I hear you say? Well, a bit later on we'll see that we use a single equals sign for something else.

To compare whether two things are different (not equal), we use `!=`:

```
1 != 2 # not equal
```

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

The `!` sign is also used in other types of criteria, so the best way to think about it is that it inverts the criteria you're testing. So in this case, it's inverting the "equals" criteria, making it "not equal".

Testing whether a value is smaller or larger than another is done with the `<` and `>` operators:

```
2 > 1  # greater than
```

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

```
2 < 4 # less than
```

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

Applying our logic with the `!` sign, we can also test whether something is *not* smaller or *not* larger:

```
1 >! 2 # not greater than
```

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

```
2 <! 4 # not less than
```

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

Why is the `!` sign before the equals sign in the "not equal" to code, but after the "less than/greater than" sign? No idea. It'd probably make more sense if they were the same, but I suppose worse things happen at sea.

There are three more logical operators, and they are the "and", "or", and "xor" operators. These are used to test whether at least one or more than one or only one of the logical comparisons are true or false:

```
1 == 1 | 2 == 3 # or (i.e. are either of these TRUE)
```

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

```
1 == 1 & 2 == 3 # and (i.e. are these both TRUE)
```

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

The xor operator is a bit different:

```
xor(1 == 1, 2 == 3) # TRUE because only 1 is
```

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

```
xor(1 == 1, 2 == 2) # FALSE because both are
```

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

For `xor()`, you need to provide your criteria in brackets, but this will make much more sense once we look at functions.

3.2 Variable assignment

Do you ever tell a story to a friend, and then someone else walks in once you've finished and so you have to tell the whole thing again?

Well, imagine after the second friend walks in, another friend comes in, and you have to start the story over again, and then another friend comes in and so on and so forth. What would be the best way to save you repeating yourself? As weird as it would look, if you wrote the story down then anyone who came in could just read it, rather than you having to go through the effort of explaining the whole thing each time.

This is essentially what we can do in R. Sometimes you'll use the same value again and again in your script. For example, say you're looking at total expenditure over a year, the value for the amount spent would probably come up quite a lot. Now, you could just type that value in every time you need it, but what happens if the value changed? You'd then have to go through and change it every time it appears.

Instead, you could store the value in a variable, and then reference the variable every time you need it. This way, if you ever have to change the value, you only need to change it once.

3.2.1 Creating variables

Creating variables in R is really easy. All you need to do is provide a valid name, use the `<-` symbol, and then provide a value to assign:

```
hello_im_a_variable <- 100
hello_im_a_variable
```

```
## [1] 100
```

Now, whenever you want to use your variable, you just need to provide the variable name in place of the value:

```
hello_im_a_variable / 10
```

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

You can even use your variable to create new variables:

```
hello_im_another_variable <- hello_im_a_variable / 20
hello_im_another_variable
```

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

When you come across other people's work, you may see that they use `=` instead of `<-` when they create their variables. Even though it's not the end of the world if you do do that, I would recommend getting into the habit of using `<-`. `<-` is purely used for assignment, whereas `=` is actually also used when we call functions, and so it can get a bit confusing if you use them interchangeably.

As a side note, you'll see that the value of the variable isn't outputted when we assign it. If we want to see the value, we need just the name.

3.2.2 Naming

Naming objects and variables in R mostly comes down to preference. There are some hard and fast rules that need to be followed which we'll discuss and also a few common naming conventions but which one you use is up to you.

3.2.2.1 Valid names

R is pretty lenient when it comes to names but there are some red lines:

- Names must start with a character or a dot (but then the second character can't be a digit)
- Names can only contain letters, numbers, underscores, and dots

Similar to this, there are some reserved words that can't be used as object names:

- break
- if
- else
- FALSE
- TRUE
- for
- function
- Inf
- NaN
- NA
- next
- repeat
- return
- while

As a sidenote, names are case sensitive. That means that you can have two objects called `test` and `Test` that can be referred to separately. Generally, this isn't the best idea.

3.2.2.2 Naming conventions

3.2.2.2.1 Nouns and verbs

Roughly speaking, it's advisable to name your variables as nouns and your functions (which get to later) as verbs. This is because variables can be considered *things* whereas functions *do things*.

For example, an appropriate name for the energy-based dataset you're working on might be `energy_dataset`. This is descriptive and unique. An example of good function names are the `sum()` and `mean()` functions; what they do is easily disseminated from their names.

3.2.2.2.2 Multiple words

Sometimes, you'll want to use names that have more than one word, like our `energy_dataset` example. If you're convinced that the best way to do this is to include an actual space, you can create objects with spaces in their names by surrounding the name in backticks ‘:

```
`dont call your variable this` <- 1
```

Please don't ever do this. It will just make things 100% more complicated down the line. Instead, I highly recommend that you use camel case (`EnergyDataset`), `_`s (`energy_dataset`) or `.`s (`energy_dataset`).

Personally, I use `_`s because camel case is more difficult to read at a glance and there is a group of R functions that use `.` in their name and you don't want to get confused with those, but it's really just a preference. I would only say that it's better to be consistent than to choose the right convention.

3.2.3 Reassigning variables

Variables are very flexible. You can overwrite a previously defined variable just be reassigning a new value to the same name:

```
variable_1 <- 100
variable_1 <- "I'm not 100 anymore"
variable_1
```

```
## [1] "I'm not 100 anymore"
```

R will also give the variable an appropriate *type* based on the value you assign. So for example, if you assign `20` to a variable, then that variable will be stored as a number. If you assign something in quotation marks like `"hello"`, then R will store it as text.

Let's look in a bit more detail at the different data types...

3.3 Data types

Data can be stored in lots of different forms. For example, `"TRUE"` and `TRUE` are stored as two different types, even though they look very similar to us.

The main different data types are:

- logical
  - TRUE
  - FALSE
- double (numeric)

  - 12.5
  - 19
  - 99999

- integer (numeric)
  - 2L
  - 34L
- character
  - "hello"
  - "my name is"
- factors
- dates
  - 2019-06-01
- datetime (POSIXct)
  - 2019-06-01 12:00:00

Let's have a look at each one in detail:

3.3.1 Logical

A logical variable can only have two *real* values, `TRUE` or `FALSE`. I say two *real* values, because you can also have things like `NA`, but that's true of any data type.

Logical variables are used a lot in response questionnaires, where the answer to the question is either "Yes" or "No" (`TRUE` or `FALSE`). I would recommend converting any character strings like "Yes" or "No" or "TRUE" or "FALSE" to a logical variable rather than leaving them as characters, because it'll make your analysis less verbose (use fewer lines of code), even if it doesn't change the underlying logic.

To test whether something is stored as logical, we use the `is.logical()` function:

```
is.logical(TRUE)
```

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

```
is.logical("TRUE")
```

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

To convert a value to logical, use the `as.logical()` function:

```
as.logical(1)
```

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

```
as.logical(0)
```

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

```
as.logical("TRUE")
```

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

```
as.logical("FALSE")
```

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

Be careful though, just because a conversion seems obvious to you, doesn't mean you'll get the expected result! For instance, what do you think `as.logical(2)` should return? See for yourself.

3.3.2 Double

The best way to think of a `double` value is as a number. It can be a whole number (but see Integers) or a decimal. R will often take care of any implicit number conversion that needs to be done under the hood, so the only thing you really need to keep in mind is that when you assign a number, be it a whole number or decimal, it will be stored as double by default.

As an aside, it's called double because it's stored using double precision.

To check whether a value is stored as double (or more generally numeric), use the `is.double()` and `is.numeric()` functions:

```
is.double(2)
```

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

```
is.numeric("not numeric")
```

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

```
is.double(2L) # see the next section for why this returns FALSE
```

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

To convert a value to a double, use the `as.double()` or `as.numeric()` functions:

```
as.double("5")
```

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

```
as.numeric("10")
```

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

```r
as.double("im going to cause a warning")
```

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

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

3.3.3 Integer

Whilst also storing numeric data (like double), integers are specific to whole numbers. Also, by default, even when you assign a whole number, like this: `number <- 1`, R will store that value as double rather than as an integer. To store something explicitly as an integer, suffix the value with an `L`, like this: `number <- 1L`. Attempting to store something that isn't an integer as an integer will result in a warning:

```r
1.5L
```

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

For the most part, I let R take care of how it stores numbers, unless I explicitly need it to be of a certain type. This is pretty rare though.

To check if something is an integer, use the `is.integer()` function:

```r
is.integer(2)
```

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

```r
is.integer(2L)
```

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

To convert to an integer, use the `L` suffix or the `as.integer()` function:

```r
1L
```

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

3.3.4 Character

Sometimes called characters, or character strings, or just strings, characters store text. If you assign a value within quotation marks, regardless of what's inside the quotation marks, it will be stored as character. For example, `"5"` stores a character string with the text "5", not the number 5. This is particularly important when you want to start combining variables. For example, `{r, error = TRUE} "5" + 5` doesn't work, because you're trying to add text to a number, which doesn't make sense.

To check whether something is stored as a character, use the `is.character()` function:

```
is.character("hello")
```

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

```
is.character(5)
```

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

```
is.character(TRUE)
```

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

To convert something to a character, use the `as.character()` function:

```
as.character(5)
```

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

```
as.character(TRUE)
```

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

3.3.5 Factors

Factors are a unique but useful data type in R. Essentially, factors store different levels that represent some sort of grouping. For example, say you were collecting some information on people from different countries, the column that holds which country the respondent is from could be stored as a factor, with the levels England, Spain, France, etc.

