Working with R

Steph Locke (Locke Data)

Contents

Preamble

About this book

Welcome to the first book in Steph Locke's R Fundamentals series!

This first book introduces the R language and the RStudio coding environment. It is by no means comprehensive but it represents the first steps in learning the modern fundamentals.

At the end of this book, you'll be comfortable with how R works and deciphering a lot of the old-school code out there.

Already know the basics of R? You'll be able to skip right onto the next book in this series, Data Manipulation in R[1].

Working with R by Stephanie Locke is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

What you need to already know

This book assumes no prior knowledge of programming, or R. A basic knowledge of mathematics will come in handy, but isn't re-

[1] `http://geni.us/datamanipulationinr`

Figure 1

quired. You will need a good understanding of the use of calculators or Excel for mathematics.

You will need the required access to be able to install (or get installed) R, RStudio, and, if on Windows, Rtools. You can code online at r-fiddle.org[2] but this might be unreliable.

- Install R from r-project.org[3]
- Install RStudio from rstudio.com[4]

If you're on Windows, it's also great if you can install Rtools[5]. This gives you the tools to install packages from GitHub later on down the line.

Steph Locke

I am a Microsoft Data Platform MVP with over a decade of business intelligence and data science experience.

Having worked in a variety of industries (including finance, utilities, insurance, and cyber-security,) I've tackled a wide range of business challenges with data. I was awarded the MVP Data Platform award from Microsoft, as a result of organising training and sharing my knowledge with the technical community.

I have a broad background in the Microsoft Data Platform and Azure, but I'm equally conversant with open source tools; from databases like MySQL and PostgreSQL, to analytical languages like R and Python.

Follow me on Twitter via @SteffLocke

Locke Data

I founded Locke Data, an education focused consultancy, to help people get the most out of their data. Locke Data aims to help organisations gain the necessary skills and experience needed to

[2] `http://www.r-fiddle.org/`
[3] `https://cloud.r-project.org/`
[4] `https://www.rstudio.com/products/rstudio/download/#download`
[5] `http://cran.r-project.org/bin/windows/Rtools/`

build a successful data science capability, while supporting them on their journey to better data science.

Find out more about Locke Data at itsalocke.com[6].

Acknowledgements

This book could not be possible without the skills of Oz Locke. My husband he may be, but most importantly he's a fantastic editor and graphic designer.

As well as Oz editing the book and helping me make it look awesome, plenty of people provided vital feedback on the book:

- Erin Grand
- Eric Nantz
- Edmund Poillion
- Duncan Greaves
- Robert Fornal
- Verena Haunschmid
- Donia Robinson
- Paula Jennings
- Nico Botes
- Edafe Onerhime

Any errors are my own but Oz and my feedback group have helped me make substantially less of them. Thank you!

Conventions

Throughout this book various conventions will be used.

In terms of basic formatting:

- This is standard text.
- `This is code or a symbol`
- Keyboard keys will be shown like Ctrl+⇧+F ctrl + shift + F
- **This is the first time I mention something important**

[6] `https://itsalocke.com/company/aboutus/`

This is a book about coding, so expect code blocks. Code blocks will typically look like this:

```
"this is a code block"
```

Directly underneath it, normally starting with two hash symbols (`##`) is the result of the code executing.

```
## [1] "this is a code block"
```

There will also be callouts throughout the book. Some are for information, some expect you to do things.

Anything written here should be read carefully before proceeding.

This is a tip relating to what I've just said.

This is kind of like a tip but is for when you're getting into trouble and need help.

This is something I recommend you do as you're reading.

This let's you know that something I mention briefly will be followed up on later, whether in this book or a later one.

If you're reading the book in print format, there are often blank pages between chapters – use these to keep notes! The book is to help you learn, not stay pristine.

Feedback

Please let me and others know what you thought of the book by leaving a review on Amazon!

Reviews are vital for me as a writer since it provides me with extra insight that I can apply to future books. Reviews are vital to other people as it gives insight into whether the book is right for them.

If you want to provide feedback outside of the review mechanism for things like errata, suggested future titles etc. you can. Use bit.ly/ldbookfeedback[7] to provide us with information.[8]

[7] `http://bit.ly/ldbookfeedback`
[8] You'll get a sticker if you do!

Part I

R at a high level

Chapter 1

About R

R is an open source language released in 2001 that's ideal for data wrangling[1] and data science[2]. It has connectors to pretty every much every data source under the sun, allows you wrangle data like nobody's business, build pretty much every type of model ever thought up, and visualise it in all the niftiest ways.[3] R really is a tremendous language to add to your tool belt.

1.1 History

R has a long lineage - it was written to re-implement the language S[4]. S was a commercial language written in the mid-1970s to enable statistical and graphical processing. Indeed much code written in S can still run today, a phenomenal feat! If you ever look at R and wonder "Why on Earth does it work like that?", the usual answer is "Because S".[5]

The previous paragraph might lead you to ponder why R is in use today and why the popularity is growing. R has a vibrant

[1] `https://en.wikipedia.org/wiki/Data_wrangling`

[2] `https://en.wikipedia.org/wiki/Data_science`

[3] These superlatives are not disingenuous, R really is that broad and amazing.

[4] `https://en.wikipedia.org/wiki/S_(programming_language)`

[5] If you'd like to find out about the assignment operators' history (`<-` and `->`) and many of the other quirks of R, the article Rbitrary `http://ironholds.org/projects/rbitrary/` is fantastic and highly irreverent reading.

ecosystem that enables people to extend, enhance, and replace any part of it. There are many paradigms in R to facilitate object-oriented programming, functional programming, and more. If you can write something in C++, FORTRAN[6], Python, or JavaScript–and of course, R!–you can write extensions for R.

There are currently more than eleven thousand extensions (referred to as packages) to R in the core ecosystem (which is a fancy word for the collected bits and pieces of R!) [7] and two and a half thousand packages in the genomics ecosystem [8].

We're also seeing emerging ecosystems and paradigms within CRAN. The tidyverse[9] is one such ecosystem, focussed primarily on analysing tabular data, and it will be used in future works extensively.

1.2 CRAN

The core ecosystem is **CRAN**, the Comprehensive R Archive Network. CRAN[10] is maintained by some great people who put in place a large number of quality gates that an R package must adhere to in order to be made widely available. They then host these packages and do great things like daily re-runs of all package tests to ensure packages are still working. CRAN is the default source of packages for most R users.

If you use RStudio, you'll use a **mirror** of CRAN hosted by RStudio. There are a number of these mirrors scattered over the globe to help reduce the load on the central servers. You can use another one of these mirrors, or even set-up your own internal CRAN.

1.3 Key points to know about R

- R works in-memory which means that the processing is fast but the amount of data you can process is limited to how

[6] Yes, it still exists and yes, R still relies on it for some key algorithms
[7] CRAN `https://www.r-project.org/`
[8] BioConductor `https://www.bioconductor.org/`
[10] `https://cran.r-project.org`

much RAM your data takes up and how much your computations will require.
- R is not multi-threaded by default, it works on a single CPU core. Making use of more than one of your cores to spread the load requires additional packages and often additional coding.
- R is quirky! In some ways, R is a lot like other common programming languages, which can make it pretty easy to pick up. However, because R is still designed to be compatible with S, it's actually pretty darn old and as a result, really odd in places.
- Coding R will give you the typical gotcha's, and add another: case sensitivity. R is (un)fortunately a language where "Red" and "red" are different and this also extends to variable and function names (which we'll discuss later.) As a consequence, the most common errors you'll find when writing code in R are:
  - Incorrectly placed or missing commas
  - Incorrectly placed or missing brackets
  - Incorrectly placed or missing operators
  - Incorrect case used when typing
- With so many packages available to extend R, the answer to "how do I write this?" is usually "there's a package for that".

We'll look at finding packages later in this book.

1.4 Summary

R is a great language for doing data analysis, data science, and more. It has its quirks but the community around it is huge and is making R easier to adopt every day.

Chapter 2

Why use R?

R as a programming language is brilliant at its core competencies – statistics and data visualisation. It's also a great "glue" language, by which I mean that you can use it to perform computations in many different languages and combine the results smoothly. As a result, R enables you to be an effective data wrangler, data scientist, and/or data visualisation practitioner.

The following section will show some uses that exemplify how little is required to do things in R, that in other languages or tools can take a substantial amount of time.

These are illustrative only. Don't worry if they don't make much sense. Writing code like these examples will be covered in later books.

2.1 Data wrangling

Here's a common issue I've had in the past: working with data from multiple sources, that should usually conform to a template, but don't. You end up with a whole stack of files that don't *quite* match the template, and as a result getting all this data read, combined, and then output somewhere else is often an incredibly difficult task.

This snippet of R code performs the following steps:

1. Make functionality from the `tidyverse` available
2. Identify files needing to be read and combined
3. Read each file individually, whilst applying a column type enforcement
4. Combine the results

Combining the results involves matching the columns by name. It will create new columns when it finds additional columns in some of the data and fill in the data. When columns are missing from a dataset it will put NAs in those column for that data.

```
library(tidyverse)

# Change "data" to where your files are.
# Remove the col_types bit if your columns
# are fairly type safe.
list.files("data", full.names = TRUE) %>%
  map_df(read_csv, col_types = cols("c")) %>%
  bind_rows() %>%
  nrow()
```

```
## [1] 105
```

The example is fairly simple in that it's reading in CSVs from a single directory. The great thing is you can consume data from multiple directories or locations by changing how you identify what files you want to process and you can process other formats simply by changing to the relevant read function for that file type.

2.2 Data science

This section is for people with an interest in using R for Data Science. As such it does presume some prior knowledge of Data Science in general. If this section isn't for you, please skip ahead to the next section.

When building a predictive model you typically need to prepare your data, sample it, scale numeric values, perform feature reduction, and build a model. In academic settings the model is the end goal but in an industry setting we often need to put the model into

an operational system. This means any transformations need to be consistently applied to new data before the model can be used to make a prediction.

This can be pretty awkward to write. If you want to scale numeric variables, you need to keep some sort of record of the mean and standard deviation in the case of a z-score[1] or the minimum and maximum values if you wanted to do a minmax score[2]. When you go on to predict values for new data you need to be able to use the same values from your original scaling process otherwise you get the wrong results. How do you then store and apply these values to new data? Probably with a lot of code and manual labour!

Having many steps to take new data, scale values, remove the extra features and get a score means there's many things that can go wrong or get missed. One of the best things you can do is to avoid multiple steps and do things in one go. You need something that can be applied to new data that does all the steps in one.

It turns out, there's an R package for that! You can use functionality from the package caret[3] to add data transformation steps to your model development. It will then keep these steps as part of the model and whenever you use the model to make predictions it will first process the data based on the parameters and transformations used on the training data.

This snippet of R code performs the following steps:

1. Make functionality from the `caret` package available
2. Split data into training and test samples with the outcome column separate
3. Build a linear regression model on data scaled using z-scores and turned into principal components
4. Apply the transformations and then our linear model to get predictions for test data

```
library(caret)

training_data    <- mtcars[1:20,-2]
training_outcome <- mtcars[1:20,2]
```

[1] http://stattrek.com/statistics/dictionary.aspx?definition=z%20score

[2] https://en.wikipedia.org/wiki/Feature_scaling#Rescaling

[3] https://topepo.github.io/caret/

```
test_data        <- mtcars[21:32,-2]
test_outcome     <- mtcars[21:32,2]

model <- train(training_data, training_outcome,
               method="lm",
               preProcess = c("scale","center","pca"))
predictions <- predict(model, test_data)
```

Using `caret` means you can end up with a single line of code to take new data and get an output – making it incredibly easy to use in an operational scenario.

2.3 Data visualisation

Data visualisation is an area where R makes it especially difficult to choose just a single example. Instead of showing the many fancy or interactive visualisations you can do in R, I'm instead going to show a part of the workflow.

You'll often need to make a chart, and then make that chart for a lot of different datasets. These could be datasets for different customers, samples, or time slices.

You can make a chart in R using the package ggplot2[4] to build a chart of all data.

I'm going to use Dino the Datasaurus and his Data Dozen buddies[5] as example data here. Dino and his friends have the same summary statistics but very different data distributions. They show why visualising data is so incredibly vital.

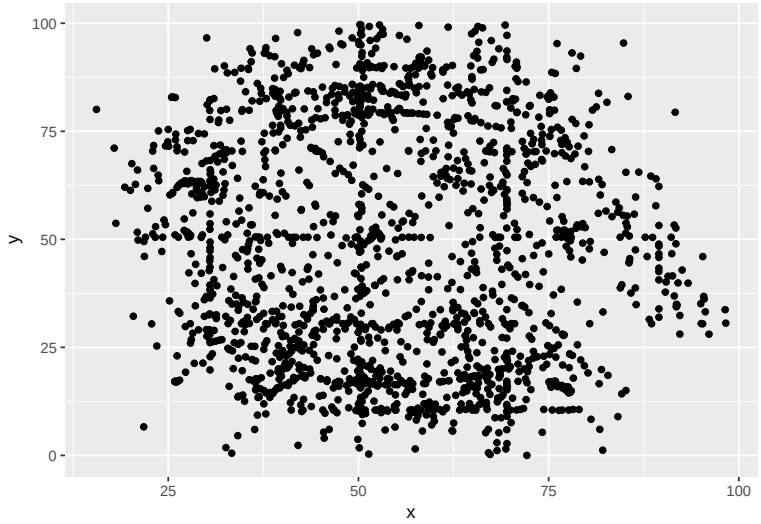
The other Dozen were generated using simulated annealing and the process is described in the paper *Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing* by Justin Matejka and George Fitzmaurice. autodeskresearch.com[6]

[4] `http://ggplot2.tidyverse.org`

[5] The original Datasaurus was created by Alberto Cairo and can be found on thefunctionalart.com

In the paper, Justin and George simulate a variety of datasets that have the same summary statistics to the Datasaurus but have very different distributions.

