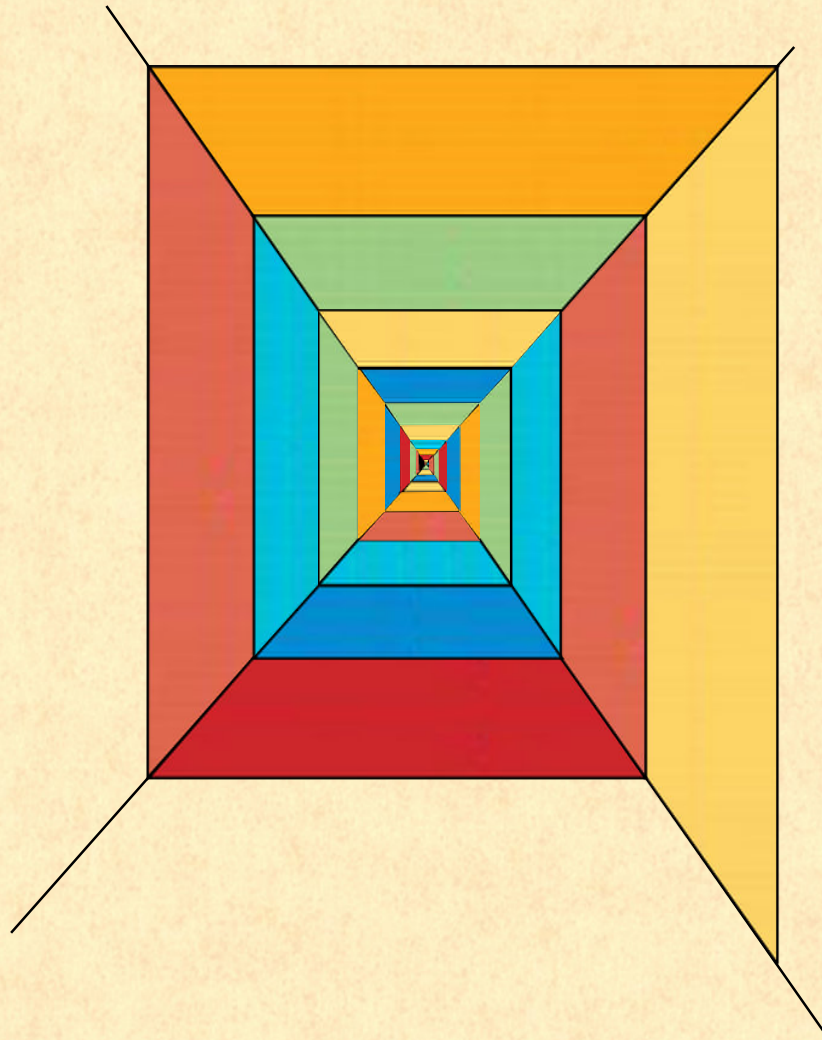# An Introduction to R

Alex Douglas, Deon Roos,
Francesca Mancini, Ana Couto
& David Lusseau

An Introduction to R

Alex Douglas, Deon Roos, Francesca Mancini, Ana Couto & David Lusseau

April 9, 2024

Contents

Preface

0.1   The aim of this book

The aim of this book is to introduce you to using R, a powerful and flexible interactive environment for statistical computing and research. R in itself is not difficult to learn, but as with learning any new language (spoken or computer) the initial learning curve can be a little steep and somewhat daunting. We have tried to simplify the content of this book as much as possible and have based it on our own personal experience of teaching (and learning) R over the last 15 years. It is not intended to cover everything there is to know about R - that would be an impossible task. Neither is it intended to be an introductory statistics course, although you will be using some simple statistics to highlight some of R's capabilities. The main aim of this book is to help you climb the initial learning curve and provide you with the basic skills and experience (and confidence!) to enable you to further your experience in using R.

0.2   Who is this book for?

We hope this book will be a useful introduction for anyone who wants to learn how to use R. It started out as a 100 page pdf manual used to support our teaching in numerous undergraduate and postgraduate biostatistics classes at the University of Aberdeen and has evolved over the last 10 years to the book you're now reading. Consequently, this book is primarily written with the advanced undergraduate or postgraduate student in mind although the material should be useful for anyone with an interest in learning R regardless of background. Our book assumes that you have no previous experience of using either R or RStudio and no background in programming or using command-line driven software. We've also tried to make the content of this book operating system agnostic and have included information for Windows, Mac and Linux users where appropriate.