A factor level is made up of two things. A label and a number that represents that group. For example, in my countries example, our factor would have the labels "England", "Spain", "France" and the values 1, 2, 3. This means that internally, a factor is essentially a collection of integers representing the level position and character strings representing the level label.

To create a factor, we just use the `factor()` function:

```
factor(c("England", "France", "Spain"))
```

```
## [1] England France  Spain
## Levels: England France Spain
```

It's also worth remembering that you can have levels that don't appear in the data you have. For example, in a questionnaire, you may provide the options "None", "Some", "All". But in your responses, you may see that no one chose the "None" option. In that case, you would still create a factor with three levels, even though only two of them appear.

You can also specify whether a factor is *ordered*. You would use an ordered factor when the levels have meaningful order. For instance, in the above example,

it would make sense that "Some" is better than "None", and "All" is better than "Some". To create an ordered factor, just specify `ordered = TRUE` in your function. By default, the factor will be ordered in the ordered the values appear, unless you specify levels (see below).

To convert something to a factor, use the `factor()` function if you want to specify levels and labels, or `as.factor()` to do it for you:

```
factor(c("Some", "All"), levels = c("None", "Some", "All"))
```

```
## [1] Some All
## Levels: None Some All
```

```
factor(c("Some", "All"), levels = c("None", "Some", "All"), ordered = TRUE)
```

```
## [1] Some All
## Levels: None < Some < All
```

```
as.factor(c("Some", "All"))
```

```
## [1] Some All
## Levels: All Some
```

Notice the difference in the output of those three lines. The first allows us to specify the levels (i.e. the values that were possible). The second does the same but we also specify the ordering of the levels, and the third just converts the provided values and generates the levels based on that data.

Note: An important change in R version 4.0.0 is that R will no longer automatically convert strings (characters) to factors when you import data using `data.frame()` or `read.table()`. Prior to 4.0.0, it would automatically convert strings to characters unless otherwise specified.

3.3.5.1 Converting from factors

Sometimes you'll need to convert data from a factor to something else, usually a character. This is fairly straightforward using the tools we've already seen:

```
as.character(factor(c("Some", "None", "All")))
```

```
## [1] "Some" "None" "All"
```

3.3.6 Dates

Dates in any language are tricky. Different countries store dates in different formats and different bits of software stores dates in different ways (looking at you Excel). This can make storing values as dates tough.

The most common way of creating a date is to use the `as.Date()` function. To use this function, you just need to provide your date as a character string:

```
as.Date("2019/01/01")
```

```
## [1] "2019-01-01"
```

But Adam, how does R know which one is the month and which is the day? Good question, thank you for asking. By default, R expects your character string to be in the order "Year/Month/Day". If you don't provide it in that format, you'll get a nonsense output:

```
as.Date("01/12/2019")
```

```
## [1] "1-12-20"
```

If your data is in a different format however, you can specify the format:

```
as.Date("01/12/2019", format = "%d/%m/%Y")
```

```
## [1] "2019-12-01"
```

Here, we're telling R that the string is in the format "Day/Month/Year". A list of the different codes that can be used in the format parameter can be found here, or by typing "R date codes" into Google.

Because nothing in life is simple, sometimes you'll get some data that has the date stored as a number. This is because the source of that data has the date stored as the number of days that have passed since an origin date. Because it's a number, our `as.Date(..., format = ...)` doesn't work. Instead, we can still use the `as.Date()` function, but we need to specify what the origin date is that the number refers to.

By default, when importing from Excel in Windows, the origin date is December 31st 1899. More commonly, the date January 1st 1970 (also known as the epoch date) is used.

Anyway, to specify your origin, we use the `origin` parameter, like this:

```
as.Date(18262, origin = "1970/01/01")
```

```
## [1] "2020-01-01"
```

Notice the format I've provided the origin in. It's the same as the default that R expects, and I would recommend copying that format wherever possible. If you're someone who just wants to watch the world burn, then you can specify a format for your origin as well...'

```
as.Date(18262, origin = as.Date("01/01/1970", format = "%d/%m/%Y"))
```

```
## [1] "2020-01-01"
```

but where's the humanity in that?

Testing whether something is a date is not as simple as the other data types unfortunately. Instead, we just use the `is()` or `class()` functions. If the first value returned is "Date", then you know it's a date:

```
is(as.Date("2020/01/01"))
```

```
## [1] "Date"     "oldClass"
```

```
class(as.Date("2020/01/01"))
```

```
## [1] "Date"
```

3.3.7 Datetimes (POSIXct)

If you thought dates were annoying, datetimes are like dates' little brother who didn't get enough attention as a child and so acts up all the time. One of the reasons for this is that datetimes aren't actually called datetimes. They're called POSIXct in R. So whenever you see that dreadful word, just remember "ah, Adam told me that means datetime" and you'll be fine.

Another thing that makes datetimes tough is that in addition to dates, datetimes (as you may have guessed) also store the time. The issue with that is that time is a more relative concept - there are lots of different time zones, so how do you know which one you're referring to. By default, R has a locale for where you currently are and will use that location for your timezone. You override that default using the `Sys.setlocale()` function, or you can use the `tz` parameter when creating your datetime as we'll see below.

With these annoyances aside however, creating datetimes isn't all that different to creating dates except that we use the `as.POSIXct()` function instead. We just provide a character string (with a `format` specification if necessary), or a number with an origin. One important departure from dates though, is that now our origin is in seconds, not days, to allow us to calculate the time.

```
as.POSIXct("2020/01/01 12:00:00")
```

```
## [1] "2020-01-01 12:00:00 UTC"
```

```
as.POSIXct("2020/01/01 12:00:00", tz = "NZ")
```

```
## [1] "2020-01-01 12:00:00 NZDT"
```

```
as.POSIXct(1577880000, origin = "1970/01/01")
```

```
## [1] "2020-01-01 12:00:00 UTC"
```

Similar to dates, there is no `as.POSIXct()` function in base R, so we use the `is()` and `class()` functions instead:

```
is(as.POSIXct("2020/01/01 12:00:00"))
```

```
## [1] "POSIXct"  "POSIXt"   "oldClass"
```

```
class(as.POSIXct("2020/01/01 12:00:00"))
```

```
## [1] "POSIXct" "POSIXt"
```

3.3.8 NA and NULL

The R language has two closely related values, **NA** and **NULL**.

NULL indicates the absence of a value. It means that a value is missing (or has length zero). A null value has no 'type' because it represents an absence of something, so passing a null value to any of the `is.[type]()` functions will return FALSE. Instead, checking whether a value is NULL is done with the `is.null()` function:

```
is.character(NULL)
```

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

```
is.null(NULL)
```

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

```
"NULL" == NULL
```

```
## logical(0)
```

On the other hand, *NA* (not available) represents an invalid value. This most often occurs when you try to convert one datatype to another where R can't assign an appropriate value.

For example, attempting to parse a character string to a date format that doesn't match will result in a NA value:

```
as.Date("10/01/20", format = "%m%Y%d")
```

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

R has tried to parse a value from one type into another. The value isn't NULL because it clearly isn't missing, but it couldn't be converted to a Date type, so it's NA. Unlike NULL, there are different NA values for each datatype (although they'll all look like NA in the console). In the example above, we actually created a NA that has the Date type:

```
errant_date <- as.Date("10/01/20", format = "%m%Y%d")
is(errant_date)
```

```
## [1] "Date"     "oldClass"
```

This is because R knows what type the NA *should* be, but it couldn't assign in a proper value.

The same behaviour can be observed for other data types:

```
as.numeric("not a number")
```

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

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

```
as.logical("not a logical")
```

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

```
is(as.numeric("not a number"))
```

```
## Warning in is(as.numeric("not a number")): NAs introduced by coercion
```

```
## [1] "numeric" "vector"
```

```
is(as.logical("not a logical"))
```

```
## [1] "logical" "vector"
```

To test whether something is NA, we use the `is.na()` function. You don't need to worry about what type the NA is, this will test if it is an NA of any type.

```
is.na(NA_character_) # this will be an NA that is of type 'character'
```

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

```
is.na(NA_integer_) # this will be an NA that is of type 'integer'
```

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

3.3.8.1 Dealing with NAs

Dealing with NAs is often contextual. Attempting to perform a mathematical calculation on a vector of values that contains at least one NA will often return NA:

```
sum(1,2,NA)
```

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

```
mean(c(1,NA))
```

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

```
NA + 1
```

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

In some cases, those NAs will represent real issues with the values and so removing the NAs or converting them to 0 will just mask the error without fixing it.

Alternatively, data imports can often return NA values because of differing data types or similar and so converting those values to 0 or removing them outright may be appropriate.

Ultimately, how you deal with NA values is a question that you'll need to answer when it happens and depending on the situation. I will give you a helping hand though and say that if you want to just remove the NA values when summing or calculating an average or similar, then these functions often have an `na.rm` parameter than can be used to remove the NA values from the supplied list of values:

```
sum(1,2,NA, na.rm = TRUE)
```

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

3.3.9 NaN and Infinity

A special case is NaN (not a number). NaNs are distinct to NA in that they represent a valid value. More precisely, NaN represents not real numbers (numeric values that cannot be represented with numbers). For example, dividing 0/0.

`Inf` and `-Inf` are similar constructs. They represent infinity and minus infinity respectively. They are valid values but they are not representable with numbers, so they have their own reserved words.

`NaN`, `Inf` and `-Inf` are all of the numeric type and do not have equivalent values in other data types.