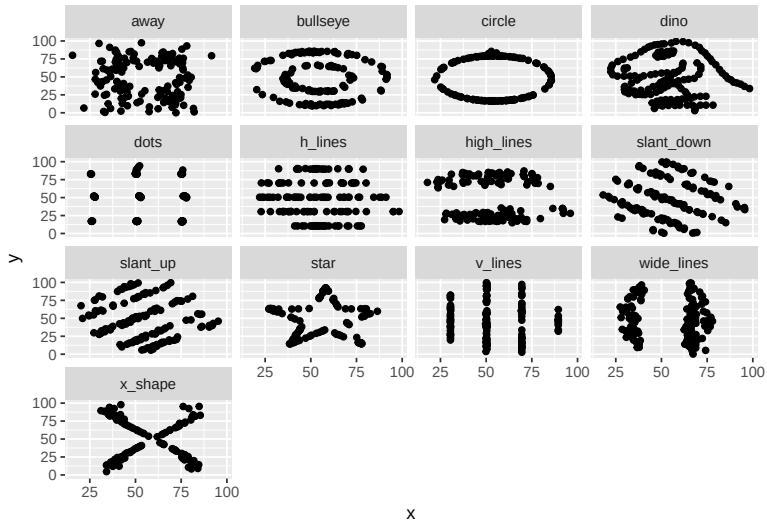
```
library(ggplot2)
library(datasauRus)

myPlot<- ggplot(datasaurus_dozen, aes(x,y)) +
          geom_point()
myPlot
```



You might just want to make it easy to see Dino and his friends. You can do this with a small multiples[7] chart, where you get a small chart per group.

```
myPlot + facet_wrap(~dataset)
```

[7] https://en.wikipedia.org/wiki/Small_multiple



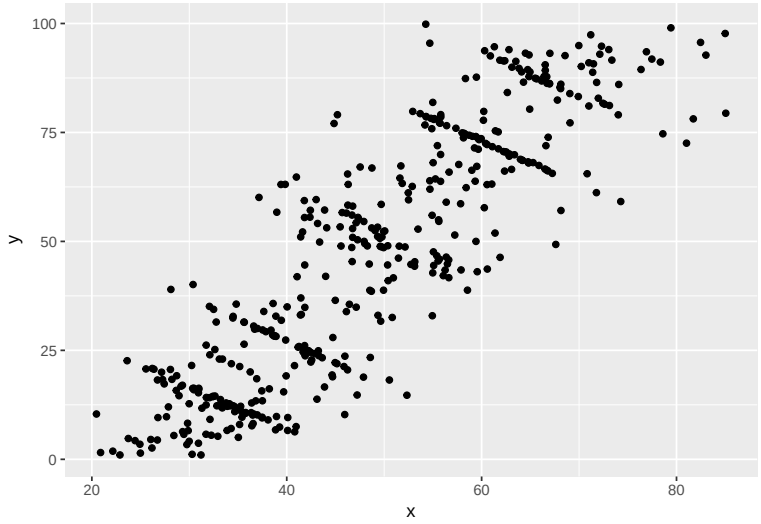
Instead of many small charts in one file, you might want to produce a separate chart for for Dino and each of his buddies in a bigger piece of analysis. In a real world setting, you might want to loop through customers and produce a chart per customer. Alternatively, you might want to make a chart object and re-use it for many different slices of data in a big piece of analysis.

We made and stored a chart object (`myPlot`) and we can actually pass it some data that will override the data currently linked to the chart. This enables to re-use the chart for different datasets.

For instance, another dataset in the `datasauRus` package is a set that illustrates the Simpson's Paradox[8] It has the same three columns (x, y, and dataset) so it can be used with `myPlot`.

```
myPlot %+% simpsons_paradox
```

[8]Simpson's Paradox is the phenomenon where high level statistics provide one conclusion but evaluating sub-groups within the data provides a very different conclusion. More information on Simpson's paradox can be found at vudlab.com/simpsons .



2.4 Summary

R is an incredibly powerful tool. It is useful in range of situations and is well worth learning. Reading this series of books will help you learn to be able to do the sorts of things outlined in this section.

Chapter 3

Using RStudio

If you don't already have it, you should install R and RStudio[1].

If you're on Windows, it's also great if you can install Rtools[2]. This gives you the tools to install packages from GitHub later on down the line.

RStudio[3] is a coding interface to R that makes it easier for you to be productive.[4] I'm devoting substantial amounts of this book to your working environment as you can use it to make learning and coding R much easier by taking the time to understand it.

The interface will be split up into a top menu and then four panes, although only three may be visible when you first start RStudio.

3.1 The console

The (bottom) left hand section is the console. This is where you can execute R code directly.

To use the console you type some code alongside the `>` and hit Enter ↵ for the code to be executed. The result will then appear

[3] `http://rstudio.com`

[4] If coding interfaces were game modes, RStudio is Easy mode, Visual Studio is Normal, R-GUI is Hard, vim is Insane, and Emacs is Legendary.

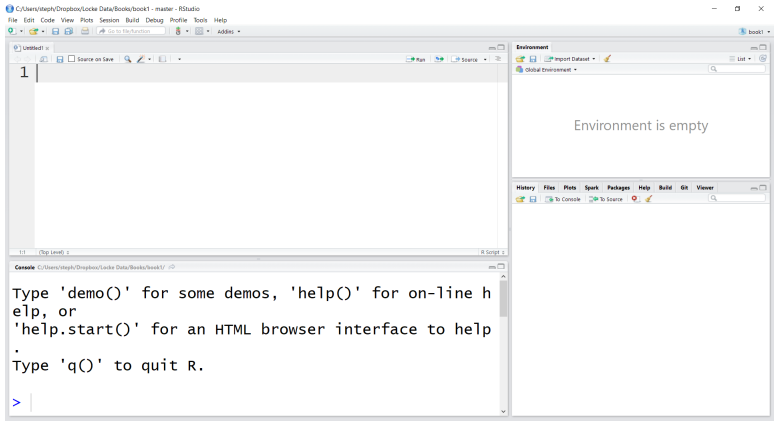


Figure 3.1

underneath your line of code.

Watch a video of using the console at youtu.be/2hg1Qg7uLwU[5].

Errors, warnings, and messages will also appear in the console. We'll discuss what these are later in the book

Use the console to add two numbers together.

If the code you entered wasn't a complete statement e.g. `1 + 2 +`, when you hit Enter ↵, you'll get a new line only the > will now be a +. This indicates the code you're writing is a continuation of the previous line. R will allow you to continue building up a complete chunk of code this way. It'll run all the lines you entered as one block once it's been completed.

If you want to clean your console and start afresh, hit Ctrl + L Ctrl + L to remove whatever has been executed in the console this session.

You can use your up and down arrow keys to navigate through previous code you've written and executed.

[5] `https://youtu.be/2hg1Qg7uLwU`

If you commit one of the most common coding errors (incorrect amounts or places of brackets and commas) you might end up with an incomplete line and basically lock your console into having the `+` symbol at the beginning of each line. If you find yourself stuck with commands just writing and writing and never executing, hit the Esc key to cancel the code and get back to the standard cursor.

3.2 Scripts

RStudio allows you to create and work with files containing code. These files give you a way to store and manage your code.

The most common file types you might use are R files (`.R`) and rmarkdown files (`.Rmd`).

rmarkdown files are for generating documents with text and R code interleaved (like this book!) They'll be covered in a future book. If you want to get going, checkout the rmarkdown[6] site.

You can create one of these files by going to *File > New > R Script*, the New File button, or with the hotkey combo[7] of Ctrl + Shift + N Ctrl+⇧+N.

Watch a video of how to do this at youtu.be/rWHV2VlQo2w[8].

In an R script you can type code and execute it by hitting Ctrl +↵ Ctrl + Enter[9], or selecting the code to run and hitting the Run button.

You can execute all the code in a script by hitting Ctrl+⇧+↵ Ctrl + Shift + Enter or hitting the Source button.

1. Open a new R script
2. Write `10*5` in it and execute the code

[7] I'm a big fan of hotkeys!

[8] `https://youtu.be/rWHV2VlQo2w`

[9] This hotkey is really nifty as it'll find the start and end of a block of code and send it all to the console to be executed.

3.3 Code completion

Whilst writing scripts or typing in our console, we can get help and be more productive by using **code completion**. Code completion will pick up from what we've typed so far and provide a navigable list of suggestions.

As we navigate through the list, it'll provide help text where possible and then it will complete the code we were typing.

- You access the code completion by hitting ⇤⇥ Tab whilst typing
- Once it's up you can keep typing to refine the list
- Your arrow keys allow you to navigate the list ↕
- Hit Esc Esc to back out of the completion capability
- Hit ⇤⇥ Tab to accept whatever value in the list is currently highlighted

Watch a video of how to do this at `youtu.be/pGOF4gTyeXA`.

1. On a new line of your script, type `a` and activate your code completion. Browse the list then cancel out of the list
2. Overwrite the `a` with an `A` and go back into the code completion. Do you get the same list? What's different and why?

3.4 Projects

So far you've seen R as a scratch-pad (via the console) and for making an isolated script, but a lot of the time we have to put data, multiple scripts, documentation and more into a **project**.

An RStudio project is a folder with an extra file. This file can be used to open RStudio, with everything laid out like it was before you closed the project. It can store preferences to allow projects to vary from the way you normally do things.[10]

[10] Like converting tabs to spaces and the number of spaces characters it should replace with.

At this point in your R coding career, keeping everything where you left off is great. Later on, and especially if you work in anything where reproducibility is valued, you can go to *Tools > Project Options ...* and set the .Rdata fields to "no" so that nothing loads up into memory when you load the project.

You can, and should, create a new R project when embarking on a new area of work. To create a project go to *File > New Project*.

This will popup a dialogue that gives you the option to create a brand new project directory, create one from some existing directory you might already have, or create one with the content of a project in your source control system (we'll talk about source control in a later book.)

Most commonly, you'll want to create a new directory project. Once selected it'll then give you the option to create an empty project, a Shiny project (a feature for creating amazing dynamic reports,) or an R package. You'll normally select the empty projects. Once an option is selected, provide a name and where the project should go.

Watch a video of how to do this at youtu.be/etkSsF6r2iU[11].

Working with source control, shiny, and creating R packages are all in later books.

You can navigate to projects using the project option in the top right corner or in the *File* section.

Create a new project to store the answers to exercises and any code you try out during this book. You don't have to save the script you were working on before this (unless you want to!)

[11] `https://youtu.be/etkSsF6r2iU`

3.5 Summary

RStudio is a fantastic environment for learning R and writing R code going forward.

Using code completion can be a great help in finding what to write and how to write it.

Other areas of the RStudio interface will be introduced as we go forward but taking the time to get to know the environment now will help you be more productive in future.

Chapter 4

Useful resources

Our coding environment, RStudio, is a great help to us. The code completion makes it easy to find things kind of related to what you're typing but sometimes you need to do a bit more digging or read a bit more than the snippet of help in the code completion window.

4.1 The built-in help

R has pretty great built-in help. You might understand some of it but there's usually lots of it and most help files give you examples to run.

The help files are accessible in the bottom right-hand corner of RStudio.

If you want to see some of the built-in help whilst using code completion, you can hit F1. Similarly, you can select a word in script and hit F1 to go to the help.

If you don't have something specific to select from you can go to the help window and use the search facility. If there's a function that matches what you're typing that'll be short-listed, and you can select it and hit ↵ Enter. Alternatively if you don't get a match, hitting ↵ Enter once you've finished typing will enact a fuzzier search.

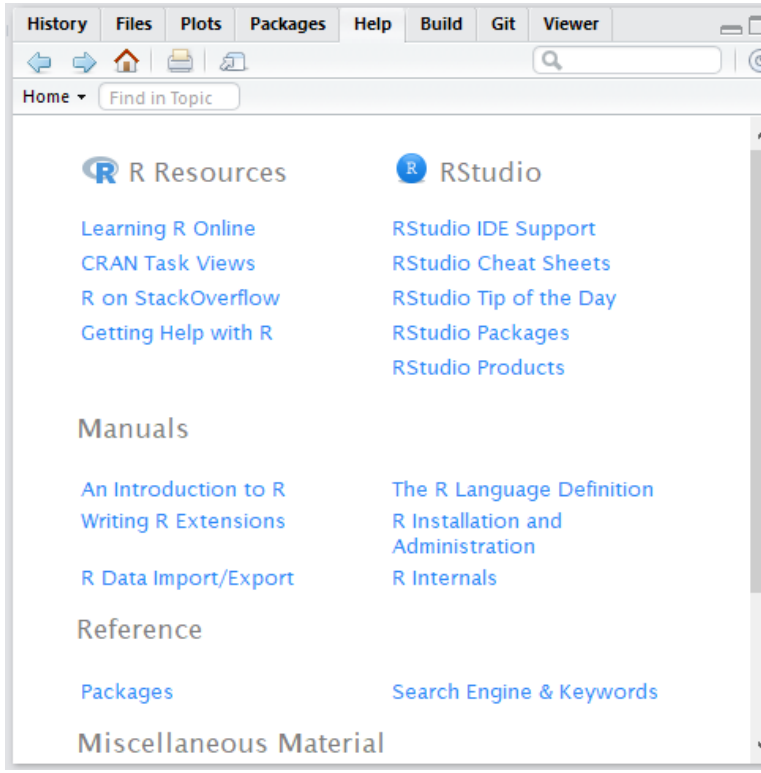


Figure 4.1

Another way you can get help is by looking at the index of functions available for a given package. You go to the Packages tab and click on the package you're interested in. This loads up the index for that package and you can then read through what's available.