0.3   Why an open book?

We decided to write this book as a freely available open source document for a number of reasons. Firstly, we hope this book will be a living document that naturally evolves over time. If you find any errors in the book or want to make a suggestion on how to improve it, please open an issue on the GitHub site or even better click on the edit button ✎ in the navigation bar at the top of this page and issue a pull request (only available on the web version of this book). Please note that this book uses a Contributor Code of Conduct. By contributing to this book, you agree to abide by its terms. If you feel like contributing to the book, please get in touch, we'd be happy to hear from you. If you want to take a peek at all the code behind the book visit the book GitHub repo. Secondly, the release of the bookdown package by Yihui Xie has made it incredibly easy to collaboratively write, update and maintain this book using RMarkdown, RStudio and GitHub. Without the bookdown package this book wouldn't exist in its current format. Lastly, R is not just software, it's also a wonderful open community with many thousands of contributors all working to make R even easier to learn and use. This is our small contribution.

0.4   Who are we?

Alex Douglas is a Professor of Biostatistics at the University of Aberdeen and has been teaching (and learning) R since version 2.1.

- Web:https://www.abdn.ac.uk/people/a.douglas/
- X:https://twitter.com/Scedacity
- GitHub:https://github.com/alexd106

Deon Roos is a Lecturer in Applied Statistics at the University of Aberdeen. He's been using and teaching R for 10 years.

- X:https://twitter.com/DeonRoos88
- GitHub:https://github.com/Deon88

Ana Couto is an ecological statistician. She has been using and teaching R for 9 years.

- Web:https://t.co/KRUlw1Grjr
- X:https://twitter.com/AnaSofiaCouto
- GitHub:https://github.com/Anofia

Francesca Mancini is an ecological modeller, working with large citizen science datasets and statistical models to derive trends in biodiversity change and investigate its drivers.

- Web:https://FrancescaMancini.github.io
- X:https://twitter.com/frances_mancini
- GitHub:https://github.com/FrancescaMancini

David Lusseau is a Professor of Marine Sustainability at the Danish Technical University. He's been using R for 17 years and teaching it for 14 years.

- X:https://twitter.com/lusseau
- GitHub:https://github.com/dlusseau

0.5 How to use this book

For the best experience we recommend that you read the web version of this book. The web version includes a navbar at the top of the page where you can toggle the sidebar on and off ☰, search through the book 🔍, change the font, font size and page colour **A** and suggest revisions if you spot a typo or mistake ✏. You can also download a pdf version of our book by clicking on the pdf icon 📄.

We use a few typographical conventions throughout this book.

R code and the resulting output are presented in code blocks in our book.

```
42 + 1
## [1] 43
```

If you're running R code in the R console it will look a little different from the code above. Code should be entered after the command prompt (`>`) and the output from code in the console will not be commented with `##`.

```
> 42 + 1
[1] 43
```

In the book text we refer to R functions using code font followed by a set of round brackets, i.e. `mean()` or `sd()` etc.

We refer to objects that we create using code font without the round brackets, i.e. `obj1`, `obj2` etc.

A series of actions required to access menu commands in RStudio are identified as `File -> New File -> R Script` which translates to 'click on the File menu, then click on New File and then select R Script'.

Clickable links to sections in the book or to external websites are highlighted in light blue text, i.e. link.

Links to short 'how-to' videos that accompany this book are indicated by a video 🎥 icon.

Links to exercises can be found at the end of each Chapter and are highlighted by a info ⓘ icon.

0.6 Book website

Although you can use this book as a standalone resource, we recommend you use it in conjunction with it's companion website. The course website contains a series of exercises which will help you practice writing R code and test your understanding of key concepts - you certainly won't learn how to use R by watching other people do it (or reading a book about it!). The website also contains solutions for each of the exercises and a plethora of links to additional tutorials and resources.

0.7 Some R pointers

A question we get asked fairly regularly is *'what's the best/easiest way to learn R?'*. Unfortunately, we don't have a ready answer to this question as everyone tends to learn R in their own way and at their own pace. Having said that, here are a few things to bear in mind that might help:

- Use R often and use it regularly - find any excuse to fire up RStudio (or just R) and get coding. This will help build and maintain all important momentum.

- Learning R is not a memory test. One of the beauties of a scripting language is that you will always have your code to refer back to when you inevitably forget how to do something.

- You don't need to know everything there is to know about R to use it productively. If you get stuck, Google it, it's not cheating and writing a good search query is a skill in itself. Just make sure you check thoroughly that the code you find is doing what you want it to do.

- If you find yourself staring at code for hours trying to figure out why it's not working then walk away for a few minutes. We've lost count of the number of times we were able to spot our mistake almost immediately after returning from a short caffeine break.

- In R there are many ways to tackle a particular problem. If your code doesn't look like someone else's, but it does what you want it to do, in a reasonable time and robustly then don't worry about it - job done.