3.4 Data structures

It's rare that you're ever going to be working on a single value in R. Instead, you're going to want to work on collections of values, like a dataset or a list or something similar. So we need to know what data structures are available in R. Here's a list of the structures which we'll go into more detail:

- vectors
- lists
- matrices
- data frames

3.4.1 Vectors

Vectors are simple arrays of data in a single dimension. You can think of vectors as a like a very simple list. For instance, you can store the numbers 1 to 10 in a vector. Or each word in a string of text could be stored in a vector.

One of the main things to remember about vectors however, is that they are atomic. That's basically a fancy word to mean that each value in a vector must be a single unit. For instance, the number `1` is a single unit. But a vector containing all the numbers between 1 and 10 is not. This is in direct contrast to lists, which are recursive, and we'll look at those next.

To create a vector, we use the `c()` function, which is short for concatenate. In other words, we're pulling together lots of different values and concatenating them into one structure.

```
c(1,2,3,4)
```

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

Technically speaking, even single values are stored as a vector in R, they just have length one. That's why if you type `is(1)`, the second things that pops up after "numeric" is "vector". R is telling us that `1` is a number and that it's also a vector.

All the values in a vector must be of the same type (e.g. character, numeric, etc.), but you can name the values in a vector. To give a value a name, you can simply provide one with a `=` sign when you create your vector:

```
c(this_is_the_first_value = 1, this_is_the_second_value = 2)
```

```
##  this_is_the_first_value this_is_the_second_value
##                        1                        2
```

3.4.2 Lists

Lists are similar to vectors in that they store values one after another. However, there are two main differences:

- Lists can contain values of any type - they are *recursive*.

Recursion is the action of doing something again and again. We call lists recursive, because we could have a list, that contains a list, that contains a list, and so on and so forth like Russian Dolls.

- Lists do not have to be made up of values of the same type.

So for instance, whilst a vector must always be the same, like `c(1,2,3)`, we could have a list that looks like this:

```
list(1, "hello", TRUE)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "hello"
```

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

As you just saw, we create lists using the `list()` function, and providing names is done the same way as it is for vectors:

```
list(
  first_value = 1,
  second_value = "hello"
)
```

```
## $first_value
## [1] 1
##
## $second_value
## [1] "hello"
```

3.4.3 Lists vs Vectors

Given that lists and vectors are intrinsically linked, it's very natural to wonder when to use on over the other. Well, the basic answer is to use whichever one has the requirements you need. If all of your values are of the same type and are atomic (numeric, integer, logical, etc.). If they aren't all the same, or you need to have a list of data structures like vectors and lists rather than just single values, then use a list.

I appreciate that this answer isn't particularly satisfactory however, and so let me give a real life example of when I've used each.

Vector

I was recently producing a simulation that I needed to run multiple times with a different value each time. The value itself was a single number ranging between 1 and 30. So I used a vector like so:

```
my_vector <- c(1:30)
# this is just shorthand for saying "all of the numbers from 1 to 30"
```

So when I ran my simulation, I had all the values I wanted to run it for in a single structure.

List

When doing data modelling, it can sometimes be helpful to create and evaluate multiple models. One way of doing that is to create multiple models and assign them to different variables:

```
model1 <- model(...)
model2 <- model(...)
```

```
model3 <- model(...)
```

The problem with this however, is that if I then want to compare the models, I'll have to write out `modelX` each time. If I have 50 models or similar, it may take a while. So instead, I often store all my models in a list. The values are complex (i.e. a model isn't just a numeric or character value) so they can't be stored in a vector, but they can be stored in a list. This means that I keep all my models together, and if I then decide that actually I want to add more models to my list, this is significantly easier than typing out more `modelX <- model(...)` lines and assigning each one as a new variable.

Before moving onto the other data structures, I just want to quickly mention that in my learning experience, understanding vectors and lists is one of the most important parts of getting to grips with R. R for many is about automating analysis and reducing the amount of time taken to do something. And vectors and lists are at the heart of this. Later on, we'll look at functions and for loops, which we can use to perform the same action or calculation on all the values in a list or vector. Together, these will be your strongest R tools.

3.4.4 Matrices

Unlike vectors, matrices are 2 dimensional. In fact, matrices resemble something a bit like a watered down version of a spreadsheet or table.

I say watered down, because matrices can only contain values of the same type (like vectors). This means that storing complex datasets in matrices isn't really very easy. Instead, matrices are an efficient way of storing and performing matrix mathematics on sets of numbers.

Creating a matrix is easy using the `matrix()` function. We provide the values we want to put into the matrix, and how many rows and columns they should be split into:

```
matrix(c(1:4), nrow = 2, ncol = 2)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

By default, the matrix is filled by column first (i.e. it starts at column 1 and fills that column, then moves onto the next one). To change this, use `byrow = TRUE`.

3.4.5 Dataframes

Dataframes are the more typical dataset storage medium. They can have columns of different types (although all of types within a column need to be the same), and they resemble more of an Excel spreadsheet than matrices.

To create a dataframe, we use the `data.frame()` function. To this function, we provide our values as columns:

```
data.frame(col_1 = c(1,2,3),
           col_2 = c("hello", "world", "howsitgoing"))
```

```
##   col_1       col_2
## 1     1       hello
## 2     2       world
## 3     3 howsitgoing
```

More specifically, R stores dataframes as essentially a list of lists, with each list representing a different column. To demonstrate this, when we type...

```
is(data.frame())
```

```
## [1] "data.frame" "list"       "oldClass"   "vector"
```

The second value in the returned vector is "list".

So at it's heart, a dataframe is a list, and each column within a dataframe is also a list. Why is that useful to know? Well, for one, this should make things make a bit more sense when we move onto subsetting. Secondly, when you start to move onto more complicated analysis, you can utilise the features of a list to create datasets that wouldn't be possible in something like Excel. For instance, we know that we can store models in a list. Well, let's say we had a dataset that had data for lots of different countries and we wanted to create a separate model for each country. We could have a dataset that had the country in one column and then the model in another:

```
data.frame(
  country = c("England", "Spain", "France"),
  model =  I(list(model(...), model(...), model(...)))
  # The I() just tells R to leave it as a list
)
```

For now though, don't worry too much about the internals. Just remember that data frames are the most flexible dataset storage medium and they'll be what you do most of your analysis with. And if you can remember that each column is technically a list, then you're ahead of the game.

3.5 Subsetting

There will be occasions where you don't want all the values in a vector/list/matrix/dataframe. Instead, you'll only want a *subset*. The way to do that is slightly different depending on the data structure you're using.

Note: In some programming languages, an index starts from 0. This means

that you have a list or array or similar, the first value is at position 0, then 1, then 2, etc. In R, the first value is at position 1. In other words, we index from 1 in R.

3.5.1 Vectors

Vectors are simple. Just use square brackets (`[]` or `[[]]`) after your vector and provide the index or indices of the values that you want:

```
c(10,20,30,40)[1]
```

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

```
c(10,20,30,40)[c(1,4)]
```

```
## [1] 10 40
```

```
c(10,20,30,40)[1:3]
```

```
## [1] 10 20 30
```

```
c(10,20,30,40)[[1]]
```

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

P.S. If you have a vector of named values, you can also use the names instead of the indices. Like `c(value_1 = 1)[["value_1"]]`.

But Adam, I hear you ask, `c(10,20,30,40)[1]` and `c(10,20,30,40)[[1]]` just gave us the same thing, so are the interchangeable?

Well, they kind of returned the same thing, but they didn't. So no, they're not interchangeable.

Essentially, `[]` returns the *container* at the provided index, where `[[]]` returns the *value* at the provided index. Let's see a practical example of how these are different:

```
c(value_1 = 10,
  value_2 = 20)[1]
```

```
## value_1
##      10
```

```
c(value_1 = 10,
  value_2 = 20)[[1]]
```

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

In the first call, we get the name of the value and the value itself. In other words, rather than just returning the value at that index, we've essentially just chopped up the vector to only returning everything from the first position. Conversely,

in the second call, we've just been given the value. What we've done here is extracted the value out from that position.

As a result of this difference, **`[]` can be used with more that one index (e.g. `[1:5]` or `[c(1,3)]`) whereas `[[]]` can only be used with a single index**.

It's a very subtle difference, but it is an important one. Make sure that if you want the value, use `[[]]`, and if you want the whole part of the vector, use `[]`.

3.5.2 Lists

Lists can be subsetted in the same way as vectors - `[]` returns the container at the index provided and `[[]]` returns the value:

```
list(
  value_1 = c(1,2,3),
  value_2 = c("hello", "there", "everyone")
)[[1]]
```

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

```
list(
  value_1 = c(1,2,3),
  value_2 = c("hello", "there", "everyone")
)[1]
```

```
## $value_1
## [1] 1 2 3
```

A key difference with lists however, is that you can also subset based on the name of the value in the list using the `$` operator:

```
list(
  value_1 = c(1,2,3),
  value_2 = c("hello", "there", "everyone")
)$value_1
```

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

This is equivalent to:

```
list(
  value_1 = c(1,2,3),
  value_2 = c("hello", "there", "everyone")
)[["value_1"]]
```

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

Another key difference is that lists can, of course, hold recursive values. This means that subsetting a list can return another list, that can also be subsetted and so on:

```
list(
  list_1 = list(
    list_2 = list(
      list_3 = "hello"
    )
  )
)[1][1][1]
```