The help window has some handy navigation features to make it easier to use:

- In-file search bar for finding words in a help file
- Navigation arrows for moving between files like Back and Forwards on a web-browser
- The New Window button creates a popup with the file so that you make it bigger or put it onto another monitor

4.2 Online

R is a great community that has produced many resources.

- You can search for previous R questions or ask new ones on the ubiquitous Stack Overflow[1]
- If you're tweeting about R, use the hash-tag *#rstats*.
  - If you want to see what's happening in the R world, I recommend you follow Mara Averick (@dataandme)[2]
  - If you want to ask a question, you can also tweet me Steph Locke (@stefflocke)[3]
- RStudio provide a trove of fantastic cheatsheets[4] – including one for being super-productive in RStudio. These are great to download and/or print in order to keep handy as you're learning
- Documentation for R packages is available on CRAN[5] but there are a number of online sites that try to improve the experience, including rdrr.io[6] and rdocumentation.org[7]
- R-bloggers[8] is a site consolidating blogs from more than 500 people. It's a great way to find how-to's
- R Weekly[9] is a curated newsletter of key goings on, new packages, and blog posts from the R world
- DataCamp[10] is a nifty online learning platform for R, Python, and more.

Finally, search is a wonderful thing and you should totally be searching for answers to your questions. Unfortunately, when you're first getting started with R, your Google-bubble[11] won't be very good at returning the results you're hoping for. To improve results, make sure to prefix searches with "R" and you can also use a customised version of Google called rseek.org[12]. Rseek specifically searches the sorts of sites mentioned above which can

[1] `http://stackoverflow.com`
[2] `http://twitter.com/dataandme`
[3] `http://twitter.com/stefflocke`
[4] `http://www.rstudio.com/resources/cheatsheets/`
[5] `http://cran.r-project.org`
[6] `http://rdrr.io`
[7] `http://rdocumentation.org`
[8] `http://r-bloggers.com`
[9] `https://rweekly.org/`
[10] `https://www.datacamp.com/`
[11] Google's tailoring of your results
[12] `http://rseek.org`

be limiting so you trade off breadth for initial accuracy.

4.3 Books

Hopefully this book and it's follow ups will be helpful for learning R! However, the aim of producing a new book every two to three months might be a bit slow for your liking so I've included some great books[13] you can also turn to for learning R.

Additionally, understanding R won't necessarily make you a great data wrangler, data visualiser, or data scientist, or even a great coder so I've put included some recommended books on topics outside of R.

4.3.1 R

- **R for Data Science: Visualize, Model, Transform, Tidy, and Import Data** An end-to-end introduction to R geni.us/rfords[14]
- **Text Mining with R: A Tidy Approach** Learn how to do Natural Language Processing in R geni.us/tidytext[15]
- **ggplot2: Elegant Graphics for Data Analysis** Learn the ins and outs of one of the most common graphical packages for R geni.us/ggplot2[16]

4.3.2 Data wrangling

- **SQL Cookbook** Helps you with SQL skills and being able to translate across different database dialects geni.us/sqlcookbook[17]
- **Data Manipulation in R** Book 2 in this series focuses on getting and analysing data in R geni.us/datamanipulationinr[18]

[13] All the links are links that will ideally take you to the most relevant Amazon site. I apologise if you're reading this and I was unable to to save you the hassle of digging out the book online. Thank you for reading this one though!

[14] `http://geni.us/rfords`

[15] `http://geni.us/tidytext`

[16] `http://geni.us/ggplot2`

[17] `http://geni.us/sqlcookbook`

[18] `http://geni.us/datamanipulationinr`

4.3.3 Data visualisation

I have many of these books, but others are recommendations from others and are now on my list to buy!

- **Show me the numbers** geni.us/showmethenumbers[19]
- **The Visual Display of Quantitative Information** geni.us/visdispquantinfo[20]
- **Information is Beautiful** geni.us/infoisbeautiful[21]
- **The Truthful Art: Data, Charts, and Maps for Communication** geni.us/truthfulart[22]
- **Data Visualisation** geni.us/datavisualisation[23]
- **Storytelling with Data** geni.us/storytellingwithdata[24]

4.3.4 Data science

- **Naked Statistics: Stripping the Dread from the Data** An engaging introduction into some statistical concepts geni.us/nakedstatistics[25]
- **Statistics in Plain English** A primer on statistics geni.us/statsplainenglish[26]
- **Data Science for Business** A non-technical introduction to data science and some of the maths geni.us/dsforbiz[27]
- **Forecasting: principles and practice** Learn all about forecasting methods (code is in R too!) geni.us/forecasting[28]
- **Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis** geni.us/regression[29]
- **Artificial Intelligence for Humans, Volume 1: Fundamental Algorithms** This and subsequent volumes are useful for getting to grips with some of the fancier bits of data

[19] http://geni.us/showmethenumbers
[20] http://geni.us/visdispquantinfo
[21] http://geni.us/infoisbeautiful
[22] http://geni.us/truthfulart
[23] http://geni.us/datavisualisation
[24] http://geni.us/storytellingwithdata
[25] http://geni.us/nakedstatistics
[26] http://geni.us/statsplainenglish
[27] http://geni.us/dsforbiz
[28] http://geni.us/forecasting
[29] http://geni.us/regression

science happening at the moment geni.us/aiforhumans[30]

4.3.5 Miscellaneous

- **The Checklist Manifesto: How to Get Things Right** Learning about how you can be effective is important to becoming effective geni.us/checkman[31]
- **The Phoenix Project** A novel on a painful transition to agility and adding value geni.us/phoenixproj[32]
- **The Mythical Man-Month: Essays on Software Engineering** Words of wisdom that are still very applicable today for working on data science projects geni.us/mythicalman[33]

4.4 In-person

The R community, as well as doing a huge amount of tweeting, actually gets together in-person quite a bit.

If you'd like to go a meetup, then you should check out this meetup directory[34]. As well as these happenings, you can also check out R-Ladies[35] events.

We have a growing number of conferences and you can find ones to attend via the conferences directory[36].

4.5 Summary

There are so many resources available to people learning R. The community is awesome and you can get involved in any medium that suits you. If you're looking for something specific and finding it difficult to hunt down a suitable resource, tweet me (@SteffLocke)!

[30] `http://geni.us/aiforhumans`
[31] `http://geni.us/checkman`
[32] `http://geni.us/phoenixproj`
[33] `http://geni.us/mythicalman`
[34] `http://jumpingrivers.github.io/meetingsR`
[35] `https://rladies.org/`
[36] `https://jumpingrivers.github.io/meetingsR/events.html`

Part II

R building blocks

Chapter 5

R data types

I recommend you add a new file to your R project, save it with a file name referencing this section and try out the code. Add your answers to the exercises and leave yourself some notes by first putting `#` sign and then typing the note after it. There's nothing like practice and taking notes for helping you retain info!

When we think of different bits of data, some of it might be numbers, text, dates, and more. R has its own set of these **data types**.

Before we get into the data types, let's see how we can get R to tell us what something is.

R uses **functions** (basically, an inbuilt bit of code that you can call on to do things with, optionally passing it some data to work with) to take some inputs and get an output. The function that we can pass a value to, and get what data type it is as the output, is the `class()` function.

```
class(1)
class(1.1)
class("1")
```

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

```
## [1] "character"
```

You can use this `class()` function if you're ever unsure what data type something is. This is great for when you're getting unexpected results!

5.1 Numbers

I gloss over a lot of the nuance here as most people will not need it. If you want some of the nuance, read the footnotes.

Numbers are split into a few different types:

- **integer**s are whole numbers like 1 or 42[1]
- **numeric**s are numbers that have a decimal portion associated with them like 1.0 or 3.133[2]
- **complex number**s are numbers that make use of the imaginary number i like 4i[3]

5.1.1 Converting to numbers

The functions `as.numeric()` and `as.integer()` allow you to convert something stored as text into a number.

These functions will give you some red text as a warning if you attempt to convert something to a number that can't be safely converted. It will still attempt to perform the conversion, but return missings (`NA`) instead of actual values.

[1] If you want to guarantee a number is an integer, you can suffix the value with a L e.g. `42L`. If you want to read more about this, check out the R manual at `http://cran.r-project.org`

[2] Numerics in R are floating point numbers - this mean every decimal gets stored usually with a large amount of extra decimal places. This can lead to some unusual results when comparing two decimal values and we'll see an example later.

[3] i is the square root of -1, which is an impossible number since any negative multiplied by itself would result in a positive. Descartes coined the term "imaginary" in reference to this number as it's a consistent value in formulae but doesn't exist in the real world.

```
as.numeric(1)
as.numeric(1.1)
as.numeric("1")
as.numeric("r")
```

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

```
as.integer(1)
as.integer(1.1)
as.integer("1")
as.integer("r")
```

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

```
## [1] 1
## [1] 1.1
## [1] 1
## [1] NA
## [1] 1
## [1] 1
## [1] 1
## [1] NA
```

5.1.2 Checking numbers

You can write checks to see if something is numeric, or an integer, with `is.numeric()` or `is.integer()`.

```
is.numeric(1)
is.numeric("1")

is.integer(1)
is.integer(1L)  # There's a footnote on this
is.integer("1")
```

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

We could also use `class()` here and inspect the result.[4] I'm going to use it here to test the results of a conversion to show you how you can **nest** functions, which means that the inner-most function gets evaluated and the results are used by the next outer-most function, and so on.

```
class(numeric("1"))
class(integer("1"))
```

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

5.1.3 Special numbers

As well as `i` to denote imaginary numbers, there are some additional symbols you might encounter or want to use.

- `pi` = 3.1415927
- `Inf` represents positive infinity. You'll often see this if you divide a positive number by zero
- `-Inf` represents negative infinity. You'll often see this if you divide a negative number by zero
- `NaN` is what happens when you really screw up a calculation and do something like `0/0`. It means the result is not a number!

5.2 Text

Text, also known as strings, is split up into two core types:

- **character**s are text as we typically think of it like "red"
- **factor**s (and the subtype **ordered factor**s) are a text type where the number of unique values is constrained e.g. the values are selected from a dropdown. Factors work by storing numbers that correspond to our values and then printing

[4] You might recall that `class(1)` had the result of "numeric" - R was not by default considering 1 as an integer for the purpose of the `class()` function. This is a property of R's evaluation of values and you can force it to consider a value to be an integer by suffixing it with an L, so `class(1L)` evaluates to "integer".

these values. This is much more space efficient when the number of unique values is low.[5]

Factors will be covered at length in a later book. The rest of this book will work with characters.

In R, you can't just type some text as it will be construed as an object or function name. To **delimit** a string you can use speech marks (") or apostrophes (') at the beginning and end of it to show where it starts and ends. These are the **text delimiters** in R.

Note you can't use the two delimiters interchangeably e.g. "red', but you can use them together to enable you have speech marks or apostrophes inside a string e.g. `'They said "Read this"'` or `"It's mine now"`.

If you need to have both inside a string you can **escape** the ones on the inside of a string to say they don't count as text delimiters. To escape a delimiter you can use a backslash(\) e.g. `"They said \"Read this\""`.

```
'They said "Read this"'
"It's mine now"
"They said \"Read this\""
```

```
## [1] "They said \"Read this\""
## [1] "It's mine now"
## [1] "They said \"Read this\""
```

Beware the copy & pasting (C&Ping) of code that isn't in "pre-formatted" mode. The aesthetically pleasing changing of speech marks or apostrophes at the beginning and end of some text will screw up your code. If you're getting weird errors around unexpected symbols or your console queuing up after C&Ping, replace all the speech marks and see if that fixes things. This can also happen with some types of space characters too.

[5] In other programming languages this is often called an **enumerated string**

5.2.1 Converting to strings

Converting to characters and factors is the same as working with numbers. You swap "numeric" for "character" or "factor" and you're done!

You'll see a difference in how these values get displayed. Basic characters are boring - they just print out. Factors look very different. There's no speech marks and there's this Levels bit. The Levels tells you what the unique values in the lookup for this datatype are.

```
as.character(1)
as.character("1 a")

as.factor(1)
as.factor("1 a")
```

```
## [1] "1"
## [1] "1 a"
## [1] 1
## Levels: 1
## [1] 1 a
## Levels: 1 a
```

5.2.2 Checking strings

We can check text in a similar way to checking numbers.

```
is.character(1)
is.character("1")

is.factor(1)
is.factor("1")
is.factor(as.factor("1"))
```

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

Additionally, the `class()` function returns the datatype.

```
class("1")
class(as.factor("1"))
```

```
## [1] "character"
## [1] "factor"
```

5.3 Logical values

Whilst we've been testing our datatypes, we've created a lot of **logical** or **boolean** values. Boolean values are `TRUE` and `FALSE`. R is case-sensitive so these have to be typed upper-case, otherwise it means something different.

You can think of the boolean values as 1 and 0, but using these in your code can result in changing your datatype to a number. If things aren't working as expected make sure to check types as you go along.

5.3.1 Converting to logicals

```
# All these return a TRUE
TRUE
as.logical(1)
as.logical("TRUE")
as.logical("true")

# All these return a FALSE
FALSE
as.logical(0)
as.logical("FALSE")
as.logical("false")
```

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

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

5.3.2 Checking logicals

There is support for checking if something is of datatype logical.

```
is.logical(1)
is.logical(TRUE)
is.logical("TRUE")
class(TRUE)
```

```
## [1] FALSE
## [1] TRUE
## [1] FALSE
## [1] "logical"
```

5.4 Dates

Dates are one of the hardest parts of programming! This is a very brief introduction to dates and they will not be covered further in this book. Expect a later book to dedicate a lot of page space to date handling.

Dates in R split into:

- **date**s do not have any time component
- **POSIX** date-times
- **POSIXct** is an integer based storage method
- **POSIXlt** is a component based storage method

You might be looking at the two POSIX times and thinking to yourself "ZOMG how am I meant to choose?". Most people use the POSIXct format[6], which is the default for many of R's functions.