- Related to our previous point, remember R is just a tool to help you answer your interesting questions. Although it can be fun to immerse yourself in all things R (we often do), don't lose sight of what's important - your research question(s) and your data. No amount of skill using R will help if your data collection is fundamentally flawed or your question vague.

- Recognise that there will be times when things will get a little tough or frustrating. Try to accept these periods as part of the natural process of learning a new skill (we've all been there) and remember, the time and energy you invest now will be more than recouped in the not too distant future.

Finally, once you've finished working your way through this book, we encourage you to practice what you've learned using your own data. If you don't have any data yet, then ask your colleagues / friends / family for some (we're sure they will be delighted!) or follow one of the many excellent tutorials available online (see the course website for more details). Our suggestion to you, is that while you are getting to grips with R, uninstall any other statistics software you have on your computer and only use R. This may seem a little extreme but will hopefully remove the temptation to *'just do it quickly'* in a more familiar environment and consequently slow down your learning of R. Believe us, anything you can do in your existing statistics software package you can do in R - often more robustly and efficiently.

Good luck and don't forget to have fun.

0.8 Thanks

The vast majority of the content of this book has been used to help teach our applied statistics courses in the School of Biological Sciences and School of Medicine, Medical Sciences and Nutrition at the University of Aberdeen. It's been tremendous fun to meet and interact with the 1000's of students who have attended these courses over the last 15 years, many of whom helped improve this book by suggesting changes and spotting our daft mistakes (we still own any that remain!). Teaching these courses would have been much more difficult (and a lot less fun) without the enthusiastic help of a small army of postgraduate instructors. Their skill, patience and passion for teaching R and statistics has been a constant source of inspiration over the years. Thank you all (you know who you are!).

Thanks also to the following who have generously contributed fixes and improvements to the book via pull requests, raising an issue or email:

- Frédéric Santos (@frederic-santos)
- azzco (@azzco)
- Cian McAuley
- Constantinos C (@Gesler16)
- Lee Sanders (@LSanders1)
- Dynamic6844 (@Dynamic6844)
- Antonio Castellano Albors
- Nazmul Hasan (@nazmulhasan202)
- Metehan Gungor (@gungorMetehan)
- Irvine Tatenda Mutiti (@it1mutiti)
- Katarzyna Zejc

0.9 License

We want you to be able to enjoy this book completely free of charge so we've licensed it according to the Creative Commons Attribution-NonCommercial 4.0 International (CC BY-NC 4.0) License (https://creativecommons.org/licenses/by-nc/4.0/).

If you teach R, feel free to use some or all of the content in this book to help support your own students. The only thing we ask is that you acknowledge the original source and authors. If you find this book useful or have any comments or suggestions we would love to hear from you (you'll find us lurking on X and GitHub).

Chapter 1

Getting started with R and RStudio

Although R is not new, its popularity has increased rapidly over the last 10 years or so (see here for some interesting data). It was originally created and developed by **R**oss Ihaka and **R**obert Gentleman during the 1990's with the first stable version released in 2000. Nowadays R is maintained by the R Development Core Team. So, why has R become so popular and why should you learn how to use it? Some reasons include:

- R is open source and freely available.
- R is available for Windows, Mac and Linux operating systems.
- R has an extensive and coherent set of tools for statistical analysis.
- R has an extensive and highly flexible graphical facility capable of producing publication quality figures.
- R has an expanding set of freely available 'packages' to extend R's capabilities.
- R has an extensive support network with numerous online and freely available documents.

All of the reasons above are great reasons to use R. However, in our opinion, the single biggest reason to use R is that it facilitates robust and reproducible research practices. In contrast to more traditional 'point and click' software, writing code to perform your analysis ensures you have a permanent and accurate record of all the methods you used (and decisions you made) whilst analysing your data. You are then able to share this code (and your data) with other researchers / colleagues / journal reviewers who will be able to reproduce your analysis exactly. This is one of the tenets of open science. We will cover other topics to facilitate open science throughout this book, including creating reproducible reports and version control.

In this Chapter we'll show you how to download and install R and RStudio on your computer, give you a brief RStudio orientation including working with RStudio Projects, installing and working with R packages to extend R's capabilities, some good habits to get into when working on projects and finally some advice on documenting your workflow and writing nice readable R code.

1.1   Installing R

To get up and running the first thing you need to do is install R. R is freely available for Windows, Mac and Linux operating systems from the Comprehensive R Archive Network (CRAN) website. For Windows and Mac users we suggest you download and install the pre-compiled binary versions.