```
## $list_1
## $list_1$list_2
## $list_1$list_2$list_3
## [1] "hello"
```

And of course, you can do the same thing with the `[[]]` operator if you only want the value and not the container.

3.5.3 Matrices

Matrices are two dimension, meaning they can't be subsetted with a single value. Instead, we still use the `[]` operator, but we provide two values: one for the row and another for the column:

```
matrix(c(1:10), nrow = 5, ncol = 2)
```

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

```
matrix(c(1:10), nrow = 5, ncol = 2)[4,1]
```

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

3.5.4 Dataframes

Dataframes can be subsetted in the same way as matrices (using the `[]` operator). However, dataframes can also be subsetted (like lists), using the `$` operator and the name of the column:

```
data.frame(
  col_1 = c(1,2,3),
```

```
  col_2 = c("hello", "there", "everybody")
)$col_1
```

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

Why does this approach work for dataframes? Well, as I alluded to before, dataframes store columns as lists. But technically, the dataframe itself is also stored as a kind of list, with each column being another entry in that list. So, just like we can subset lists using `$`, we can subset dataframes with it as well because a dataframe is like a fancy list.

3.5.5 Subsetting by criteria

Sometimes, you might not know the indices of the items you want to extract from a datastructure. Instead, you might want to do something like "extract all numbers from a vector that are less than three". To do this, we essentially find the indices of the values that match our criteria and then subset the data structure like we learned previously.

Let's look at subsetting a vector as an example:

```
vector1 <- c(10,15,14,20,21,50)
```

Let's say want to extract all of the values below 20. To find the indices of the values that match our criteria, we just use our logical operators:

```
vector1 < 20
```

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

This returns TRUE if the value is less than 20, and FALSE if it isn't. We can then pass this vector of TRUE and FALSEs in `[]` after the vector to only return the values we want:

```
vector1[vector1 < 20]
```

```
## [1] 10 15 14
```

Other data structures can also be subsetted in the same way, but for matrices or dataframes, it's easier to use something like `subset` or `dplyr::filter()` (although `subset` has its own limitations).

You'll notice that this is ever so slightly different to the way we were subsetting before. Previously, we were providing just the indices of the values we wanted (e.g. 1,2 and 4). But here, we're actually providing a vector of TRUE and FALSE values to indicate which values we want. The structure is slightly different, but the logic is the same.

This does mean however, that you can also provide a vector of TRUEs and FALSEs yourself manually if you wish. There are two reasons why I would avoid

this however:

1. It takes longer to write out
2. If you don't provide the same number of logical values (i.e. TRUEs and FALSEs) as there are values in the vector, then the logical values are **recycled**. That means that if you have a vector that's 6 values long, and you provide a logical vector to subset it that is only three values long, then your logical vector is going to be repeated. This can lead to unwanted results:

```
vector1[c(TRUE, FALSE)]
```

```
## [1] 10 14 21
```

Here, because I've only specified two logical values, when it comes to subsetting time, those two values will be recycled to create a vector like this `c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)`. This is why we get three values returned instead of the expected one.

So while you can manually subset with a vector of logical values indicating whether to return that value as is returned, it's best to stay away from it.

3.6 Functions

Being a functional programming language, functions are at the heart of R. We've already used lots of functions in the previous chapters, but now we're going to look in more detail about what a function is.

3.6.1 Function basics

John Chambers, creator of the S programming language upon which R is based and core member of the R programming language project, said this:

> "To understand computations in R, two slogans are helpful:
>
> - Everything that exists is an object.
> - Everything that happens is a function call.
>
> — John Chambers"

For now, we're going to focus on that second statement. What does it mean?

Well, a function is quite simple. It has an input, it does something, and then it gives an output. A really simple example of this is just typing `print(1)` into the console and hitting enter. You've given an input, there was a calculation, and now there's an output. Something's happened (1 was printed in the console) and it was done by calling a function (`print`).

If you're well-versed in mathematics, you'll know that functions in maths are the same. $f(x) = 3x$ means that the to get `y`, you take `x` and multiply it by three. In this case, our input is `x`, our bit in the middle is multiplying by three, and then our output is `y`.

If you haven't used functions in mathematics then don't worry. Even by getting this far in the book, you've already used functions loads of times. For example, how do you create a vector? If you remember, you use the `c()` function, which we know stands for "concatenate". So, every time you've created a vector, you've used a function without even knowing it. The input was whatever you provided in the brackets. The computation was to concatenate everything together. And then the output was the vector.

Similarly, whenever you created a factor or a matrix or a dataframe or whatever, you used a function. You provided an input, there was a computation to change that input, and then you got an output.

As confusing as functions will inevitably become, just try to remember the core of what a function is: When you call a function, there's an input, something happens, and there's an output.

3.6.1.1 Functions in R

So more specifically, what do functions looks like in R? Well, a good starting point is that when we call (use) a function, its almost always followed by brackets (`()`) when you use them. This helps make it clear what values you're providing as your inputs. For example, the `c()` function, the `data.frame()` function, the `sum()` function are all followed by `()`, which is how you provide your inputs.

I say that nearly almost all functions are followed by `()`, because some aren't. A simple example of this is `+`. `+` is still a function:

```
is.function(`+`)
```

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

```
# the backticks just mean I'm referring
# to the + function without using it
```

But it doesn't have brackets. Instead, we can use a shorthand where we provide the values we want to give to the function either side of it (e.g. `1 + 2`). Importantly however, the logic is exactly the same, and you can still use the `+` like a normal function with brackets:

```
`+`(1,2)
```

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

It's just that this looks a little weird to us, so we often use the shorthand way. But the long and short of it is: an easy way to tell when someone is calling

(using) a function is to look for the `()` after the function name.

3.6.1.2 Inputs

We know that to use a function in R, we have to provide inputs*. And we also know that we provide our inputs within the brackets after the function name. But how do we know what values are allowed?

*Technically, sometimes you don't have to provide an input to a function (e.g. `Sys.Date()`, which gives us the current date without putting anything in the brackets). But in the interests of clarity, just imagine that the inputs to these functions are blank rather than that they don't have any input at all.

By typing a `?` followed by the name of the function into the console (e.g. `?length()`), you'll get a help page showing you the input parameters allowed by the function. So if we use `?length()` as an example, the help page tells us that the `length()` function expects one input parameter, `x`, and that needs to be an `R` object. Nice and simple.

In some cases, you'll see a `...` as one of the input parameters. This essentially means that you can provide an indeterminate number of values for that input. I know that sounds confusing, but the `c()` function is a good way of demonstrating this. When you create a vector, you can provide an (essentially) infinite number of values to the function. So the `c()` function basically bundles everything you provide to it into that `...` parameter.

3.6.1.2.1 Explicit input parameters

If you type `?c()` into the console however, you'll see that there are also some other input parameters: `recursive` and `use.names`. Well Adam, if `...` just bundles everything I provide into a single input, then how do those work? Well this outlines the importance of providing **explicit** input parameters. When we're explicit, we're saying exactly which input parameter we're referring to with each value we provide. And to do this, we just provide the name of the input parameter when we give it. Let's look at the `substr()` function as an example.

The `substr()` function simply returns part of a character string that you provide. So, if I was to type:

```
substr("Hello", 1, 3)
```

```
## [1] "Hel"
```

I get the first to the third characters in the string "hello". With this function call however, I haven't been explicit. Instead, I've just provided the inputs in the order that they're listed in the documentation:

- x

  - a character vector

- start
  - the first element to be extracted

- stop
  - the last element to be extracted

To be explicit, I need to provide the name of the input parameter that I'm referring to when I provide my inputs:

```
substr(x = "Hello", start = 1, stop = 3)
```

```
## [1] "Hel"
```

```
substr(start = 1, stop = 3, x = "Hello")
```

```
## [1] "Hel"
```

Notice how, when I'm being explicit, it doesn't matter what order I provide my inputs in, R knows which value should be mapped to which input parameter.

Also, notice how we're using `=` here and not anything else like `<-`? This is another reason why I suggest not using `=` for assignment: we use `=` when we're providing input parameters and so it's good to keep them separate.

So how does this link back with the `...`? Well, with the `c()` function, every unnamed parameter you provide is bundled into the `...` parameter. To give values for the `recursive` and `use.names` parameters, you'd need to provide them *explicitly* (e.g. `recursive = TRUE`). This will be true of many functions where you see a `...`. If you're not explicit with the parameters that you don't want to be included in the `...`, you're going to have a bad time.

3.6.1.2.2 Optional input parameters

For many functions, certain parameters have a predefined value that they will default to. This provides a level of flexibility whilst not requiring lines and lines of code for every function call; there's a default value, but you can override it if needed.

Optional parameters are easily distinguished in the documentation of a function because they will a value already assigned to them like this: `use.names = TRUE`.

For instance, when we create a vector using the `c()` function, there are two optional parameters (`recursive` and `use.names`) that already have the values `TRUE` and `FALSE` assigned to them. To override these defaults, we just need to provide a new value to the parameter like this:

```
c(1,10,15, use.names = FALSE)
```

```
## [1]  1 10 15
```

3.6.1.3 Outputs

First and foremost, in R you can have as many input as you like to a function. However, a function will only ever return one *thing*. I say one *thing*, because functions can return a list which itself can contain multiple values, but just keep this in mind: **Functions in R have a single return value**.

3.6.1.3.1 Reassigning outputs

Functions in R do not edit the inputs you provide in place. Instead, they essentially work on copies of the inputs you provide. Here's a quick example:

```
x <- 1
sum(x, 1)
```

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

```
x
```

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

As you can see, when we call the `sum()` function with `x` as an input parameter, the value of `x` stays the same.

If you do want to edit your original value, you just need to reassign the output of the function call back to the variable. I know that sounds complicated, but it's quite simple:

```
x <- 1
x <- sum(x,1)
x
```

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

This works because the right-hand side of the assignment line is executed first. In other words, when the `sum(x,1)` is evaluated, `x` is still equal to one. This makes sense because otherwise it'd be very hard to keep track of what `x` was equal to!

This behaviour (not changing the input parameter value in place) is a major point of difference between functions and what are called methods in other languages. If you're coming from something like Python, you may be used to changing objects through methods: `object.AddNew` or something like that. In R, functions do not change variables in the global environment because they are executed in their own environment. To learn more about environments, there is an environments chapter in the "For Teachers" section.

Chapter 4

For Teachers

Now we have a basic understanding of some of the core concepts in R, let's have a look at them in a bit more detail. In contrast to the previous section, here were going to focus on the underpinnings of R. You can kind of think of this section as being less practical and more theoretical.

If you're going to teach R, this section will provide you with a good level of understanding. Hopefully, you'll not only know *how* to do something when asked, but *why* you should do it that way.

4.1 Functions - Advanced

In the "For Students" section, we looked at what a function is and how to use one. In this section, we're going to look more at the structure of a function and how you might go about writing your own functions.

When you start writing our own functions, you'll start to see a massive improvement in your efficiency. By extracting out common tasks to functions and by keeping your functions simple, you can easily expand and debug your project. A rough rule of thumb is that if you've copied some code more than twice, think about extracting it out to a function.

Let's take a look at an example of where it may be appropriate to shorten your workflow by using a function. Let's say you've got 2 datasets, and you want to get the standard deviation and mean of one column and then create a normal distribution based on those values:

```
dataset1 <- data.frame(
  observation_number = c(1,2,3,4,5),
  value = c(10,35,13,20,40)
)
```

```
dataset2 <- data.frame(
  observation_number = c(1,2,3,4,5),
  income = c(100,200,150,600,900)
)
```

Without using functions, our workflow might look like this:

```
mean_ds1 <- mean(dataset1$value)
sd_ds1 <- sd(dataset1$value)
normal_dist_ds1 <- rnorm(1000, mean = mean_ds1, sd = sd_ds1)

mean_ds2 <- mean(dataset2$income)
sd_ds2 <- sd(dataset2$income)
normal_dist_ds2 <- rnorm(1000, mean = mean_ds2, sd = sd_ds2)
```

This isn't too bad, but what if we wanted to add another dataset? We'd have to copy and paste the code yet again. Then, say we wanted not to use the mean but the median, we'd have to replace each call to the `mean()` function with `median()` in each block of code.

If we extract out the commonalities to a function, then not only do we reduce the amount of code we're using, but this also makes future changes or fixes much easier.

We'll look more specifically out how we create functions in the next few sections, but for now let's imagine what we'd want our function to look like. It'd need to calculate the mean and standard deviation of a column, but that column name or position in the dataframe might change - the column is called 'value' in the first dataset but is called 'income' in the second, and even though they're both the second column in the dataset, we don't want to rely on that in case we have a new dataset where the column we want to use isn't in that position.

Let's revisit this once we understand how we create functions in R a bit better. If you're comfortable with creating functions in R, then you can skip to the solution.

4.1.1 Creating functions

R and its packages give you access to hundreds of thousands of different functions, all tailored to perform a particular task. Despite this wide array to choose from however, they will always be cases where there isn't a function to do exactly what you need to do. For those of you coming over from Excel, this can often be a serious source of frustration where there isn't an Excel function for you to use and there isn't an easy way to create one without knowing VBA.

R is different. Creating functions can be very simple and will really change the way you work.

Creating functions will also highlight an important delineation. Previously, we've been focusing on *calling* functions. Calling a function is essentially using it. But in order to call a function, it needs to be *defined*. Base functions are already defined (i.e. someone has already written what the function is going to do), but when you're creating your own functions, you are *defining* a new function that you're presumably going to call later on.

4.1.1.1 Function structure

If we go back to the beginning of this chapter, we learned that everything that exists is an object. Functions are no exception, and so we create them like we do all our other objects. There is a slight diversion however. When we define a function, we assign it to an object with the `function` keyword like this:

```
my_first_function <- function() {}
```

Notice how we've got two sets of brackets here. The first (`()`) is where we define our input parameters. The second (`{}`) is where we define the body of our function.

Let's do a simple example. Let's create a function that adds two numbers together:

```
my_sum_function <- function(x, y) {
  x + y
}
```

So in this example, I've defined that when anyone uses the function, they need to provide two input parameters named `x` and `y`. Something that people tend to struggle with is that the names of your input parameters have no implicit meaning. They are just used to reference the value provided in the body of the function and, hopefully, make it clear what kind of thing the user of the function should be providing. This is why for example in some functions that require a dataframe there will be an input parameter called `df` or similar. But importantly, these names are technically just arbitrary.

In the body of the function, we can see that we're just doing something really simple: we're adding `x` and `y` together with `+`.

Once I've run the code to **define** my function, I can then **call** it like I would any other function:

```
my_sum_function(x = 5, y = 6)
```

```
## [1] 11
```

4.1.1.1.1 Optional input parameters

When defining your function, you can define optional parameters. These will likely be values where most of the time you need it to be one thing, but there are edge cases where you need it to be something else. Defining optional parameters is really easy; whenever you define your function, just give it a value and that will be its default:

```
add_mostly_2 <- function(x, y = 2){
  x + y
}

add_mostly_2(x = 5)
```

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

```
add_mostly_2(x = 5, y = 3)
```

```
## [1] 8
```

Sometimes you'll want to provide users with a set of options. To do so, provide a default value of a vector and use the `match.arg()` function like this:

```
greet_me <- function(greeting = c("hello", "welcome")) {
  greeting <- match.arg(greeting)
  greeting
}
```

This will ensure that users of the function can only provide one of the values in the vector. If they use the default, then the first value in the vector will be used:

```
greet_me("welcome")
```

```
## [1] "welcome"
```

```
greet_me()
```

```
## [1] "hello"
```

```
greet_me("wassup") # This will error because 1 wasn't an options for y
```

```
## Error in match.arg(greeting): 'arg' should be one of "hello", "welcome"
```

Note: The `match.arg()` function only works with character vectors.

4.1.1.2 ...

You'll notice a crucial distinction between R's `sum()` function and ours. The base function allows for an indeterminate number of input parameters, whereas we've only allowed 2 (`x` and `y`). This is because the base `sum()` function uses a `...`. This `...` is essentially shorthand for "as many or as few inputs as the user wants to provide". To use the `...`, just add it as in an input parameter:

```
dot_dot_dot_function <- function(x, y, ...) {
}
```

The `...` works particularly well when you might be creating a function that *wraps* around another one. A wrapping function is just a function that makes a call to another one within it, like this:

```
sum_and_add_2 <- function(...){
  sum(...) + 2
}
```

All we're basically doing in the above wrapping around the `sum()` function to add some specific functionality.

By using the `...` here, we can just pass everything that the user provides to the `sum()` function. This means we don't have to worry about copying any input parameters.

4.1.1.2.1 Return values

As I mentioned in the "For Students" section, functions have a single return value. By default, a function will return the last evaluated object in the function environment. In our `my_sum_function` example, our last evaluation was `x + y`, so the output of that was what was returned by the function.

You can also be explicit with your return values by using the `return()` function. The `return()` function will return whatever is provided to the `return()` function. This can be useful if you want to return a value prematurely:

```
early_return_function <- function(x,y, return_x  = TRUE) {
  if (return_x) {
    return(x)
  }
  x + y
}
```

Here, we can see more clearly that `x` is returned when `return_x` is TRUE and `x + y` is returned otherwise.

Certain style guides suggest that you should **only** use `return()` statements for early returns. In other words, the "normal" return value for your function should be defined by what's evaluated last. Personally, I think you should use whatever makes it clearer for you. I quite like seeing explicit `return()` values in a function because I find it makes it clearer what all the possible return values are, but this is just personal preference.

4.1.2 Input validation

Unlike some other languages, functions do not have a specific data type tied to each input parameter. Any requirements that should be imposed on an input parameter (e.g. it should be numeric) are done by the function creator in the body of the function. So for instance, when you try to sum character strings, the error you get occurs because of type-checking in the body of the function, not when you provide the input parameters.

```
function_without_check <- function(x, y) {
  x + y
}
function_without_check(x = 2, y = "error for me please")
```

```
## Error in x + y: non-numeric argument to binary operator
```

```
function_with_check <- function(x, y) {
  if (!is.numeric(x) || !is.numeric(y)) {
    warning("x or y isn't numeric. Returning NA")
    return(NA_integer_)
  } else {
    x + y
  }
}
```

4.1.3 Functions as objects

Functions are technically just another object. This means that you can use functions like you would any other object. For instance, some functions will accept other functions as an input parameter. When we move onto the `apply` logic, the `lapply()` (list-apply) function requires a `FUN` parameter that is the function the be applied to each value in the provided list.

```
sum_list <- list(
  c(1,2),
  c(5,10),
  c(20,30)
)

lapply(sum_list, FUN = sum)
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 15
```

```
##
## [[3]]
## [1] 50
```

Linked with the idea that functions are just another type of an object, there is an important distinction between `sum` and `sum()`. The first will return the `sum` *object*. That is, not the result of applying inputs to the `sum` function, but the function itself. If you just type the name of the function into the console, it will show you the code for that function (it's definition):

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

Conversely, `sum()` will *call* the `sum` function with the inputs provided in the brackets.