[6] According to my unscientific twitter poll at `http://twitter.com/`

5.4.1 Converting to dates

You can convert to date-time's with `as.Date()`,`as.POSIXct()`, and `as.POSIXlt()`. Ideally, you'll provide a string with the date(time) in ISO8601 formats e.g. "YYYY-MM-DD hh:ss". If not, you'll want to read up on the date-time format specifications[7] for R.

```
as.Date("2017-12-31")
as.POSIXct("2017-12-31")
as.POSIXlt("2017-12-31")

as.Date("2017-12-31 23:59")
as.POSIXct("2017-12-31 23:59")
as.POSIXlt("2017-12-31 23:59")
```

```
## [1] "2017-12-31"
## [1] "2017-12-31 GMT"
## [1] "2017-12-31 GMT"
## [1] "2017-12-31"
## [1] "2017-12-31 23:59:00 GMT"
## [1] "2017-12-31 23:59:00 GMT"
```

Note that it's assuming a time zone based on my device as I've not provided a default. It's prudent to set the time zone in order to avoid the results of your code changing based on where the code is run or when[8].

```
as.POSIXct("2017-12-31 23:59", tz = "UTC")
```

```
## [1] "2017-12-31 23:59:00 UTC"
```

5.4.2 Checking dates

Unfortunately, R does not provide functions for checking whether the class of something is a date-time type without extending its functionality. We have to use `class()` as a consequence.

`SteffLocke/status/895198115594153988`

[7] `http://stat.ethz.ch/R-manual/R-devel/library/base/html/strptime.html`

[8] Daylight savings time can catch you out

```
class(as.Date("2017-12-31"))
class(as.POSIXct("2017-12-31"))
class(as.POSIXlt("2017-12-31"))
```

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

You'll see that the POSIX values not only returns the class we expected but "POSIXt" as well. POSIXt is an interchange format behind the scenes of dates in R. You don't directly use it and you can ignore it from here on in.

5.4.3 Getting dates and times

R has some functions for getting current date-time values[9].

```
Sys.Date()
Sys.time()
Sys.timezone()
```

```
## [1] "2017-12-13"
## [1] "2017-12-13 12:25:01 GMT"
## [1] "Europe/London"
```

Datetime data types in R can feel a little rudimentary or clunky. I usually use the package `lubridate` for better date handling capabilities.

5.5 Missings

Every data type has an `NA`, an identifier for a missing value.

If you use an NA in an object (more on those in a later chapter) it will take on the data type used in the object. You can, however, make NAs directly.

[9] This is an area showing those wonderful R quirks - the `Sys.*` functions are inconsistently cased

```
NA
NA_integer_
NA_character_
```

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

5.5.1 Checking NAs

You can check what data type an NA is, using the `class()` function.

```
class(NA)
class(NA_integer_)
class(NA_character_)
```

```
## [1] "logical"
## [1] "integer"
## [1] "character"
```

You can check if something is NA with the `is.na()` function.

```
is.na(NA)
is.na(1)
```

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

5.6 Summary

There a few more datatypes out in the wild but numbers, strings, booleans, and dates are the core types you'll encounter.

There are normally `as.*` and `is.*` functions for converting to a datatype or checking if something is a given datatype. You can use `class()` to uncover the datatype too.

Data type	Example
Integer	1
Logical	TRUE
Numeric	1.1
String / character	“Red”
Factor (enumerated string)	“Amber” or 2 in c(“Red”,“Amber”,“Green”)
Complex	i
Date	“2017-12-13”

5.7 R Data Types Exercises

- Convert TRUE to an integer
- What is the datatype of the value returned by `Sys.time()`?
- What is the datatype of the value returned by `Sys.timezone()`?
- Make this quote into an R string
  - “Do you think this is a game?”, he said. “No, I think Jenga’s a game”, Archer responded.

Chapter 6

Basic operations

Now that we have some datatypes, we can start learning what we can do with them.

6.1 Maths [1]

In R, we have our common **operators** that you're probably used to if you've performed calculations on computers before.

Action	Operator	Example
Subtract	-	`5 - 4` = 1
Add	+	`5 + 4` = 9
Multiply	*	`5 * 4` = 20
Divide	/	`5 / 4` = 1.25
Raise to the power	^	`5 ^ 4` = 625

R adheres to **BODMAS**[2] so you can construct safe calculations that combine operators in reliable ways.

[1] I'm British, deal with it.

[2] Brackets, Other, Division, Multiplication, Addition, Subtraction. Note that in some countries it's BEDMAS, where the E stands for Exponents, which is a special Other

```
(1 + 2^3) - 5 * (4/2)
```

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

Additionally, there are some other operators worth knowing about.

Action	Operator	Example
Basic sequence	:	`1:3` = 1, 2, 3
Integer division	%/%	`9 %/% 4` = 2
Modulus	%%	`9 %% 4` = 1

The colon (`:`) is a really snazzy way of generating a sequence of numbers that step by 1. You specify a beginning number and an end number and R will produce all the whole numbers including and between the two numbers. This even works for negative numbers or producing descending values.

```
1:5
5:1
-1:5
5:-1
```

```
## [1] 1 2 3 4 5
## [1] 5 4 3 2 1
## [1] -1  0  1  2  3  4  5
## [1]  5  4  3  2  1  0 -1
```

Integer division (`%/%`) tells you how many times the first number can be divided by the second without returning a fractional value.

```
1:8
1:8%/%3
1:8%/%4
```

```
## [1] 1 2 3 4 5 6 7 8
## [1] 0 0 1 1 1 2 2 2
## [1] 0 0 0 1 1 1 1 2
```

The **modulus** (`%%`) tells you how much is left over after performing an integer division.

```
1:8
1:8%%3
1:8%%4
```

```
## [1] 1 2 3 4 5 6 7 8
## [1] 1 2 0 1 2 0 1 2
## [1] 1 2 3 0 1 2 3 0
```

For reasons[3] not worth worrying about, R uses the `%` sign as the start of special operators – usually these are custom built, contain text, or reserved symbols.

6.2 Comparison

The next important thing to know about is how to write comparisons; ways of looking at two or more things and finding out if they're the same, or different.

6.2.1 Common operators

The less thans and greater thans are symbols that are in pretty much every language for comparisons, but the test to see if two values are the same or not can often vary across languages.

```
2 < 3
3 > 2
2 >= 2
2 <= 2
```

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

In R, you test if two values are exactly the same with `==` and you test if they're different with `!=`.

[3] Read Rbitrary `http://ironholds.org/projects/rbitrary/` for more info

```
2 == 2
2 != 2
```

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

You can test if a value is present in a list of acceptable values using the `%in%` operator. This may seem a little trivial right now, but once we start covering more than one value at a time, and working with strings, it'll really start to shine!

```
2 %in% 1:3
```

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

6.2.2 A gotcha

Testing for equality can get a little weird with R because it uses a different way of storing numbers than we would expect. It doesn't store numbers quite as precisely as we expect - somewhere at the very end of a large number of decimal places, the value can be rounded incorrectly. It doesn't make a difference to most of our calculations but it will often hit when you're comparing two decimal values.

Let's see an example.

Both these calculation return what we think of as `0.2`

```
0.5 - 0.3
0.6 - 0.4
```

```
## [1] 0.2
## [1] 0.2
```

Indeed, if we test `0.2` is the same as `0.2` we get a `TRUE` which matches our expectations.

```
0.2 == 0.2
```

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

But, when we perform two calculations, even though they come out to the same value to us, there's a little bit of imprecision in how they're stored that stops them from being *exactly the same* number.

```
(0.6 - 0.4) == (0.5 - 0.3)
```

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

To avoid this issue, if you're comparing decimal values that result from calculations it is better to use the `all.equal()` function. `all.equal()` adds a tolerance to the comparison which means the very subtle imprecision is ignored. The default tolerance is 1.5×10^{-8}, in other words the imprecision is *very, very small*.

```
all.equal(0.6 - 0.4, 0.5 - 0.3)
```

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

6.2.3 Summary

Action	Operator	Example
Less than (lt)	<	`5 < 5` = FALSE
lt or equal to	<=	`5 <= 5` = TRUE
Greater than (gt)	>	`5 > 5` = FALSE
gt or equal to	>=	`5 >= 5` = TRUE
Exactly equal	==	`(0.5 - 0.3) == (0.3 - 0.1)` is FALSE
Exactly equal	==	2 == 2 is TRUE
Not equal	!=	`(0.5 - 0.3) != (0.3 - 0.1)` is TRUE
Not equal	!=	2 != 2 is FALSE
Equal	all.equal()	`all.equal(0.5-0.3,0.3-0.1)` is TRUE
In	%in%	`"Red" %in% c("Blue","Red")` is TRUE

6.3 Logic

Once we can do a single check, we inevitably want to do multiple checks at the same time.

To combine multiple checks, we can use *logical operators*.

6.3.1 Common operators

The ampersand (`&`) allows us to combine two checks to do an AND check, which is "are both things true?".

```
TRUE & TRUE
TRUE & FALSE
FALSE & FALSE
(2 < 3) & (4 == 4)
(2 < 3) & (4 != 4)
```

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

The pipe, or bar (`|`)[4] allows us to do an OR check, which is "are either of these things true?".

```
TRUE | TRUE
TRUE | FALSE
FALSE | FALSE
(2 < 3) | (4 == 4)
(2 < 3) | (4 != 4)
```

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

The exclamation point (`!`) allows us to a perform a NOT check, by negating or swapping a check's result. This allows you say things like "is this check true and that check not true?".

[4]Getting this symbol can be painful as it varies substantially by keyboard, so apologies if it takes you a while to hunt this symbol down.

```
TRUE & TRUE
TRUE & !FALSE
!FALSE & !FALSE
(2 < 3) & (4 == 4)
(2 < 3) & !(4 != 4)
```

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

6.3.2 Other operators

Less commonly, there other logical checks you might to perform.

We can do an XOR, where one and only one of two values being checked is true.

```
xor(TRUE, FALSE)
xor(TRUE, TRUE)
xor(FALSE, FALSE)
```

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

6.3.3 Summary

We can produce sophisticated checks from a few simple building blocks. This will come in very handy down the line when doing things like filtering datasets or creating new fields in your data.

Action	Operator	Example
Not	!	`!TRUE` is FALSE
And	&	`TRUE & FALSE` is FALSE
And	&	`c(TRUE,TRUE) & c(FALSE,TRUE)` is FALSE, TRUE
Or	|	`TRUE | FALSE` is TRUE
Xor	xor()	`xor(TRUE,FALSE)` is TRUE

6.4 Summary

This basic operations section has hopefully taught you how to manipulate values and construct comparisons. These are important building blocks in data analysis, and whilst we've been working with only a single value at a time, in the next section we'll see how it works with more data.

Action	Operator	Example
Subtract	-	`5 - 4` = 1
Add	+	`5 + 4` = 9
Multiply	*	`5 * 4` = 20
Divide	/	`5 / 4` = 1.25
Exponent	^	`5 ^ 4` = 625
Less than (lt)	<	`5 < 5` = FALSE
lt or equal to	<=	`5 <= 5` = TRUE
Greater than (gt)	>	`5 > 5` = FALSE
gt or equal to	>=	`5 >= 5` = TRUE
Exactly equal	==	`(0.5 - 0.3) == (0.3 - 0.1)` is FALSE
Exactly equal	==	2 == 2 is TRUE
Not equal	!=	`(0.5 - 0.3) != (0.3 - 0.1)` is TRUE
Not equal	!=	2 != 2 is FALSE
Equal	all.equal()	`all.equal(0.5-0.3,0.3-0.1)` is TRUE
In	%in%	`"Red" %in% c("Blue","Red")` is TRUE
Not	!	`!TRUE` is FALSE
And	&	`TRUE & FALSE` is FALSE
And	&	`c(1,1) & c(0,1)` is FALSE, TRUE
Or	|	`TRUE | FALSE` is TRUE,
Xor	xor()	`xor(TRUE,FALSE)` is TRUE

6.5 Basic Operations Exercises

1. What is the result of `pi^2`?
2. Is pi greater than 3?
3. Construct a statement to check if 5 is both greater than 3 and less than or equal to 6
4. What are the results if you check to see if a sequence of 1 to 5 is less than or equal to 3?

Chapter 7

R objects

So far we've just worked with some single values to get to grips with how some of the various operations work. Of course, we rarely work with a single value! If we did, we could just use a calculator.

This section helps you get to grips with some different ways of storing data and how to manipulate your datasets in the "traditional" way. This will help you understand a lot of code written in the past, and will equip you to understand the material in the next book, which focuses on data manipulation of tabular data.

7.1 Storing values

When we were performing operations, we got some values output to the console. One of the key principles in writing code is Don't Repeat Yourself (DRY) so we need to know how we can avoid repeating ourselves in R. One of the ways you can do that is to store a value for use later.

In R, we can store values by **assigning** them a name. This makes a **variable** or **object**. We can do this with a few different operators, but the traditional operator is a `<-`[1]. The format for assigning a value is `nameofthing <- value`.

[1] Other valid values are the equals symbol `=` and you can also do a right-handed assignment with `->`

```
my_variable <- 5 + 3
my_variable > 6 & my_variable < 10
```

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

By default, when you store values R doesn't return the results to the console. You can change that behaviour by putting brackets around the assignment like `(my_variable <- 5 + 3)`.

Valid names for a variable include upper-case letters, lower case letters, numbers anywhere but the beginning, periods (`.`), and hyphens (`_`).