See this video for step-by-step instructions on how to download and install R and RStudio

1.1.1   Windows users

For Windows users select the 'Download R for Windows' link and then click on the 'base' link and finally the download link 'Download R 4.3.3 for Windows'. This will begin the download of the '.exe' installation file. When the download has completed double click on the R executable file and follow the on-screen instructions. Full installation instructions can be found at the CRAN website.

1.1.2   Mac users

For Mac users select the 'Download R for (Mac) OS X' link. The binary can be downloaded by selecting the 'R-4.3.3.pkg'. Once downloaded, double click on the file icon and follow the on-screen instructions to guide you through the necessary steps. See the 'R for Mac OS X FAQ' for further information on installation.

1.1.3   Linux users

For Linux users, the installation method will depend on which flavour of Linux you are using. There are reasonably comprehensive instruction here for Debian, Redhat, Suse and Ubuntu. In most cases you can just use your OS package manager to install R from the official repository. On Ubuntu fire up a shell (Terminal) and use (you will need root permission to do this):

```
sudo apt update
sudo apt install r-base r-base-dev
```

which will install base R and also the development version of base R (you only need this if you want to compile R packages from source but it doesn't hurt to have it).

If you receive an error after running the code above you may need to add a 'source.list' entry to your etc/apt/sources.list file. To do this open the /etc/apt/sources.list file in your favourite text editor (gedit, vim, nano etc) and add the following line (you will need root permission to do this):

```
deb https://cloud.r-project.org/bin/linux/ubuntu disco-cran35/
```

This is the source.list for the latest version of Ubuntu (19.04 Disco Dingoat the time of writing). If you're using an earlier version of Ubuntu then replace the source.list entry to the one which corresponds to the version of Ubuntu you are using (see here for an up to date list). Once you have done this then re-run the apt commands above and you should be good to go.

1.1.4 Testing R

Whichever operating system you're using, once you have installed R you need to check its working properly. The easiest way to do this is to start R by double clicking on the R icon (Windows or Mac) or by typing R into the Console (Linux). You should see the R Console and you should be able to type R commands into the Console after the command prompt >. Try typing the following R code and then press enter (don't worry if you don't understand this - we're just checking if R works)

```r
plot(1:10)
```
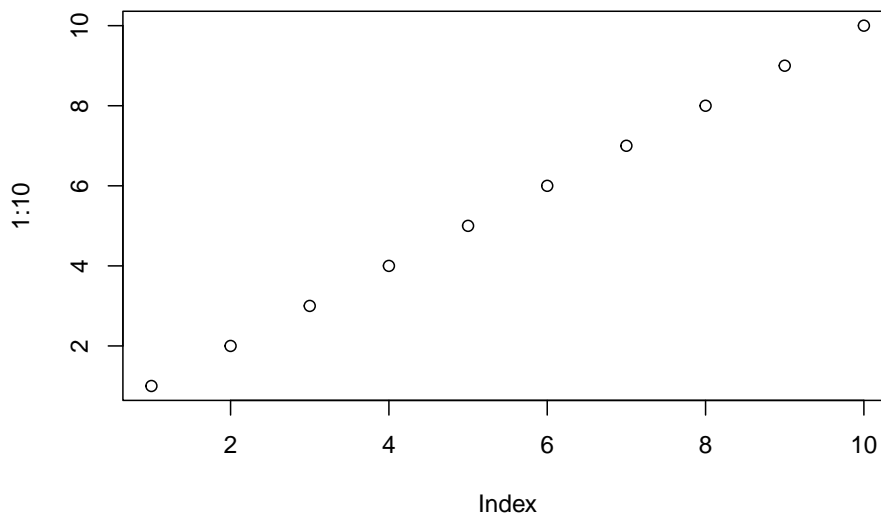

A plot of the numbers 1 to 10 on both the x and y axes should appear. If you see this, you're good to go. If not then we suggest you make a note of any errors produced and then use Google to troubleshoot.

1.2 Installing RStudio

Whilst its eminently possible to just use the base installation of R (many people do), we will be using a popular **I**ntegrated **D**evelopment **E**nvironment (IDE) called RStudio.

RStudio can be thought of as an add-on to R which provides a more user-friendly interface, incorporating the R Console, a script editor and other useful functionality (like R markdown and GitHub integration). You can find more information about RStudio here.

RStudio is freely available for Windows, Mac and Linux operating systems and can be downloaded from the RStudio site. You should select the 'RStudio Desktop' version. Note: you must install R before you install RStudio (see previous section for details).

See this video for step-by-step instructions on how to download and install R and RStudio

1.2.1   Windows and Mac users

For Windows and Mac users you should be presented with the appropriate link for downloading. Click on this link and once downloaded run the installer and follow the instructions. If you don't see the link then scroll down to the 'All Installers' section and choose the link manually.