```
sum(1,2)
```

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

4.1.4 Example answer

Going back to our previous example of when we might want to make a function, what might our function actually look like. Recapping, we know we need to calculate the mean and sd of a column, but that the name of the column might change. Using what we've just learnt, let's have a go:

```
create_norm_dist_from_column <- function(dataset, column_name, n = 1000) {
  ds_mean <- mean(dataset[[column_name]])
  ds_sd <- sd(dataset[[column_name]])
  rnorm(n = n, mean = ds_mean, sd = ds_sd)
}

normal_dist_ds1 <- create_norm_dist_from_column(dataset1, "value")
normal_dist_ds2 <- create_norm_dist_from_column(dataset2, "income")
```

Now, instead of copying the code each time we need it, we've extracted the common computations to a function and then we call the function where we need. Hopefully this demonstrates the logic behind why functions can be so useful.

This is an example of the concept of abstraction, which is a common theme in programming. If you're interested in learning more about abstraction, the opeRate book that I wrote to turn the understanding you've hopefully built up over this book into actual data analysis skills looks at abstraction in more detail.

4.2 Environments

As your scripts become more complex, the number of variables and functions that you assign will start to increase. Pair this with the fact that you may be using lots of external packages that will all contain lots of functions and the number of objects you're working with can easily get into the hundreds and even thousands.

This presents a scoping issue: If I refer to the object `x`, what `x` do I mean if there's more than one? In other words, in which scope should R search for the `x` object?

R uses environments to solve this issue. Environments are collections of objects that can be used to group similar objects and provides a replicable naming convention for retrieving objects that may have the same name from the appropriate environment.

In this chapter, we're going to understand the concepts underpinning environments and scope in R.

4.2.1 Environment basics

At its core, an environment is a collection of objects. A bit like a list, environments store multiple objects in a single structure.

To create a new environment, we use the `new.env()` function.

```
new_env <- new.env()
```

To add items to our environment, we can add them like we would a list using the `$` operator:

```
new_env$first_object <- "hello"
```

To list all of the objects in an environment, we use the `ls()` function:

```
ls(new_env)
```

```
## [1] "first_object"
```

Importantly, you can't have two objects in the same environment with the same name. If you try, you'll just overwrite the previous value:

```
new_env$first_object <- "world"
new_env$first_object
```

```
## [1] "world"
```

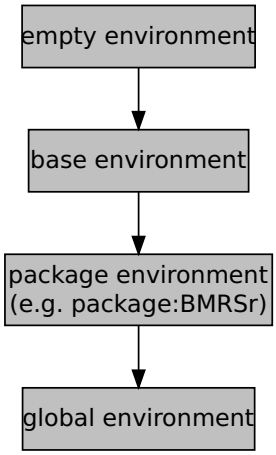
4.2.2 Environment inheritance

Environments have parents and children. In other words, there is a hierarchy of environments, with environments being encapsulated in other environments while also encapsulated other environments.

Every environment (with the exception of what we call the empty environment) has a parent. For example, when I created my `new_env` environment before, this was created in the **global environment**. The global environment is the environment that objects are assigned to when working in R interactively. The global environment's parent environment will be the environment of the last package you loaded. Packages have environments to avoid name conflicts with functions and to help R know where to look for a function. These package environments will contain everything that the package developer included with the package (i.e. functions, maybe some datasets, etc.).

At the top of the environments of packages you've loaded will be the **base environment** which is the environment of base R. Finally, the base environment's parent is the **empty environment** which does not have a parent.

The hierarchy of these environments looks like this:



4.2.3 Scope

So we know that objects in the same environment can't have the same name, but what happens when two different environments happen to have objects with the same name? This is where the concept of **scope** comes in. **Scoping** is the set of rules that governs where R will look for a value.

Well, R will search for the object in order of environment, starting at the most specific environment (so the global environment in the above diagram) and moving up. For example, we know that there is a function in base R called `sum()`. But, if I define a new function in the global environment called `sum` then which function will be called when I type `sum(...)`. Well, because we know that the search path starts from the most specific environment, R will look for `sum` in the global environment first and it'll find the `sum` that I've just defined. At this point, it'll stop looking because `sum` has been found.

For this reason, it's a good idea to use a package like `conflicted` to manage the packages you loaded, otherwise which function you use when you have two functions with the same name from different packages will be defined by which one you loaded later.

Alternatively, you can specific with which environment R should look for a particular function by prefixing the function with its package and `::`. For example, if I decided against my better judgement to define a function called `sum` in my environment, and then I wanted to call the base function, I could do so like this:

```
base::sum(1,2)
```

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

4.2.4 Function environments

Functions, when they called, create their own more specific environment. The parent of this environment will be the environment in which it was called (most often this will be the global environment).

This breeds some specific behaviours. For example, say you've written a function that expects two input parameters, `x` and `y`. Well, what would happen if someone had already defined an `x` and `y` variable in their script? Which value should R use?

Let's see what happens.

```
sum_custom <- function(x,y) {
  x + y
}

x <- 10
y <- 5

sum_custom(x = 1, y = 2)
```

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

In this case, the fact that there is already an `x` and `y` in the global environment doesn't really make much difference. The function creates its own more specific environment when it's called, and it looks for the `x` and `y` variables in here first. It finds them and uses those values (1 and 2).

But what happens if a variable doesn't exist in the more specific function environment? Let's take a look.

```
sum_custom <- function(x,y) {
  x + y + w
}

w <- 5

sum_custom(x = 1, y = 2)
```

```
## [1] 8
```

In this case, the function looks in the specific environment for `w`, but it doesn't exist. The only objects that exist in the function environment are the `x` and `y` that we've provided. So when R doesn't find it in the more specific environment, it looks in the less-specific global environment. It finds, and so it uses the value it finds.

This can be a dangerous thing, so always make sure that you're function is accessing the values you think it is.

So does R work the other way? Does it ever look in a more specific environment? Nope.

```
sum_custom <- function(x,y){
  im_a_sneaky_variable <- 10
  x + y
}

im_a_sneaky_variable
```

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

Once the function is called, objects in its environment are inaccessible. The long and short of it is, R will start from specific environments and then look upwards, never downwards.

4.2.4.1 'Super' assignment

There will be occasions however when you need to make changes to the global environment. For instance, say you want to increment a counter every time a function is called, regardless of where it's called from. In these cases, you can

use the controversial `<<-` operator. This is used as an assignment operator to assign a value to the global environment. Observe...

```
sum_custom <- function(x,y) {
  count <<- count + 1
  x + y
}

count <- 0

sum_custom(1,2)
```

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

```
count
```

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

```
sum_custom(2,3)
```

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

```
count
```

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

Note how when we assign `0` to our `count` variable outside of the function, we don't need to use `<<-`. This is because we're already assigning to the global environment.

Use the `<<-` with care and only assign something to the global environment if you really need to. Otherwise, you may start overwriting variables in your global environment without ever realising it.

4.3 Objects and Classes

Objects and classes are a fairly ubiquitous concept across programming languages and data analysis tools. We'll briefly look at what an object is in R, but by no means is this an exhaustive description.

4.3.1 What is a class?

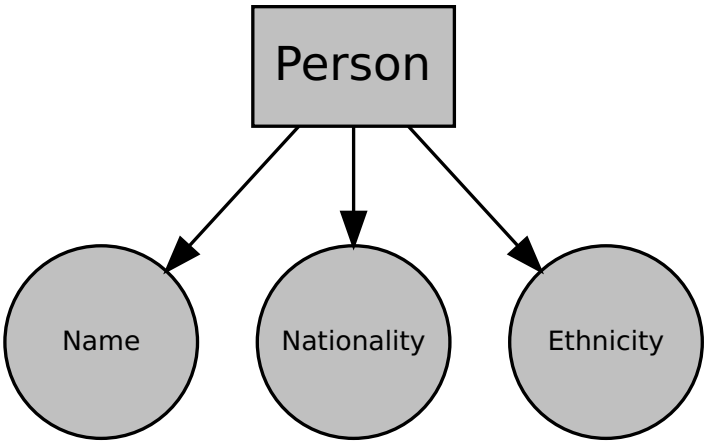
The world is complicated. Everything in the world is unique and defined by an infinite number of properties and features. For example, if you take a person, they can be defined by their name, where they're from, where their parents are from, their hair colour, their likes and dislikes, and so on and so on. When we want to store data in a structured and formal way in a computer system, however, this isn't particularly helpful. Instead, we need to store data in a predefined

structure. This is essentially what an class is; a pre-defined structure to store important attributes or features of a **thing**.

Let's take the person instance again. Let's say we're going to store data on a number of individuals, we won't be able to store everything about them. So we'll choose a subset of their attributes or features to store that are relevant to what we need. But to make things more efficient, we're going to store the same information for each one. So lets say we're going to do some geographical analysis, we might want to include a person's name, their nationality, and perhaps their ethnicity. So for each person we want to store, we can store these three attributes. And we might call this data structure a "person". Well this is exactly what an class is; a collection of attributes and features that is shared objects instances of the same type.

So our class is "person", and it has the attributes "name", "nationality", and "ethnicity". Now this obviously doesn't capture everything about a person, but it's enough for what we want to do.

Graphing that object might look something like this:



4.3.2 What is an object?

An object is just an instance of a class. So if you create a person from our "person" called John, then "John" is the object an "person" is the class.

When you create a dataframe for example, you're creating a `data.frame` object from the basic structure of the `data.frame` class.

4.3.3 Objects and Classes in R

Looking more specifically at R, what kind of objects do we see. Well, according to John Chambers, the founder of the S programming language upon which R is based, everything is:

> "To understand computations in R, two slogans are helpful:
>
> - Everything that exists is an object.
> - Everything that happens is a function call.
>
> — John Chambers"

So every function, dataframe, plot, list, vector, integer, everything is an object.

To see the class of an object in R, use the `class()` function.