There are a number of different competing conventions for how you name variables. The most common conventions are shown below. I have no strong feelings for any system and only ask that you pick one and stick with it within a single script. Whatever you do, don't forget names are case sensitive!

```
myfirstvariable <- 1
myFirstVariable <- 1
MyFirstVariable <- 1
my_first_variable <- 1
my.first.variable <- 1
```

You can create names breaking the rules governing valid names by placing the rule breaking name between two back-ticks (‘). I don't recommend you do this with variables you'll create, but you'll often end up with names that break conventions when importing data, especially when you import from spreadsheets.

```
`name with spaces` <- 1
`2017` <- 1
` ` <- 1
`$$$` <- 1
```

Variables you store get stored in-memory. This means they'll hang around whilst R is open and will be gone after that. You can see variables you've created in the Environment tab.

RStudio will by default save your variables for you so that next time you open it up, your variables are stored.

RStudio saving your variables is a blessing because you don't have to worry about keeping RStudio open the all time.

It's also a pretty major curse because you'll inevitably create something at some point through the console or an Untitled R file and then lose that bit of code. Now when your script runs in a fresh session it'll fail. You'll risk tearing your hair out and worse as you go through the pain of debugging this.

I recommend you get in the habit early of not working with your session being saved so that you don't miss vital lines of code out from your scripts. Turn it off in *Tools > Global Options*, untick "Restore .RData into workspace at startup"

If you need to manage what's been stored you can list objects with `ls()` and remove them with `rm()`.

```
today <- Sys.Date()
rm("today")
```

Another way you can remove a single object is to overwrite the object with value NULL.

```
today <- Sys.Date()
today <- NULL
```

If you want to remove everything you can use`ls()` inside `rm()` but you have to tell the function you're providing a list of variable names.

```
ls()
rm(list = ls())
```

You can also achieve the same results by using the broom symbol in the Environment tab in RStudio.

7.2 Vectors

A **vector** is a collection of values that hold the same datatype. It is **one-dimensional** in that none of the **elements** in the collection correspond to other values like they might in a table of values.

A single value is actually a vector of **length** 1.

When I introduced the colon (`:`) as a means of generating a sequence, we were in fact generating a vector where each element was a number in the sequence. The vector has a length which is as long as the number of values generated by the sequence.

```
-1:1
```

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

Another way of producing a vector is to use the combine function (`c()`). This is great for combining a number of disparate character strings into a vector.

```
c("red", "yellow", "blue")
```

```
## [1] "red"    "yellow" "blue"
```

A single value is a still a vector. What we see when we use the `c()` function is that we're combining vectors. As a result we can also use it on longer vectors too.

```
c(1:3, 2:1, 5:8)
```

```
## [1] 1 2 3 2 1 5 6 7 8
```

When we combine values into a single vector, R will change everything to the same datatype using some conversions.

```
c(1, FALSE)
c(1, "FALSE")
```

```
## [1] 1 0
## [1] "1"     "FALSE"
```

This **implicit conversion** is something to be careful of and can really screw up your results! If in doubt, check the datatype with the `class()` function, or look for the vector in your global environment and see what the datatype is from there.

We can also give names to values being included in a vector.

```
c(first = "Steph", last = "Locke")
```

```
##   first    last
## "Steph" "Locke"
```

7.3 Getting information about vectors

Our `class()` function will still work with a vector with a length greater than 1 to get you it's datatype.

Let's look at a sequence of numbers and one of the built-in vectors that contains the alphabet.

```
class(1:10)
LETTERS
class(LETTERS)
```

```
## [1] "integer"
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"
"N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
## [1] "character"
```

We can use the `length()` function to find out the number of elements in a vector.

```
length(pi)
length(LETTERS)
```

```
## [1] 1
## [1] 26
```

To extract the names of values in a vector, we can use the `names()` function.

```
steph <- c(Steph = "forename", Locke = "surname")
names(steph)
```

```
## [1] "Steph" "Locke"
```

7.4 Calculations on multiple vectors

When we perform calculations on two vectors, R will try to perform the operation for each set of elements. This is an **element-wise** or **pair-wise** calculation methodology.

In SQL, it's equivalent to where you might write `colA*colB` and you'll get the answer calculated for every row in the table. In Excel, it's equivalent to a Fill Down of multiplying two values on the same row.

Let's looks at how this works in practice in R.

We have two vectors, each containing two elements.

```
vecA <- 1:2
vecB <- 2:3
```

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

If we want to multiply the two vectors by each other, R will match each element in the first vector with its counterpart in the second and multiply the two values together to make a new element.

```
vecA * vecB
```

```
## [1] 2 6
```

We can also do this with vectors of different lengths to a certain extent. The most common scenario is operating on a vector by doing something with a single value.

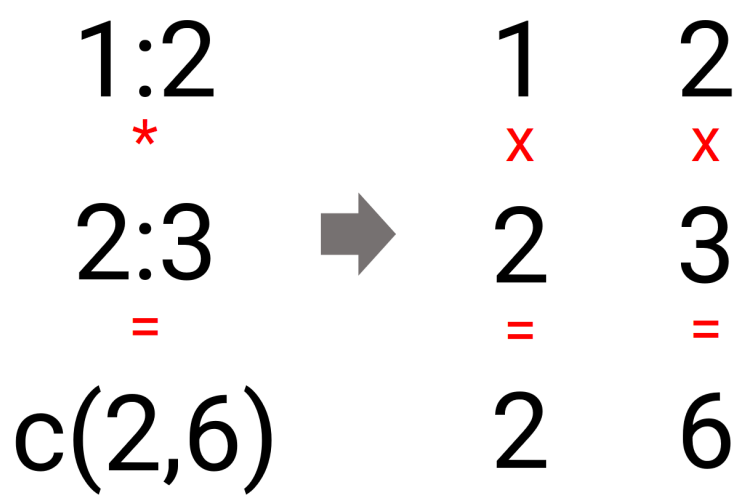


Figure 7.1

A single value is a vector of length 1. When R gets a request to do something with a vector of length X and a vector of length 1, it will basically repeat the vector of length 1 X times to make two vectors the same length. It will then perform the calculation element-wise.

```
vecA * 3
```

```
## [1] 3 6
```

You can also use this functionality of making a vector the same length as another, known as **recycling**, work for other mis-matched vector sizes. The only rule is that one of the vector lengths must divide cleanly by the other.

- Two vectors of the same length divide by the other's length exactly one time and won't need to recycle
- A vector of length one always cleanly divides any other vector's length and so will be recycled
- A vector of length 2, will divide any vector with an even length and so will be recycled in those cases, but it cannot recycle cleanly for odd length vectors

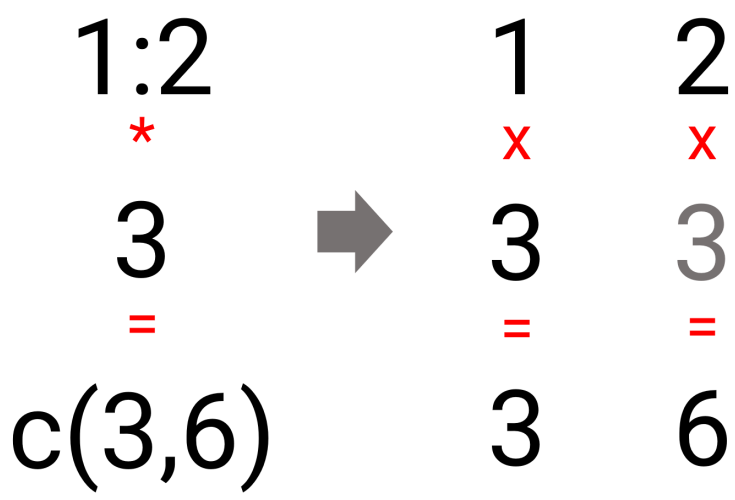


Figure 7.2

```
1:10 * 2
1:10 * 2:3
1:10 * 2:4
```

```
## [1] 2 4 6 8 10 12 14 16 18 20
## [1] 2 6 6 12 10 18 14 24 18 30
## [1] "longer object length is not a multiple of shorter
object length"
```

Vector recycling is useful and dangerous – it can help you make elegant code or give you unexpected results. Especially when starting out, I recommend you make your vectors either the same length or length 1.

7.4.1 Bitwise

Our logical operators that we covered earlier, work in a pairwise fashion. They'll return a vector of the same length as the longest one used in your logical statement.

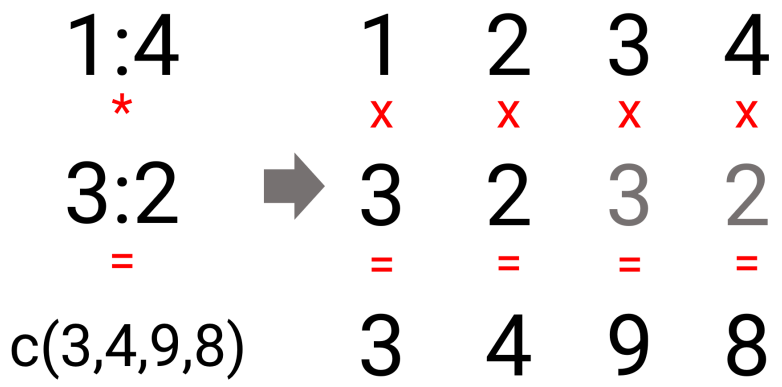


Figure 7.3

```
a <- 1:2 > 1
b <- 2:3 > 1
```

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

Making logical statements returns vectors with a logical datatype.

```
a & b
a | b
```

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

Occasionally, you expect to only be operating on a single pair of values and want to enforce that R should only do the calculation on the first pair. In R, this called a **bitwise** AND (`&&`) or OR (`||`).

A bitwise logical statement will only do the check for the first elements in the vectors and ignore all the others.

```
a && b
a || b
```

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

Use bitwise operators with extreme care!

7.5 data.frames

A **data.frame** is a table similar to what we're used to working with in most data analysis tools. It will contain a number of rows with columns containing different pieces of information. Each column in a data.frame has a datatype but it does not have to be the same datatype as the other columns.

We can construct a data.frame from individual vectors via the `data.frame()` function.

```
data.frame(a = 1:2, b = c("blue", "red"))
```

```
##   a    b
## 1 1 blue
## 2 2  red
```

You can also give row names to the rows you end up making, however, I recommend you add these in as a column instead as it'll make them easier to work with long-term.

```
baddf <- data.frame(a = 1:2,
                    b = c("blue","red"),
                    row.names = c("First","Second"))

gooddf <- data.frame(a = 1:2,
                    b = c("blue","red"),
                    ID = c("First","Second"))
```

Throughout many of the examples, I'll use the example datasets that are available by default in R.

```
View(iris)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

The `View()` function is specific to RStudio and provides a nice visual grid view of a data.frame and it allows you to search and sort the table for some initial exploration.

More commonly, we'll import data from an outside source.

7.6 Importing data.frames

You can import data via code, but one of the easiest ways of getting started is to load data via RStudio and have it generate the code for you.

To import data...

1. Go to the Environment tab and select Import Dataset
2. Select the relevant type of data you want to import
3. Browse to the file you want to upload.

Keeping data in the project directory is ideal as it keeps everything in one place and makes imported code easier to read.

You can tweak the advanced settings and then select the *Import* button to load the data directly into memory. Alternatively, you can copy the code it generated for you and paste it into a script. By doing this copy and pasting, you will make the import reproducible. Next time you need to load the data you can just run the code, instead of using the interface again.

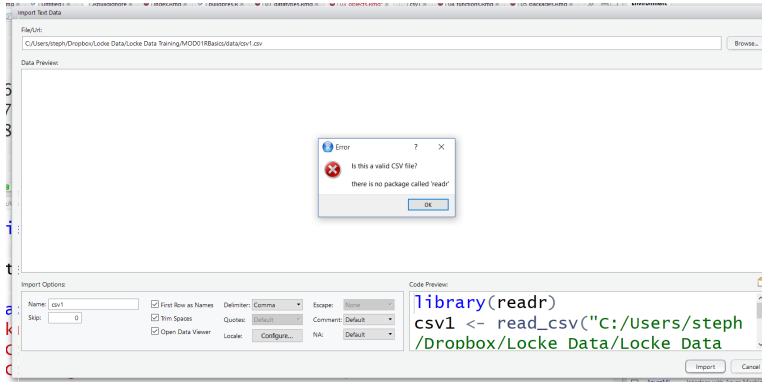


Figure 7.4

7.6.1 Error!!

If you were tying to do this import you may have gotten an error when you tried to load a file because you don't have some of the required functionality that RStudio expects you to have.

It will tell you the name of the thing you're missing. In my case - I'm missing the package "readr". To make this available to us, we go to the Packages tab and then:

1. select *Install*
2. type "readr"
3. select the *Install* button
4. accept any popups for things like restarting R

7.7 Getting information about data.frames

Our data.frames are **composites**, they are the result of combining a number of vectors with different data types. As a consequence, when we run our `class()` function, it tells us an object is a `data.frame` and no longer returns the underlying datatype.

```
class(iris)
```

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

You do not get the number of rows in a data.frame when you run the `length()` function, instead you get the number of columns[2]. Alternatively, you can run the more clearly named `ncol()` function to return the number of columns in a data.frame.

```
length(iris)
ncol(iris)
```

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

You can get the number of rows via the `nrow()` function.

```
nrow(iris)
```

```
## [1] 150
```

Similarly to `length()`, the `names()` function when applied to data.frame's only works on the columns, so you can use it to get column names. A clearer alternative is to use the `colnames()` function. You can use `rownames()` to get names for rows, if they exist.

```
mydf <- data.frame(a = 1:2,
                   b = c("blue","red"),
                   row.names = c("First","Second"))

names(mydf)
colnames(mydf)
rownames(mydf)
```

```
## [1] "a" "b"
## [1] "a" "b"
## [1] "First"  "Second"
```

[2]This is because a data.frame is actually just a prettily printed list, and each column in an element in said list, and length returns the number of elements overall.

7.8 Lists

Lists are a catch-all object. They literally hold any and all types of the objects covered in this section, including list objects!

You can create lists with the `list()` function, and like with our other objects you can have named and unnamed elements.

In this example, we create a list object holding two vectors.