1.2.2   Linux users

For Linux users scroll down to the 'All Installers' section and choose the appropriate link to download the binary for your Linux operating system. RStudio for Ubuntu (and Debian) is available as a `*.deb` package. The easiest way to install `deb` files on Ubuntu is by using the `gdebi` command. If `gdebi` is not available on your system you can install it by using the following command in the Terminal (you will need root permission to do this)

```
sudo apt update
sudo apt install gdebi-core
```

To install the `*.deb` file navigate to where you downloaded the file and then enter the following command with root permission

```
sudo gdebi rstudio-xenial-1.2.5XXX-amd64.deb
```

where '-1.2.5XXX' is the current version for Ubuntu (rstudio-xenial-1.2.5019-amd64.deb at the time of writing). You can then start RStudio from the Console by simply typing

```
rstudio
```

or you can create a shortcut on your Desktop for easy startup.

1.2.3 Testing RStudio

Once installed, you can check everything is working by starting up RStudio (you don't need to start R as well, just RStudio). You should see something like the image below (if you're on a Windows or Linux computer there may be small cosmetic differences).
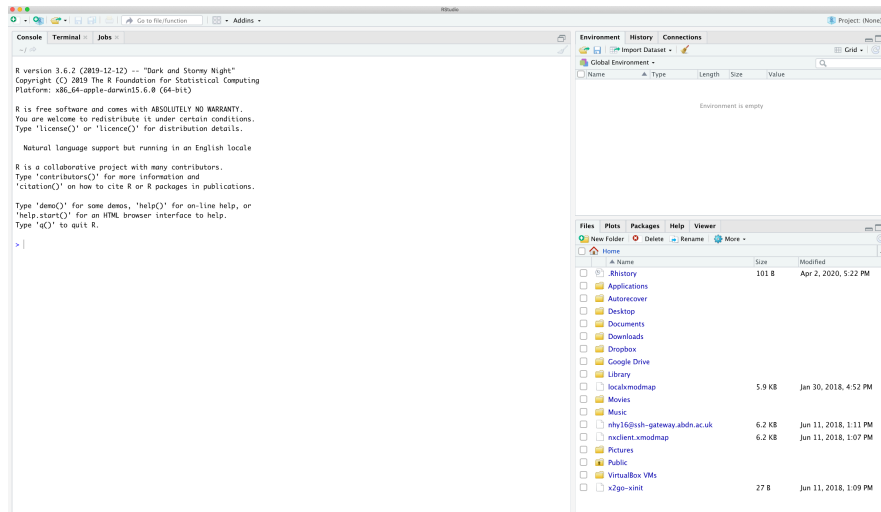
1.3 RStudio orientation

When you open R studio for the first time you should see the following layout (it might look slightly different on a Windows computer).
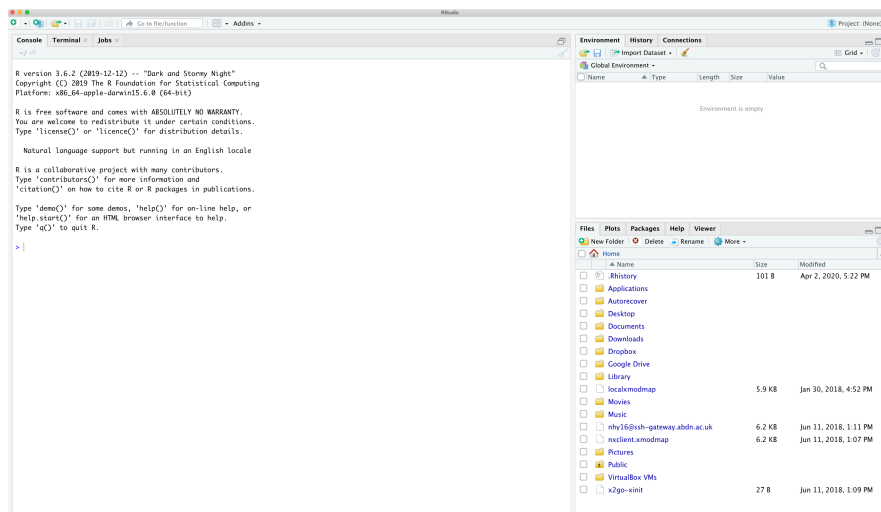
The large window (aka pane) on the left is the **Console** window. The window on the top right is the **Environment / History / Connections** pane and the bottom
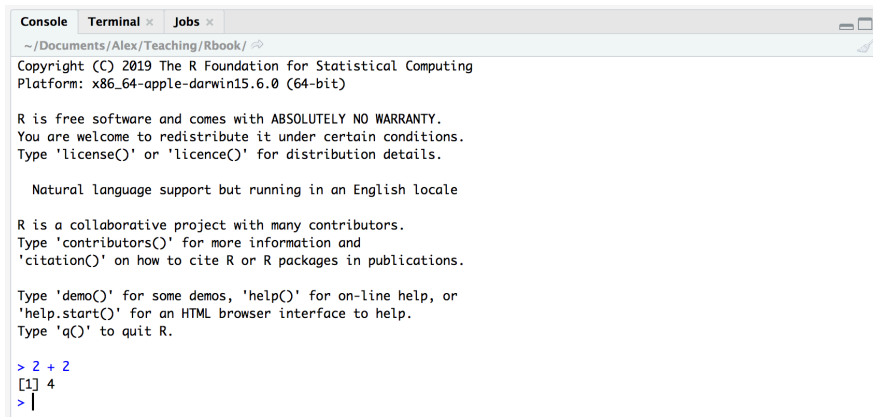
right window is the **Files / Plots / Packages / Help / Viewer** window. We will discuss each of these panes in turn below. You can customise the location of each pane by clicking on the 'Tools' menu then selecting Global Options –> Pane Layout. You can resize the panes by clicking and dragging the middle of the window borders in the direction you want. There are a plethora of other ways to customise RStudio.