```
class(data.frame()) # here we're finding the class of an empty data.frame
```

```
## [1] "data.frame"
```

```
class(data.frame) # here we're finding the class of the data.frame() function
```

```
## [1] "function"
```

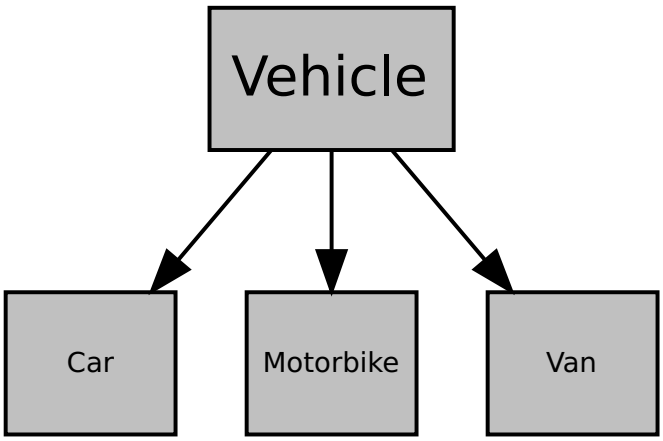
```
class(1)
```

```
## [1] "numeric"
```

4.3.3.1 Inheritance

Sometimes, different classes are interrelated. For example, if you were storing data on vehicle, then you might have a "vehicle" class. But if you're storing data on lots of different vehicles, you might also have classes for each type of vehicle (e.g. "car", "motorbike", etc.). All of these classes are still vehicles, and so you don't want to have to repeat yourself when you define each of those classes. In other words, all of those classes are going to have some common attributes like colour, horsepower and so on. Similarly, each different type of vehicle will have some attributes that are unique to that type of vehicle. For instance, motorbikes can have sidecars but vans and cars don't. Cars and vans have doors but motorbikes don't. This highlights the benefits of inheritance. By creating a "vehicle" class and allowing your subsequent classes to inherit all of the attributes of the "vehicle" class, you can avoid duplication while allowing distinct classes. This way, when you want to add any attributes to all of your vehicles, you can just do it via the "vehicle" class rather than changing each type of vehicle class.

Diagramming this relationship:



Inheritance is an extremely deep topic which we won't go into here, but R objects also use inheritance. To see the inheritance tree of an object, use the `is()` function:

```
is(1L)
```

```
## [1] "integer"             "double"              "numeric"
## [4] "vector"              "data.frameRowLabels"
```

Here we can see that an integer object is made up of the integer, double, numeric classes. The order of inheritance goes from left to right so we know that the integer class inherits from the double class, which inherits from the numeric class.

4.3.3.2 Object-Oriented Systems

Unfortunately (or fortunately, depending on your point of view), R doesn't have a single way of storing objects. In fact, there are 2 object-oriented (OO) programming systems in base R, and more (like R6) can be added via packages. These two base OO systems are S3 and S4, and they differ in the way that objects are constructed, stored, and interacted with. We're not going to go into the difference here, but I recommend Hadley's Advanced R to better understand the difference between the two. For now, I'm just going to explain the basics of the S3 system as I think it's the easiest to understand and helps convey the philosophy behind why we use objects slightly more easily.

4.3.3.2.1 S3

In the S3 system, we rely on generic functions and methods. Generics are functions that have a single common goal, but that can be used for objects that might be very different. For example, `print()`-ing a dataframe is going to be different to `print()`-ing a plot or an API response or whatever. But `print()` always seems to know what to do. The reason it does is that the `print()` function is a **generic** function that actually uses a more specific function to achieve it's goal. In other words, we achieve a fairly high level goal like printing by calling a function that is specific to the object we're working on under the hood. These more specific functions are called **methods**.

As a real world analogy, think of the process of talking to someone. The common goal in talking is to communicate. But, depending on the language that someone speaks, the actual act of talking is going to be slightly different for different people you talk to. In this case, you can think of **communicating** as being the **generic** - it's the eventual goal. And **talking in the appropriate language** as being the **method**.

Going back to R, if you type `print.` into the console, the autocompleter will show you lots and lots of `print.something()` functions. These are all of the **methods** for all of the different printable objects in R. `print.date()` will print a date object, `print.data.frame()` will print a dataframe object and so on. But when you just use the `print()` function on an object, R will automatically choose which method it needs for the object you've passed as an input parameter.

```
# here we're using the generic
print(as.Date("2020/06/10"))
```

```
## [1] "2020-06-10"
```

```
# because we're printing a Date object,
# this is the method that is actually used
print.Date(as.Date("2020/06/10"))
```

```
## [1] "2020-06-10"
```

If an object inherits multiple classes, then R will look for the correct method starting from it's most specific class. So for example, if an object has classes `c("class_1", "class_2")`, then R would look for `print.class_1()` and only look for `print.class_2()` (and then `print.default()`) if it couldn't find `print.class_1()`.

While you can often tell if something is an S3 method by it being a generic followed by a `.` and then an object name, don't rely on this, because people often use `.` to separate words in functions that can make them look like S3 methods when they're not.

The important thing I hoped you've gleaned from this explanation is that you don't really need to know what your object is to use a generic. Yes, there are generics that only have methods for specific objects and so it may be important to know what type your object is to make sure you can use a particular generic,

R takes care of which method is dispatched for you. If that doesn't convince you that you don't need to understand the underlying structure of an object to get hands on with R, I don't know what will.

Note: In some languages, methods are essentially functions or sub-routines that are tied to an class. For instance, a class that represents a person's bank account might have the method `Balance()`, which will return how much money a person has in their account. There are certainly some similarities between how methods are used in R and some other languages, they are a bit different. Mainly, in R, methods are not attributes of a class but are separate functions.

4.3.4 Creating classes

The obvious question is "Can I make my own class in R?". The answer is yes. And quite easily too. Whether you should or not is a different story. We're just going to look very briefly at creating a class in using S3 for now, but I'd recommend Hadley's Advanced R for a deeper look at the different systems and how to use them.

4.3.4.1 Class attribute

In S3, the class of an object is defined by its `class` attribute, which stores a vector of class names. To see an object's class, we use the `class()` function:

```
class(1)
```

```
## [1] "numeric"
```

```
class(TRUE)
```

```
## [1] "logical"
```

This class attribute will determine what method is dispatched when we call a generic on this object. So here, if we called `print()` on `1`, then it would look for `print.numeric()` and dispatch that method to print the object. Technically, this just calls `print.default()` because the default method knows how to print numeric values but the logic remains.

Something which will make OO programmers everywhere cringe, is that the class attribute for an R object is modifiable:

```
var_1 <- 1
class(var_1) <- "custom_class"
class(var_1)
```

```
## [1] "custom_class"
```

Now, when we attempt to print the `var_1` variable, R will look for the appropriate method based on it's class (`custom_class`), which would be

`print.custom_class()`. In this case, a `print.custom_class` method doesn't exist, so it'll default to the `print.default()` method.

Importantly, we've learnt here that setting the class of an object is as easy as using the `class()` function and the `<-` operator.

4.3.4.2 Constructor functions

To ensure our objects in the same class all have a similar structure, we want to make a constructor function that requires certain values. For example, let's create a class to store addresses, and we'll say that we need the house name, the road and the city. Our constructor function might look like this:

```
address <- function(house_name, road, city) {
  address <- list(
    house_name = house_name,
    road = road,
    city = city
    )
  class(address) <- "address"
  return(address)
}
```

Now to create an object from my constructor function, I just need to use the `address()` function:

```
my_address <- address("4", "Pleasant Drive", "London")
class(my_address)
```

```
## [1] "address"
```

Now, if I called `print(my_address)`, R would look for a `print.address()` method. But, it wouldn't find one and so would turn to `print.default()` which would print our object like any other list. Therefore, we might want to write our own method to print our new object how we want to.

4.3.4.3 Creating methods

To create a method, all we need to do is create a function with the name of our generic followed by a '.' and then name of our class. So from our previous example, our method would look like this:

```
print.address <- function(address) {
  cat("The house name is: ", address$house_name, "\n")
  cat("The road is: ", address$road, "\n")
  cat("The city is: ", address$city, "\n")
}
```

Now when we call `print()` on our new object, it will find the `print.address()` method and use that to print our object:

```
print(my_address)
```

```
## The house name is:  4
## The road is:  Pleasant Drive
## The city is:  London
```

4.3.4.4 Creating generics

You can also create your own generic functions for your classes. Generic functions have a very simple structure. Let's say we want to create a new generic called `post`:

```
post <- function(x) {
  UseMethod("post")
}
```

This is just telling R that when the `post()` function is called on an object, look for the appropriate `post` method to use. Without us writing any methods, this would be useless because R wouldn't have any methods to use!

Let's create a default method that can be applied if we don't have a more specific class:

```
post.default <- function(x){
  cat("Your object is now in the post! Expect it in 5 to 10 weeks")
}
```

In our example, this default method isn't going to be very useful but it demonstrates that we need a default method to be dispatched if we haven't written a specific method for that object class.

Let's write a method for our `address` object:

```
post.address <- function(x) {
  cat("Your object has been posted to ", x$house_name, " ", x$road, " ", x$city, "!")
}
```

Now, if we call our new generic on two different classes of object and compare:

```
post(1)
```

```
## Your object is now in the post! Expect it in 5 to 10 weeks
```

```
post(my_address)
```

```
## Your object has been posted to  4   Pleasant Drive   London !
```

We can see that in the top example, `post()` has dispatched the `post.default()` method. In the second example though, it's dispatched the `post.address()` method.

As you can see, the S3 system is quick and easy. For smaller projects, this should be more than enough to get you by. Because it's quick and easy though, it's not particularly sturdy. You can change the class of any object, which will change the methods that are called on it and so can really mess things up. If you're careful though, creating custom objects in your projects can be very powerful.

4.4 Expressions

When we use R, we write code which is then passed to the console to be executed (evaluated). Before the code is executed though, it is just an **expression**.

An expression can therefore be defined as a section of R code that has not yet been evaluated. That does not mean that all expressions have to be *valid*. For example, a piece of code like this `mean()` is a valid expression, but will error when it is evaluated because `mean` is missing its required arguments.

Expressions themselves are made up of 4 constituent parts: calls, constants, names and pairlists. For now though, we're not going to look at the bits that make up expressions, but instead we'll focus on expressions as a whole.

4.4.1 Creating expressions

Creating an expression (an unevaluated piece of code) is done in base R using the `quote()` function. Unfortunately, the `expression()` function in R doesn't actually create an expression, so use `quote()` instead.

When creating single line expressions, you can just provide the expression directly within the `quote()` function:

```
quote(x + 10)
```

```
## x + 10
```

When providing multiple line expressions, wrap the argument in `{}` like this:

```
quote({
  x + 10
  y - 5
})
```

```
## {
##     x + 10
##     y - 5
```

```
## }
```

Unfortunately, testing whether something is an expression in R isn't that easy, because the base R functions are made for the constituent parts of the expression (e.g. `is.call()`, `is.name()`, etc.). Instead, you can use the `is_expression()` function from the `rlang` package to test whether something is an expression:

```
rlang::is_expression(
  quote(1 + 1)
)
```

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

4.4.2 Evaluating expressions

Once you've created your expression, you can evaluate it using the `eval()` function:

```
my_expr <- quote(1 + 1)
eval(my_expr)
```

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

Of course in this example, this is essentially just the same as `1 + 1` as we're evaluating the expression in the same environment in which it was created. However, the `eval()` function accepts an `envir` parameter where you can pass an environment for the expression to be evaluated in:

```
new_environ <- new.env()
new_environ$w <- 10
my_expr <- quote(w + 5)
eval(my_expr) # this will error because x doesn't exist in our parent environment
```

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

```
eval(my_expr, new_environ) # this won't error because the provided environment has an object name
```

```
## [1] 15
```

Using this, you can create expressions in one environment without evaluating them, and then evaluate them later in different environments to where they were created.

4.4.3 Substitution

As well as hard coding in the objects and names in our expression, we can substitute in values from our environment. For example, lets say we wanted to create an `x <- y + 1` expression, but we wanted to change what the value of `y` was

when we created it. We could acheive this by using the `substitute()` function. `substitute()` requires two paramters, `expr` which must be an expression, and `env` which must be an environment or a list and contains the objects you want to substitute.

```
substitute(x <- y + 1, list(y = 1))
```

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

As you can see, this doesn't *evaluate* the expression, it simple substitutes the provided names with the values provided in the `env` parameter. This can be a really powerful tool for building up expressions.

4.4.4 Quasiquotation

A related subject to expressions and substitution is the idea of **quasiquotation**, used heavily in the `tidyverse` packages. Quasiquotation is the process of quoting (creating expressions) and unquoting (evaluating) parts of that expression.

A good example of quasiquotation in action is the `dplyr` package. Within the `dplyr` package functions, you'll provide column names to various analysis and data manipulation functions. When you provide those names however, you provide them as raw names (i.e. not in quotation marks): `dplyr::mutate(data, new_column = old_column + 1)`. Those column names are then quoted (as in `quote()`) and then evaluated in the context of the dataset that you've provided.

I won't go into quasiquotation here because Hadley's chapters on the subject in his Advanced R book summarises the topic much better than I ever could. But if you're interested, I would recommend using the `tidyverse` packages and trying to understand how quoting and unquoting has been implemented in those packages. If you can get your head round it and even implement similar ideas in your own projects, you can greatly expand your flexibility and efficiency.

4.5 If / Else

Building on our logical operators, there will often be times where you want to split the logic of your code depending on a criteria. For example, if you've created a function that can accept a character string or a number, you might want to split the body of the function to do something slightly different depending on the class of the provided argument.

If / else statements in R has a simple structure:

```
if (criteria_statement) {
  what_you_want_to_do
} else if (other_criteria) {
```

```
  something_else_you_want_to_do
} else {
  something_you_want_to_do_if_all_else_fails
}
```

Putting this into practice, a real If / else block may look like this:

```
x <- 1
if (x == 1) {
  return("x is 1")
} else if (x == 2) {
  return("x is 2")
} else {
  return("x is not 1 or 2")
}
```

```
## [1] "x is 1"
```

Implementing this in a function could look like this:

```
what_is_it <- function(x) {
  if (is.character(x)) {
    return("x is a character")
  } else if (is.numeric(x)) {
    return("x is numeric")
  } else {
    return("x is something else")
  }
}

what_is_it("hello")
```

```
## [1] "x is a character"
```

```
what_is_it(2)
```

```
## [1] "x is numeric"
```

```
what_is_it(TRUE)
```

```
## [1] "x is something else"
```

4.6 Iteration

Functions are an important stepping stone in reducing the amount of code you need to do something (making your code less **verbose**). Another tool in acheiving this goal is the idea of iteration. Iteration just means the process of doing something more than once.

We'll often find ourselves doing the same thing again and again in programming. Calculating the means for lots of different columns, or different datasets, or making plots for example are operations that you'll rarely only do once. There are two approaches to this: imperative programming and functional programming. First, we'll take a look at imperative programming (for loops) as this is more akin to lots of other programming language. Later, however, we'll look at how we can better utilise the fact that R is a functional programming to solve some iteration problems more easily and with fewer errors.

4.6.1 Imperative programming

4.6.2 For loops

For loops are almost completely ubiquitous across different programming languages. They allow us to perform actions over a list or set of objects. They are flexible and explicit even if they can be difficult to understand. A simple for loop in R follows a simple structure:

```
for (identifier in list) {
 do_something_with_it(identifier)
}
```

In the above, the identifier is used to access the value that is currently being iterated upon. Let's look at a practical example:

```
for (i in c(1,2,3)) {
  print(i)
}
```

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

In this loop, we go through the vector of values 1, 2 and 3, and we print it using the `i` identifier that we've assigned to the value we're currently iterating upon. So the code inside the loop will run three times (once for each value in the vector we've provided). The first time, `i` will equal 1. The code will execute, and then `i` will take the value of 2 and so on.

This isn't a particular useful example. Let's look at more realistic example. Let's say you've got a list with 2 dataframes in that have the same structure:

```
dataframe_list <- list(
                    data.frame(
                      obs = c(1,2,3),
                      value = c(10,11,9)
                     ),
                     data.frame(
```

```
                    obs = c(1,2,3),
                    value = c(100,200,150)
                  )
                )
```

And you want to calculate the mean for the `value` column for each one:

```
for (df in dataframe_list) {
  mean(df$value)
}
```

Or, if we refer back to our function example from the Functions chapter, we could apply our function to each dataset to get a normal distribution for each. Rather than just printing those numbers, we'll also construct a list to store the output:

```
output_list <- list()
for (df in seq_along(dataframe_list)) {
  output_list[[df]] <- create_norm_dist_from_column(dataframe_list[[df]], "value")
}
head(output_list[[1]], 5)
```

```
## [1] 10.426583  8.005476 10.348768 11.836536  9.236816
```

In this case, rather than looping through the *values* in the `dataframe_list` list, we're looping through the *indices* using `seq_along()`. This lets us keep track of where we are in the `dataframe_list` so we can assign the output to the right position in `output_list`.

Note: If you can avoid expanding a list (i.e. increasing the size of a list by one everytime to add to it), then this is preferable. However, for very small lists, expanding a list rather than pre-defining its size and filling those values isn't really a big deal.

4.6.3 While loops

While loops are closely related to for loops. Instead of looping through an object or through the indices of an object, a while loop runs when a criteria is fulfilled:

```
x <- 0
while(x < 2) {
  x <- x + 1
  print(x)
}
```

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

4.6.4 Functional programming

For loops and its derivatives are very powerful, but they are arguably less important in functional languages like R than they are in other languages like C and Python.

Instead, within R we can leverage functions to wrap the loops. For example, let's look back at our `dataframe_list` example. We can turn our loop into a function that we can call whenever we have a list of dataframes:

```
norm_dists <- function(dfs, column = "value") {
  output_list <- list()
  for (df in seq_along(dfs)) {
    output_list[[df]] <- create_norm_dist_from_column(dfs[[df]], column)
  }
  output_list
}

head(
  norm_dists(dataframe_list)[[1]], 1
)
```

```
## [1] 10.3629
```

Now, whenever we have a list of dataframes we can just use our `norm_dists()` function and provide the column name to the `column` parameter.

Chapter 5

Exercises

Here are some interactive exercises covering the basic concepts we've looked at in the last X modules.

This exercises are hosted on a Shiny server here