```
mylist <- list(a = 1:3, LETTERS)
mylist
```

```
## $a
## [1] 1 2 3
##
## [[2]]
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"
"N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

At least initially, most people tend to work with their data in a data.frame and may only interact with a list as a consequence of doing something like building a linear regression model. Lists are very common outputs to statistical functions because you need things like a formula, fitted results, coefficients, model metrics, and more. If you do build a model though, there's a bunch of helper functions for extracting different components so you don't even have to think about the fact you're working with a list.

7.8.1 Getting information about lists

The `length()` function will tell you how many elements there are in a list.

```
length(mylist)
```

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

`names()` lets you get the element names and returns a blank (`""`) where no name was provided.

```
names(mylist)
```

```
## [1] "a" ""
```

7.9 Other object types

There a number of other object types in R. They won't be covered in detail in this book, because they tend to be used by a small fraction of R users.

- A **matrix** is a two-dimensional object that can only contain one datatype
- An **array** is a multi-dimensional object that also can contain only one datatype
- A **table** object is similar to a matrix but is created by producing a contingency table

In R, developers can also create other object types specific to their requirements. People use this to create geospatial objects and more. I don't recommend you think about creating your own custom objects, especially at this point in your R writing career. If/when you want to write your own custom classes then my preferred package for that is `R6`.

7.10 Useful functions

Whatever the object type, there are some functions that come in handy for exploring it and getting some useful metadata.

You get the contents of any object by writing its name.

```
mylist
```

```
## $a
## [1] 1 2 3
##
## [[2]]
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"
"N" "O" "P" "Q"
```

```
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

However, if you're working with a lot of data, you probably don't want to fill up your console that way. R has two functions, `head()` and `tail()`, which allow you to see values from the beginning or end of an object. For objects containing many elements, such as lists or data.tables, `head()` and `tail()` returns the first or last 5 values respectively.

```
head(LETTERS)
tail(LETTERS)
```

```
## [1] "A" "B" "C" "D" "E" "F"
## [1] "U" "V" "W" "X" "Y" "Z"
```

If you want to examine an R object, you can use the `str()` function to get the structure of the object.

```
str(mylist)
```

```
## List of 2
##  $ a: int [1:3] 1 2 3
##  $  : chr [1:26] "A" "B" "C" "D" ...
```

7.11 Summary

You can perform calculations on the fly or store results for later use. You can assign values with the `<-` operator.

Functions like `class()`, `length()`, and `head()` work well to extract information about R objects.

R performs calculations over vectors so that you only have to provide two or more vector names and the operation you want performed. R will then perform this operation pair-wise for the vectors.

You can import datasets in R by using the "Import Dataset" function. This will also give you the code to use so that you can write code that another person will be able to use. This is great because it makes your work reproducible and automatable!

As well as vectors and data.frames, there are list object types and some other object types. These are less commonly used, although lists are quite common when getting outputs from statistical functions.

7.12 R objects exercises

1. See what's in the built-in variable `letters`
2. Write a check to see if "A" is present in `letters`
3. Find out which values in the sequence 1 to 10 are greater than or equal to 3 and less than 7
4. Make a vector containing the numbers 1 to 50
5. Make a vector containing two words
6. What happens when you combine these two vectors?
7. Make a data.frame using the two vectors
8. What happened to your text vector?
9. Make a list containing some of the variables you've created so far
10. Retrieve the head or tail of the `iris` dataset

Part III

Basic data manipulation

Chapter 8

Grid references

With R objects, it's possible to use a grid reference system to select values from an object.

In vectors and lists, you can specify the element position as they only have a single dimension. In data.frames, you can pinpoint the element via the row and the column.

You can provide a grid reference by adding square brackets after a name e.g. `mylist[ ]`. Inside the square brackets, we can provide values in a few different ways to say which part of the object's "grid" is required.

If you want everything in an object, you can just use the object's name or put empty square brackets after it i.e. `LETTERS` and `LETTERS[]` are identical.

```
LETTERS[]
```

```
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"
"N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

8.1 Grid references with numbers

To select a specific element, you provide the number indicating its position in the object.[1]

This is similar to Excel. When you only have a single column of values in a spreadsheet, you can identify a value to someone by telling them what row number it's on. When you have a table, you need to tell someone both the row and the column for someone to find the exact value.

8.1.1 Single value selection with vectors

To select a single element from a vector, we need to put the element's position inside square brackets after the vector.

To select the second element in the vector `LETTERS` we put its position (2) into the grid.

```
LETTERS[2]
```

```
## [1] "B"
```

We're not bound to selecting values from objects that are stored either! For instance, we can generate a sequence of numbers and subset from it directly.

```
(10:25)[13]
```

```
## [1] 22
```

8.1.2 Single value selection with data.frames

In a data.frame, you can provide one or two values. These are comma separated inside the square brackets and row numbers get specified first e.g. `iris[ row , column ]`. If you want to select all rows or all columns you leave that part of the reference blank

[1] The first element in an R object is at position 1. This is contrary to a number of programming languages where the first element is at position 0.

e.g. `iris[1, ]` to return the first row and `iris[ ,2]` to return the second column.

```
mydf <- data.frame(a = 1:5, b = 6:10, c = 11:15)
```

a	b	c
1	6	11
2	7	12
3	8	13
4	9	14
5	10	15

Whenever I'm going to output a data.frame, I'm going to output in a formatted way as opposed to how it'll appear in the console. This is to make it easier to see the changes as the console view can get a little much! You can use the `View()` function to see the data.frame in a nice browser in RStudio.

If we provide a row number by using `df[ X , ]`, we will get a data.frame object back with just one row.

```
mydf[1, ]
```

a	b	c
1	6	11

If we provide a column number by using `df[ , Y ]`, we will get a vector back.[2]

```
mydf[, 1]
```

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

If we specify a row and a column by using `df[ X , Y ]`, we get a vector back containing a single element although in we'd normally refer to it as a single value for brevity.

[2]This is not quite accurate but it's a good starting point. In fact, a data.frame is actually a list with pretty print methods so you could theoretically have a column that is a list.

```
mydf[3, 3]
```

```
## [1] 13
```

8.1.3 Single value selection with lists

When we use the grid reference system to select stuff from lists, R returns a list with just the element you selected in it.

Our example list contains two vectors. Both vectors are stored as elements but the sequence one to three was additionally given a name.

```
mylist <- list(a = 1:3, LETTERS)
mylist
```

```
## $a
## [1] 1 2 3
##
## [[2]]
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"
"N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

We can select elements based on their position, irrespective of whether they have names.

```
mylist[2]
```

```
## [[1]]
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"
"N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

If an element was named, then that name will be kept and displayed.

```
mylist[1]
```

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

8.1.4 Multiple values

Remember how a single value is still counted as a vector by R? This means that when we say `letters[1]` the 1 is actually a vector, and *that* means that we can provide longer vectors in our grid specifications too!

For a vector, that means we can provide a single vector with the positions of the elements to return.

```
LETTERS[1:5]
```

```
## [1] "A" "B" "C" "D" "E"
```

The ranges don't have to be continuous either.

```
LETTERS[c(1:5, 23:26)]
```

```
## [1] "A" "B" "C" "D" "E" "W" "X" "Y" "Z"
```

In fact, you can repeat numbers to get the same value out multiple times.

```
LETTERS[c(1, 1, 1)]
```

```
## [1] "A" "A" "A"
```

These things all hold true for data.frames too. This means we can provide ranges to both rows and columns to subset by the position of values in the table.

```
mydf[1:3, 1:2]
```

a	b
1	6
2	7
3	8

8.1.5 Negative values

As well as positive specifications, we can also use negative values. These tell R which bits of the grid that you *don't* want.

Here I exclude the first five letters.

```
LETTERS[-(1:5)]
```

```
## [1] "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R"
"S" "T" "U" "V"
## [18] "W" "X" "Y" "Z"
```

When it comes to data.frames we can provide negative values in both rows and columns to produce a subset we're interested in.

```
mydf[-3, -2]
```

	a	c
1	1	11
2	2	12
4	4	14
5	5	15

8.1.6 Missing values

You might be wondering what happens if you refer to a row number or element position that is not between 1 and the length of your object.

In such a scenario, R will actually return an **NA** (a missing value) for that position.

```
LETTERS[23:29]
```

```
## [1] "W" "X" "Y" "Z" NA  NA  NA
```

```
mydf[5:6, ]
```

	a	b	c
5	5	10	15
NA	NA	NA	NA

8.2 Grid references with names

Where names are used, we can provide these names in our grid references.

```
mylist["a"]
```

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

This works for column (and row) names.

```
mydf[, "a"]
```

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

We can provide longer vectors containing column names too. Recall when we used numbers, one value in the column returned a vector, but multiple values resulted in a data.frame? The same is true here.

```
mydf[, c("a", "b")]
```

a	b
1	6
2	7
3	8
4	9
5	10

8.3 Grid references with conditional values

Whilst we often want to subset data.frames to some specific columns, a lot of the time with vectors and data.frames we want to be able to apply a condition that determines which values are returned. We want to **filter** rows.

Like with SQL, you can apply a filter by telling R which rows (or elements) it should and shouldn't return. You do this be providing a set of boolean values where TRUE means the row should be returned and FALSE says it should be excluded.

We can provide hard-coded boolean values to the row and column parts of our grid reference system.

For instance, if I wanted to exclude the second column in the data.frame I could say to include the first and third by giving them a TRUE in my filter and I could exclude the second column by giving it a FALSE in my filter.

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

a	c
1	11
2	12
3	13
4	14
5	15

You might be used to using 0 and 1 as shorthand for boolean values. Unfortunately, if you try to use this you find NAs returned for any values you intended to be excluded by using 0, and you'll get the first value repeated everywhere you used a 1 to indicate inclusion.

8.3.1 Building conditional vectors

Hard-coding TRUE and FALSE values is probably not your idea of fun and certainly isn't mine. We can use our knowledge of building

comparisons to generate our booleans for inclusion.

Let's say we wanted all the letters of the alphabet up to and including "e". We could use our comparison operators to compare every letter against "e" and return a TRUE where it is "e" or occurs before "e" in the alphabet, and it would return a FALSE when it occurs after "e".

This gives us an include and exclude instruction for each of the 26 letters. We can then use this boolean vector as our filter in the grid reference system.

```
earlyletters <- LETTERS <= "E"
LETTERS[earlyletters]
```

```
## [1] "A" "B" "C" "D" "E"
```

This can be simplified by doing the comparison directly within the grid reference.

```
LETTERS[LETTERS <= "e"]
```

```
## [1] "A" "B" "C" "D"
```

You're not limited to single comparisons either. You can use AND (`&`) and OR (`|`) to produce compound statements.

If we wanted every letter between (and including) "B" and "E" we can check to see which elements of `LETTERS` are "B" or are after "B" and combine this with our existing "E" check using an `&`.

```
LETTERS[LETTERS <= "E" & LETTERS > "B"]
```

```
## [1] "C" "D" "E"
```

8.3.2 Conditional filters for data.frames

If we wanted to select all columns in our data.frame that had names beginning with "a" or "b", we could compare the names to the letter "c" and use this set of boolean values to be our filter.

To extract the column names, we can use `colnames()`. This returns a vector of character values and we can run a comparison.

```
abcols <- colnames(mydf) < "c"
```

Now we can use that in our grid reference system.

```
mydf[, abcols]
```

a	b
1	6
2	7
3	8
4	9
5	10

Using the grid reference system, if we wanted to apply a filter to our rows based on some column's data we would first need to extract the column's values, then produce our filter, then apply our filter.

Don't worry if this sounds long-winded and crazy to you. You're thinking that because it's true! A little bit later in this section we'll cut out some of the craziness.

For instance, if we wanted everything from our table where our rows had a value for column "a" less than four, we would need to get column "a"s values, compare it to 4, and use this in our row area of the grid reference.

```
lt4 <- mydf[, "a"] < 4
mydf[lt4, ]
```

a	b	c
1	6	11
2	7	12
3	8	13

Or we could have written it all in one go.

```
mydf[mydf[, "a"] < 4, ]
```

8.3.3 Recycling values

When R has two mismatched vectors in terms of length, it will try to recycle values. We saw this earlier when we worked with vectors.

You can use this to provide shorter vectors of value (although I don't recommend you do so often).

An elegant demonstration of this is returning every other letter in the alphabet.

We need a filter that puts TRUE against the odd number positions and a FALSE against the even number positions. We could write a comparison that checks the position number is odd but that would be quite long winded.

Instead, we can rely on recycling to take a pair of values and repeat them. We can provide a vector containing TRUE and FALSE it will recycle them so that every odd numbered position gets a TRUE and every even numbered position gets a FALSE.

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

```
##  [1] "A" "C" "E" "G" "I" "K" "M" "O" "Q" "S" "U" "W" "Y"
```

8.4 Mixed grid references

You cannot provide a mix of element positions, element names, and booleans in a single vector to get a subset. This is because you have to provide a vector and a vector containing a mix of datatypes will convert everything to a single datatype.

We can verify with our list. We've seen how referring to position 1 works, and referring to the element called "a", so if we wanted to specify both of these we could put them in a vector. The conversion to strings happens though and then R searches the list for an element called "1", can't find it, and returns an NA.

```
c(1, "a")
mylist[c(1, "a")]
```

```
## [1] "1" "a"
## $<NA>
## NULL
##
## $a
## [1] 1 2 3
```

Whilst you can't combine the methods in a single section of the grid reference system, you can use different systems in different positions. This is most useful for data.frames when we want to subset our rows by a condition, and only return certain columns at the same time.

```
mydf[1:2, c("a", "b")]
```

a	b
1	6
2	7

8.5 Other reference methods

If you need to select a given named value or column from an object, there are some alternative selection methods you'll use.