See this video for a quick introduction to RStudio

1.3.1   Console

The Console is the workhorse of R. This is where R evaluates all the code you write. You can type R code directly into the Console at the command line prompt, `>`. For example, if you type `2 + 2` into the Console you should obtain the answer `4` (reassuringly). Don't worry about the `[1]` at the start of the line for now.

```
Console   Terminal ×   Jobs ×
~/Documents/Alex/Teaching/Rbook/
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

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

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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

> 2 + 2
[1] 4
>
```

However, once you start writing more R code this becomes rather cumbersome. Instead of typing R code directly into the Console a better approach is to create an R script. An R script is just a plain text file with a `.R` file extension which contains your lines of R code. These lines of code are then sourced into the R Console line by line. To create a new R script click on the 'File' menu then select New File –> R Script.
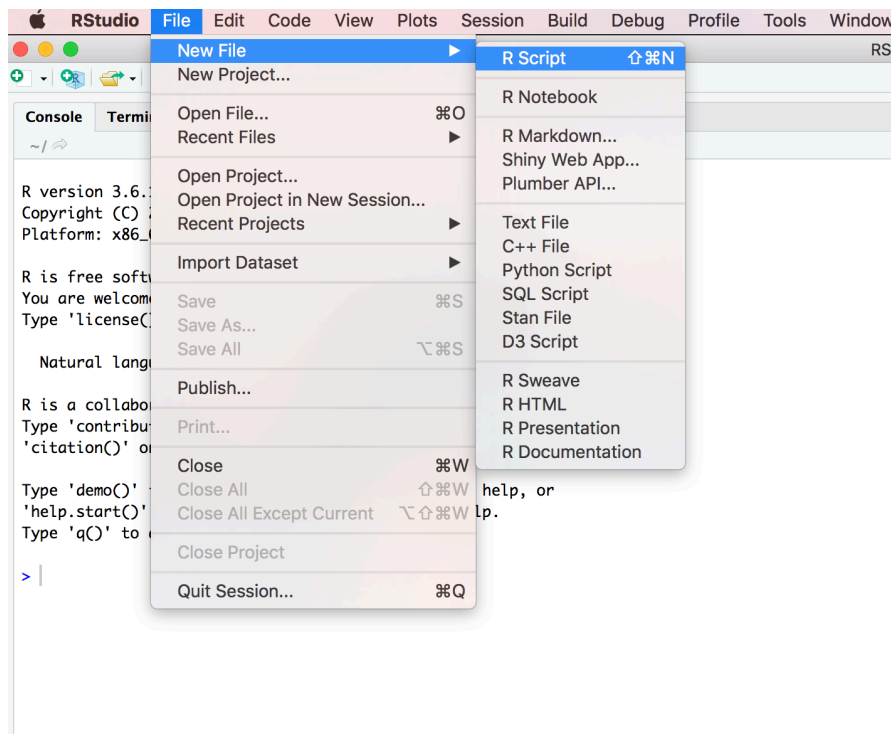
Notice that you have a new window (called the Source pane) in the top left of RStudio and the Console is now in the bottom left position. The new window is a script editor and where you will write your code.
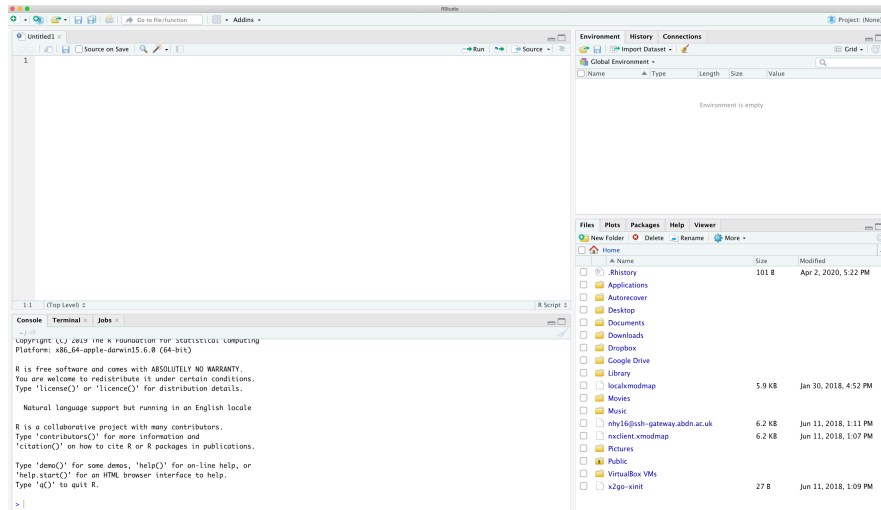
To source your code from your script editor to the Console simply place your cursor on the line of code and then click on the 'Run' button in the top right of the script editor pane.
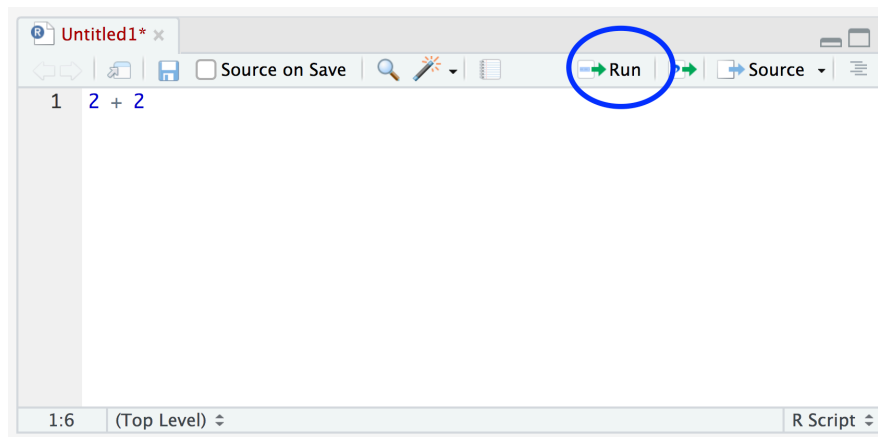
You should see the result in the Console window. If clicking on the 'Run' button starts to become tiresome you can use the keyboard shortcut 'ctrl + enter' (on Windows and Linux) or 'cmd + enter' (on Mac). You can save your R scripts as a .ʀ file by selecting the 'File' menu and clicking on save. Notice that the file name in the tab will turn red to remind you that you have unsaved changes. To open your R script in RStudio select the 'File' menu and then 'Open File…'. Finally, its worth noting that although R scripts are saved with a .ʀ extension they are actually just plain text files which can be opened with any text editor.