There are double square brackets for when you expect one, and only one, named element. This is mainly used for lists.

```
mylist[["a"]]
```

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

There is a much nicer option though for lists and data.frames. That option is using the dollar sign ($) to access named elements in lists or columns in data.frames.

```
mylist$a
```

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

```
mydf$b
```

```
## [1]  6  7  8  9 10
```

The $ methodology has some benefits: It uses fewer characters and you can use code-completion with it.

We can use both these notations inside our grid reference system. This becomes very handy for writing row conditions for data.frames.

Taking our earlier example of subsetting rows where column "a"'s values are less than 4 becomes much simpler.

```
mydf[mydf$a < 4, ]
```

```
##   a b  c
## 1 1 6 11
## 2 2 7 12
## 3 3 8 13
```

This is the old-school way of working with data.frames. It's important to be able to write queries of your data this way, or at least read other people's code but as soon as you can you should move onto the `data.table` or `tidyverse` ways of working with data.frames. The next book in this series will focus on the `tidyverse` way of working with data.frames.

Chapter 9

Changing objects

By utilising our reference systems, not only can we select data of interest to us, but we can add new data, update existing values, and even delete values.

You can update part or all of simple objects by assigning new values against a grid-reference.

Adding additional values in a vector involves specifying new element positions using the grid system and assigning a value to that part of the object.

```
letters[27] <- "|"
tail(letters)
```

```
## [1] "v" "w" "x" "y" "z" "|"
```

Similarly, we can specify a row in a data.frame and provide all the necessary values to make a complete row.

```
mydf[6, ] <- c(pi, Inf, -Inf)
```

a	b	c
1.000000	6	11
2.000000	7	12
3.000000	8	13
4.000000	9	14
5.000000	10	15
3.141593	Inf	-Inf

For data.frames, if you want to create a new column, it's usually much easier to use our `$` notation. You specify the column and assign it new values.[1]

```
mydf$d <- 5
```

a	b	c	d
1.000000	6	11	5
2.000000	7	12	5
3.000000	8	13	5
4.000000	9	14	5
5.000000	10	15	5
3.141593	Inf	-Inf	5

Updating values involves providing a set of values of the same size as the destination.

Here I overwrite the first three elements in our lower case alphabet vector with the first three elements in our upper case alphabet vector.

```
letters[1:3] <- LETTERS[1:3]
head(letters)
```

```
## [1] "A" "B" "C" "d" "e" "f"
```

I can update rows by specifying the row and providing a complete set of new values.

```
mydf[1, ] <- 1:4
```

[1] This works because data.frames are actually lists so you're creating a new element containing these values.

	a	b	c	d
	1.000000	2	3	4
	2.000000	7	12	5
	3.000000	8	13	5
	4.000000	9	14	5
	5.000000	10	15	5
	3.141593	Inf	-Inf	5

If you provide something that is not the same size, R will apply the recycling rules. Again, this is nifty and terrible at the same time.

Even though there are currently four columns in our table, we're only providing two values here. Those two values will be recycled across the columns.

```
mydf[2, ] <- 1:2
```

	a	b	c	d
	1.000000	2	3	4
	1.000000	2	1	2
	3.000000	8	13	5
	4.000000	9	14	5
	5.000000	10	15	5
	3.141593	Inf	-Inf	5

If you want to delete values, you can overwrite an object after doing a negative selection. Here I remove the first row of the data.frame.

```
mydf <- mydf[-1, ]
```

		a	b	c	d
2		1.000000	2	1	2
3		3.000000	8	13	5
4		4.000000	9	14	5
5		5.000000	10	15	5
6		3.141593	Inf	-Inf	5

An alternative method is to specify a subset and assign the the value NULL. NULL removes contents in lists and data.frames.

In a list, I can specify one or more elements and assign NULL to

it, in order to remove the specific elements.

```
mylist[2] <- NULL
mylist
```

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

I can remove a column in a data.frame by assigning NULL to it.

```
mydf$c <- NULL
```

	a	b	d
2	1.000000	2	2
3	3.000000	8	5
4	4.000000	9	5
5	5.000000	10	5
6	3.141593	Inf	5

Rows usually get deleted by selecting everything but the the rows you want to discard and overwriting the data.frame variable.

```
mydf <- mydf[-1, ]
```

	a	b	d
3	3.000000	8	5
4	4.000000	9	5
5	5.000000	10	5
6	3.141593	Inf	5

Chapter 10

Summary

In R, you can subset objects using positive, negative, and boolean values. You're able to apply the same methodology to vectors, lists, and data.frames.

When working with data.frames or lists you can use the dollar (`$`) notation to refer to values in a succinct way. You can use this within data.frame subsets to build filters for rows based off the values in columns.

Inserting, updating, or deleting values usually involves specifying a subset and assigning values to it. When deleting, you often assign a value of NULL. You can also use NULL to remove variables in a similar fashion.

Chapter 11

Data manipulation exercises

1. Select all `LETTERS` before "X"
2. Select the first 5 rows from the built-in data.frame `iris`
3. Select the first 2 columns from `iris`
4. Select the column Sepal.Length from `iris` by name
5. Select rows from the `iris` data.frame where the `Sepal.Length` is greater than 5.8cm
6. Select rows from the `iris` data.frame where the Sepal.Width is below the average for that column
7. Select everything from `iris` except the Species column
8. Create a copy of the `iris` data that just contains the first 100 rows and call it `myIris`
9. Update the species column to the value "Unknown" in `myIris`
10. Delete rows from `myIris` where the sepal length is greater than 5.5

Part IV

R functionality

Chapter 12

R functions

In previous sections we've seen R **functions** that are used on objects to perform some activity. Functions seen so far include:

- `class()` and `is.*()` functions for checking datatypes
- `as.*` for converting to datatypes
- `length()` and `names()` for metadata
- `head()` and `tail()` for getting a small amount of elements from an object
- `ncol()`, `nrow()`, `colnames()`, and `rownames()` for getting data.frame metadata
- `Sys.Date()` and `Sys.time()` for getting current date-time values

There are a huge range of functions out there, whether available in R straight away, or from adding extra functionality.

Understanding how functions work and being able to use them correctly will help you learn, and use R effectively.

12.1 Using a function

A function does some computation on an object. The use of a function consists of:

1. A function's name
2. Parentheses

3. 0 or more inputs

Each input is provided to an **argument** or parameter within a function.

These arguments have names, although you don't often need to provide the names.

You can find out what arguments a function takes by using the code completion and its help snippet, or by searching for the function in the Rstudio Help tab.

When you're inside the brackets of a function you can get the list of available arguments and auto-complete them.

Try getting the code completion to work with the function `tail()`.

12.2 Examining functions

One of the niftiest things about R is being able to see the code for a function. You can examine how many functions work by just typing their name without any parentheses.

```
Sys.Date
```

```
## function ()
## as.Date(as.POSIXlt(Sys.time()))
## <bytecode: 0x000000000725af60>
## <environment: namespace:base>
```

The first line(s) show how the arguments are specified. Subsequent lines show the code and the final lines starting with `<` can be mostly ignored.

12.3 Function input patterns

Functions tend to conform to certain patterns of inputs.

12.3.1 No inputs

Some functions don't require the user to provide info and so they don't have any arguments. `Sys.Date()` and similar functions do not need user input because the functions provide information about the system.

```
Sys.Date
```

```
## function ()
## as.Date(as.POSIXlt(Sys.time()))
## <bytecode: 0x000000000725af60>
## <environment: namespace:base>
```

Looking at the function definition, we can see that there are no arguments specified in the first line.

12.3.2 Single inputs

Other functions only have a single allowed input. `length()` returns the length of an object so it only allows you to provide it with an object.

```
length
```

```
## function (x)  .Primitive("length")
```

We can see in this definition[1] that the function takes the argument `x`.

12.3.3 Many inputs

Some functions have multiple inputs, although not all of them are necessarily **mandatory**. `head()` and `tail()` have been used so far

[1] the code looks a bit odd - this is because it's specified a bit differently to most functions, but fear not! In R, there are some different ways of writing code. The normal way is called **S3** but functions that are designed to work with properties of objects like length use a different system called **S4**. For the most part you'll rarely need to dig into S4 code as most R functionality is built in S3 and will allow you to check out it's code easily.

with only a single input but they take an optional argument as to how many elements should be returned.

```
head(letters)
head(letters, 2)
```

```
## [1] "A" "B" "C" "d" "e" "f"
## [1] "A" "B"
```

The `rnorm()` function allows us to generate a vector of values from a normal distribution. We can tell it how many values we need (`n`), and we can optionally provide the mean (`mean`) and standard deviation (`sd`) to describe the Normal curve that values should be selected from.

```
rnorm
```

```
## function (n, mean = 0, sd = 1)
## .Call(C_rnorm, n, mean, sd)
## <bytecode: 0x000000002c1abf58>
## <environment: namespace:stats>
```

Looking at how `rnorm` is specified we can see that we're expected to provide `n`, but `mean` and `sd` are given values of 0 and 1 respectively by default.

```
rnorm(n = 5)
rnorm(n = 5, mean = 10, sd = 2)
```

```
## [1]  0.6131313 -0.3567561  0.1159489 -1.0260021  0.8866003
## [1]  6.414939 10.017482 11.266005 10.261910  8.227831
```

12.3.4 Unlimited inputs

Other functions can take an unlimited amount of input values. Functions like `sum()` will sum the values from a number of objects.

```
sum(1:3, 1:9, pi)
```

```
## [1] 54.14159
```

The ellipsis (`...`) is used to denote when the user can provide any number of values.

```
sum
```

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

12.4 Naming arguments

Every input provided to a function is associated with an argument.

Each argument must have a name. Even functions that allow unlimited inputs assign these inputs to a name. Behind the scenes, they get put into a list object and the list gets called `...` (or ellipsis).

There are some typical names for arguments that take your data object. These include:

- `x`
- `data`
- `.data`
- `df`

You don't usually have to provide the argument names, just put things in the relevant places in the function. Sometimes though, you *will* need to use argument names.

Here are my rules of thumb for knowing when you need to name names:

1. You're using the arguments in an order that is different from the function author's intended order (you might be skipping some arguments as the default values are fine or you might just prefer a different order)
2. The arguments you want to specify show up after the `...` in a function's argument list
3. You want to give a specific name to a value in a `...` argument

We can provide names[2] for clarity or so we can use arguments out of order if we prefer to.

[2]Note that you don't have to type the full name as R will attempt to match up values, but doing that can and does get a lot of R users into trouble so I don't recommend it.

```
rnorm(n = 5, mean = 10, sd = 2)
rnorm(mean = 10, sd = 2, n = 5)
```

```
## [1]  7.897466  9.872175 10.626260  9.177773 10.422403
## [1]  9.374904  9.676517  3.984936 10.841058  8.245642
```

A common behaviour change that you'll need to work with is how missing (`NA`) values get handled. Functions that allow you change this behaviour, usually have an argument called things like `na.rm`, `na.omit`, and `na.action`.

```
sum(1:5, NA)
sum(1:5, NA, na.rm = TRUE)
```

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

In the `sum()` example, I used the `na.rm` argument's name. This is because otherwise the `TRUE` would be considered part of the values being passed for summing. Without the name, the value gets considered as part of the `....`

```
sum(1:5, NA, TRUE)
```

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

A function will sometimes have `...` at the end of its list of arguments when it utilises other functions and those have optional / default values.

For instance the `predict()` function allows us to take a model we've built and apply it to some new data.

It works for many different types of model and these different models expect different types of inputs. Some models expect data.frames, others expect time series data, etc.

There's lots of potential variations, the only thing that is mandatory is the model object.

```
predict
```

```
## function (object, ...)
## UseMethod("predict")
## <bytecode: 0x000000000e443458>
## <environment: namespace:stats>
```

The `predict()` function then determines what type of model object you've provided it and passes the model, and any other values you provided, to the relevant function, returning back the results.

```
linearMod <- lm(Sepal.Length ~ ., data = iris)
logisticMod <- glm(Species ~ ., data = iris, family = binomial)

predict(linearMod, iris[1, ])
predict(logisticMod, iris[1, ])
```

```
##        1
## 5.004788
##         1
## -38.02709
```

12.5 Summary

R uses functions as the means of performing operations[3].

Functions can take 0 or more arguments. All arguments may be mandatory, but some can be optional or even undefined.

You can use argument names to provide arguments in different orders to that defined by the function author or to provide them in the case where an ellipsis (`...`) is used in a function.

12.6 R functions Exercises

Using what you've learned about investigating the components of functions…

1. Use `pmin()` to find the smallest values element-wise of the three vector `1:51`, `25:75`, `30:-20`

[3]Indeed, even operators like `+` are actually functions behind the scenes

2. Use `paste()` to combine the upper case letters into a single string with `", "` between each letter
3. Use `list.files()` to see what files are in your current directory. Return the fully qualified names not just the file names
4. View the code for `ncol()` and work out how the number of columns is being determined

Chapter 13

R packages

An R package is a bundle of functions and/or datasets. It extends the capabilities that the "base" and "recommended" R packages have. By using packages we can do data manipulation in a variety of ways, produce all sorts of awesome charts, generate books like this, use other languages like Python and JavaScript, and of course, do all sorts of data analysis.

13.1 Installing packages

Once you've identified a package that contains functions or data you're interested in using[1], we need to get the package onto our machine.

To get the package, you can use an R function or you can use the Install button on the Packages tab.

```
install.packages("datasauRus")
```

If you need to install a number of packages, `install.packages()` takes a vector of package names.

[1] Using CRAN or sites like rdrr.io

```
install.packages(c("datasauRus", "tidyverse"))
```

Updating packages involves re-running `install.packages()` and it's usually easier to trigger this by using the Update button on the Packages tab and selecting all the packages you want to update.

13.1.1 Installing from GitHub and other sources

The `install.packages()` function works with CRAN, CRAN mirrors [2], and CRAN-like repositories[3]

If you want to install BioConductor packages, there are some helper scripts available from the BioConductor website, bioconductor.org[4].

Other package sources, such as GitHub, will involve building packages before they can be installed. If you're on Windows, this means you need an additional piece of software called Rtools[5]. The other handy thing you'll need is the package `devtools` (available from CRAN). `devtools` provides a number of functions designed to make it easier to install from GitHub, BitBucket, and other sources.

```
library(devtools)
install_github("lockedata/pRojects")
```

13.2 Recommended packages

Here are my recommended packages – many of these will be covered in later books.

[2] Copies of CRAN made available to avoid overloading central servers.

[3] The most common CRAN-like systems are building your own local CRAN with the package `miniCRAN` and using the package `drat` to make repositories, especially remote repositories.

[4] `http://www.bioconductor.org/install/`

[5] `http://cran.r-project.org/bin/windows/Rtools/`

13.2.1 tidyverse

The `tidyverse` is a suite of packages designed to make your life easier. It's well worth installing and many of the packages in this recommendations section are part of the `tidyverse`.

```
install.packages("tidyverse")
```

13.2.2 Getting data in and out of R

The following packages can be used to get data into, and out of R:

- Working with databases, you can use the `DBI` package and its companion `odbc` to connect to most databases
- To get data from web pages, you can use `rvest`
- To work with APIs, you use `httr`
- To work with CSVs, you can use `readr` or `data.table`.[6]
- To work with SPSS, SAS, and Stata files, use `readr` and `haven`

13.2.3 Data manipulation

The `tidyverse` contains great packages for data manipulation including `dplyr` and `purrr`.

Additionally, a favourite data manipulation package of mine is `data.table`. `data.table` tends to have a bit of a steeper learning curve than the `tidyverse` but it's phenomenal for brevity and performance.

13.2.4 Data visualisation

- For static graphics `ggplot2` is fantastic - it adds a sensible vocabulary to help you construct charts with ease
- `plotly` helps you build interactive charts from scratch or make `ggplot2` charts interactive
- `leaflet` is a great maps package
- `ggraph` helps you build effective network diagrams

[6] `data.table` tends to be faster for CSV read and writes.

13.2.5 Data science

- `caret` is an interface package to many model algorithms and has a raft of insanely useful features itself
- `broom` takes outputs from model functions and makes them into nice data.frames
- `modelr` helps build samples and supplement result sets
- `reticulate` is a package for talking to Python and, therefore, enables you to work with any deep learning framework that is based in Python. `tensorflow` is a package based on `reticulate` and allows you to work with tensorflow in R
- `sparklyr` allows you to run and work with Spark processes on your R data
- `h2o` is a package for working with H2O, a super nifty machine learning platform

13.2.6 Presenting results

- `rmarkdown` is the core package for combining text and code and being able to produce outputs like HTML pages, PDFs, and Word documents
- `bookdown` facilitates books like this
- `revealjs` allows you to make slide decks using `rmarkdown`
- `flexdashboard` and `shiny` allow you to make interactive, reactive dashboards and other analytical apps

13.2.7 Finding packages

As well as using online search facilities like CRAN[7] and rdrr.io[8] for packages, there are some handy packages that help you find other packages!

- `ctv` allows you to get all the packages in a given CRAN task view[9], which are maintained lists of package for various tasks
- `sos` allows you to search for packages and functions that match a keyword

[7] `http://cran.r-project.org/search.html`
[8] `http://rdrr.io`
[9] `http://cran.r-project.org/web/views/`

13.3 Loading a package

To make functions and data from a package available to use, we need to run the `library()` function.

```
library("utils")
```

The `library()` function accepts a vector of length 1, so you need to perform multiple calls to the function to load up multiple packages.

```
library("utils")
library("stats")
```

Once a package is loaded, you can then use any of its functions.

You can find what functions are available in a package by looking at it's help page.

Alternatively, you can type the package's name and hit Tab. This auto-completes the package's name, adds two colons (`::`) and then shows the list of available functions for that package. The double colon trick is very helpful for when you want to browse package functionality.

```
utils::find()
```

Any function in R can be prefixed with its package name and the double colon (`::`) - this is great for telling people where functions are coming from and for tracking dependencies in long scripts.

13.4 Learning how to use a package

R documentation is some of the best out there.

Yes, I will complain about the impenetrable statistical jargon some package authors use, but the CRAN gatekeepers require that packages generally have a really high standard of documentation.

Every function you use will have a help page associated with it. This page usually contains a description, shows what parameters the function has, what those parameters are, and most importantly, there's usually examples.

To navigate to the help page of an individual function in an R package you:

- Hit F1 on a function name in a script
- Type `??fnName` and send to the console

```
`?`(`?`(mean))
```

- Search in the Help tab
- Use the `help()` function to open up the packages index page and navigate to the relevant function

```
help(package = "utils")
```

- Find the relevant package in the Packages tab and click on it. Scroll through the index that opens up on the Help page to find the right function

As well as the function level documentation, good packages also provide a higher level of documentation that covers workflows using the packages, how to extend package functionality, or outlines any methodologies or research that led to the package.

These pieces of documentation are called **vignettes**. They are accessible on the package's index page or you can use the function `vignette()` to read them.

```
vignette("multi")
```

13.5 Summary

R packages bundle functionality and/or data.

You can install packages from the central public repository (CRAN) via `install.packages()` or install them from GitHub with the package `devtools`. R packages contain documentation that helps

you understand how functions work and how the package overall works.

When you want to make use of functionality from a package you can either load all of a package's functionality by using the `library()` function or refer to a specific function by prefixing the function with the package name and two colons (`::`) e.g. `utils::help("mean")`.

There are many packages out there for different activities and domain-specific types of analysis. Use online search facilities like rdrr.io[10] or CRAN task views[11] to find ones specific to your requirements.

13.6 R packages Exercises

1. Install `datasauRus`
2. Load the library `datasauRus`
3. Browse `datasauRus`'s help pages
4. Read the `datasauRus` vignette

[10] `http://rdrr.io`
[11] `http://cran.r-project.org/web/views/`

Part V

Conclusion

Chapter 14

Conclusion

You've reached he end of this first book in the series on R fundamentals. Thank you for reading!

In this book you learnt how to:

- effectively use your RStudio environment
- perform basic tasks like importing data, subsetting and exploring data, and
- use R functions and new packages

The next book will tackle data manipulation and tabular based analysis, the next step in getting to grips with the fundamentals of R.

Appendix A

Answers

A.1 R Data Types Exercises

- Convert TRUE to an integer

```
as.integer(TRUE)
```

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

- What is the datatype of the value returned by `Sys.time()`?

```
class(Sys.time())
```

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

- What is the datatype of the value returned by `Sys.timezone()`?

```
class(Sys.timezone())
```

```
## [1] "character"
```

- Make this quote into an R string > "Do you think this is a game?", he said. "No I think Jenga's a game", Archer responded.

```
'"Do you think this is a game?", he said.
 "No I think Jenga\'s a game", Archer responded.'
```

```
## [1] "\"Do you think this is a game?\", he said. \n \"No
I think Jenga's a game\", Archer responded."
```

A.2 Basic Operations Exercises

- What is the result of `pi^2`?

```
pi^2
```

```
## [1] 9.869604
```

- Is pi greater than 3?

```
pi > 3
```

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

- Construct a statement to check if 5 is both greater than 3 and less than or equal to 6

```
(5 > 3) & (5 <= 6)
```

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

- What are the results if you check to see if a sequence of 1 to 5 is less than or equal to 3?

```
1:5 <= 3
```

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

A.3 R Objects Exercises

- See what's in the built-in variable `letters`

```
letters
```

```
## [1] "A" "B" "C" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
"n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z" "|"
```

- Write a check to see if "A" is present in `letters`

```
"A" %in% letters
```

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

- Find out which values in the sequence 1 to 10 are greater than or equal to 3 and less than 7.

```
myseq <- 1:10
myseq >= 3 & myseq < 7
```

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

- Make a vector containing the numbers 1 to 50;

```
fifty <- 1:50
```

- Make a vector containing two words;

```
words <- c("fifty", "words")
```

- What happens when you combine these two vectors?

```
c(fifty, words)
# the numbers get converted to text
```

```
## [1] "1" "2" "3" "4" "5" "6" "7" "8"
## [9] "9" "10" "11" "12" "13" "14" "15" "16"
## [17] "17" "18" "19" "20" "21" "22" "23" "24"
## [25] "25" "26" "27" "28" "29" "30" "31" "32"
## [33] "33" "34" "35" "36" "37" "38" "39" "40"
## [41] "41" "42" "43" "44" "45" "46" "47" "48"
```

```
## [49] "49" "50" "fifty" "words"
```

- Make a data.frame using the two vectors

```
fiftywords <- data.frame(fifty, words)
```

- What happened to your text vector?

```
# It got recycled 25 times
```

- Make a list containing some of the variables you've created so far.

```
list(fifty, words, fiftywords)
```

```
## [[1]]
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23
## [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46
## [47] 47 48 49 50
##
## [[2]]
## [1] "fifty" "words"
##
## [[3]]
## fifty words
## 1 1 fifty
## 2 2 words
## 3 3 fifty
## 4 4 words
## 5 5 fifty
## 6 6 words
## 7 7 fifty
## 8 8 words
## 9 9 fifty
## 10 10 words
## 11 11 fifty
## 12 12 words
## 13 13 fifty
## 14 14 words
```

```
## 15 15 fifty
## 16 16 words
## 17 17 fifty
## 18 18 words
## 19 19 fifty
## 20 20 words
## 21 21 fifty
## 22 22 words
## 23 23 fifty
## 24 24 words
## 25 25 fifty
## 26 26 words
## 27 27 fifty
## 28 28 words
## 29 29 fifty
## 30 30 words
## 31 31 fifty
## 32 32 words
## 33 33 fifty
## 34 34 words
## 35 35 fifty
## 36 36 words
## 37 37 fifty
## 38 38 words
## 39 39 fifty
## 40 40 words
## 41 41 fifty
## 42 42 words
## 43 43 fifty
## 44 44 words
## 45 45 fifty
## 46 46 words
## 47 47 fifty
## 48 48 words
## 49 49 fifty
## 50 50 words
```

- Return the some rows from the `iris` dataset

```
head(iris)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

A.4 Data manipulation exercises

- Select all `LETTERS` before "X";

```
LETTERS[LETTERS < "X"]
```

```
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M"
"N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W"
```

- Select the first 5 rows from the built-in data.frame `iris`;

```
iris[1:5, ]
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa

- Select the first 2 columns from `iris`;

```
head(iris[, 1:2])
```

Sepal.Length	Sepal.Width
5.1	3.5
4.9	3.0
4.7	3.2
4.6	3.1
5.0	3.6
5.4	3.9

- Select the column Sepal.Length from `iris` by name.

```
head(iris[, "Sepal.Length"])
```

```
## [1] 5.1 4.9 4.7 4.6 5.0 5.4
```

- Select rows from the `iris` data.frame where the `Sepal.Length` is greater than 5.8cm;

```
head(iris[iris$Sepal.Length > 5.8, ])
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
51	7.0	3.2	4.7	1.4	versicolor
52	6.4	3.2	4.5	1.5	versicolor
53	6.9	3.1	4.9	1.5	versicolor
55	6.5	2.8	4.6	1.5	versicolor
57	6.3	3.3	4.7	1.6	versicolor
59	6.6	2.9	4.6	1.3	versicolor

- Select rows from the `iris` data.frame where the Sepal.Width is below the average for that column.

```
head(iris[iris$Sepal.Width < mean(iris$Sepal.Width), ])
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
2	4.9	3.0	1.4	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
13	4.8	3.0	1.4	0.1	setosa
14	4.3	3.0	1.1	0.1	setosa
26	5.0	3.0	1.6	0.2	setosa
39	4.4	3.0	1.3	0.2	setosa

- Select everything from `iris` except the Species column;

```
head(iris[, -5])
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
5.1	3.5	1.4	0.2
4.9	3.0	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5.0	3.6	1.4	0.2
5.4	3.9	1.7	0.4

- Create a copy of the `iris` data that just contains the first 100 rows and call it `myIris`;

```
myIris <- iris[1:100, ]
```

- Update the species column to the value "Unknown" in `myIris`;

```
myIris$Species <- "Unknown"
```

- Delete rows from `myIris` where the sepal length is greater than 5.5.

```
myIris <- myIris[myIris$Sepal.Length <= 5.5, ]
```

A.5 R functions Exercises

1. Use `pmin()` to find the smallest values element-wise of the three vector `1:51`, `25:75`, `30:-20`

```
pmin(1:51, 25:75, 30:-20)
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 15 14
## [18] 13 12 11 10 9 8 7 6 5 4 3 2 1 0 -1 -2 -3
## [35] -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17
```

```
-18 -19 -20
```

- Use `paste()` to combine the upper case letters into a single string with `", "` between each letter

```
paste(LETTERS, collapse = ", ")
```

```
## [1] "A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q,
R, S, T, U, V, W, X, Y, Z"
```

- Use `list.files()` to see what files are in your current directory. Return the fully qualified names not just the filenames.

```
list.files(full.names = TRUE)
```

- View the code for `ncol()` and work out how the number of columns is being determined.

```
ncol

# ncol performs the `dim()` function on the
# object we pass in. This returns a vector
# of length 2. We then subset to the second
# element, which is the number of columns.
```

```
## function (x)
## dim(x)[2L]
## <bytecode: 0x000000001d5d2c18>
## <environment: namespace:base>
```

A.6 R packages Exercises

- Install `datasauRus`

```
install.packages("datasauRus")
```

- Load the library `datasauRus`;

```
library(datasauRus)
```

- Browse `datasauRus`'s help pages;

```
help(package = "datasauRus")
```

- Read the `datasauRus` vignette;

```
vignette("Datasaurus")
```