1.3.2 Environment/History/Connections

The Environment / History / Connections window shows you lots of useful information. You can access each component by clicking on the appropriate tab in the pane.

- The 'Environment' tab displays all the objects you have created in the current (global) environment. These objects can be things like data you have imported or functions you have written. Objects can be displayed as a List or in Grid format by selecting your choice from the drop down button on the top right of the window. If you're in the Grid format you can remove objects from the environment by placing a tick in the empty box next to the object name and then click on the broom icon. There's also an 'Import Dataset' button which will import data saved in a variety of file formats. However, we would suggest that you don't use this approach to import your data as it's not reproducible and therefore not robust (see Chapter 3 for more details).

- The 'History' tab contains a list of all the commands you have entered into the R Console. You can search back through your history for the line of code you have forgotten, send selected code back to the Console or Source window. We usually never use this as we always refer back to our R script.

- The 'Connections' tab allows you to connect to various data sources such as external databases.

1.3.3 Files/Plots/Packages/Help/Viewer

- The 'Files' tab lists all external files and directories in the current working directory on your computer. It works like file explorer (Windows) or Finder (Mac). You can open, copy, rename, move and delete files listed in the window.

- The 'Plots' tab is where all the plots you create in R are displayed (unless you tell R otherwise). You can 'zoom' into the plots to make them larger using the magnifying glass button, and scroll back through previously created plots using the arrow buttons. There is also the option of exporting plots to an external file using the 'Export' drop down menu. Plots can be exported in various file formats such as jpeg, png, pdf, tiff or copied to the clipboard (although you are probably better off using the appropriate R functions to do this - see Chapter 4 for more details).

- The 'Packages' tab lists all of the packages that you have installed on your computer. You can also install new packages and update existing packages by clicking on the 'Install' and 'Update' buttons respectively.

- The 'Help' tab displays the R help documentation for any function. We will go over how to view the help files and how to search for help in Chapter 2.

- The 'Viewer' tab displays local web content such as web graphics generated by some packages.

1.4 Alternatives to RStudio

Although RStudio is becoming increasingly popular it might not be the best choice for everyone and you certainly don't have to use it to use R effectively. Rather than using an 'all in one' IDE many people choose to use R and a separate script editor to write and execute R code. If you're not familiar with what a script editor is, you can think of it as a bit like a word processor but specifically designed for writing code. Happily, there are many script editors freely available so feel free to download and experiment until you find one you like. Some script editors are only available for certain operating systems and not all are specific to R. Suggestions for script editors are provided below. Which one you choose is up to you: one of the great things about R is that *YOU* get to choose how you want to use R.

1.4.1 Advanced text editors

A light yet efficient way to write your R scripts using advanced text editors such as:

- Atom (all operating systems)
- BBedit (Mac OS)
- gedit (Linux; comes with most Linux distributions)
- MacVim (Mac OS)

- Nano (Linux)
- Notepad++ (Windows)
- Sublime Text (all operating systems)

1.4.2   Integrated development environments

These environments are more powerful than simple text editors, and are similar to RStudio:

- Emacs and its extension Emacs Speaks Statistics (all operating systems)
- RKWard (Linux)
- Tinn-R (Windows)
- vim and its extension NVim-R (Linux)

1.5   R packages

The base installation of R comes with many useful packages as standard. These packages will contain many of the functions you will use on a daily basis. However, as you start using R for more diverse projects (and as your own use of R evolves) you will find that there comes a time when you will need to extend R's capabilities. Happily, many thousands of R users have developed useful code and shared this code as installable packages. You can think of a package as a collection of functions, data and help files collated into a well defined standard structure which you can download and install in R. These packages can be downloaded from a variety of sources but the most popular are CRAN, Bioconductor and GitHub. Currently, CRAN hosts over 15000 packages and is the official repository for user contributed R packages. Bioconductor provides open source software oriented towards bioinformatics and hosts over 1800 R packages. GitHub is a website that hosts git repositories for all sorts of software and projects (not just R). Often, cutting edge development versions of R packages are hosted on GitHub so if you need all the new bells and whistles then this may be an option. However, a potential downside of using the development version of an R package is that it might not be as stable as the version hosted on CRAN (it's in development!) and updating packages won't be automatic.

1.5.1   CRAN packages

See this video for step-by-step instruction on how to install, use and update packages from CRAN

To install a package from CRAN you can use the `install.packages()` function. For example if you want to install the `remotes` package enter the following code into the Console window of RStudio (note: you will need a working internet connection to do this)

```r
install.packages('remotes', dependencies = TRUE)
```

You may be asked to select a CRAN mirror, just select '0-cloud' or a mirror near to your location. The `dependencies = TRUE` argument ensures that additional packages that are required will also be installed.

It's good practice to occasionally update your previously installed packages to get access to new functionality and bug fixes. To update CRAN packages you can use the `update.packages()` function (you will need a working internet connection for this).

```r
update.packages(ask = FALSE)
```

The `ask = FALSE` argument avoids having to confirm every package download which can be a pain if you have many packages installed.

1.5.2 Bioconductor packages

To install packages from Bioconductor the process is a little different. You first need to install the `BiocManager` package. You only need to do this once unless you subsequently reinstall or upgrade R.

```r
install.packages('BiocManager', dependencies = TRUE)
```

Once the BiocManager package has been installed you can either install all of the 'core' Bioconductor packages with

```r
BiocManager::install()
```

or install specific packages such as the 'GenomicRanges' and 'edgeR' packages.

```r
BiocManager::install(c("GenomicRanges", "edgeR"))
```

To update Bioconductor packages just use the `BiocManager::install()` function again.

```
BiocManager::install(ask = FALSE)
```

Again, you can use the `ask = FALSE` argument to avoid having to confirm every package download.

1.5.3 GitHub packages

There are multiple options for installing packages hosted on GitHub. Perhaps the most efficient method is to use the `install_github()` function from the `remotes` package (you installed this package previously). Before you use the function you will need to know the GitHub username of the repository owner and also the name of the repository. For example, the development version of `dplyr` from Hadley Wickham is hosted on the tidyverse GitHub account and has the repository name 'dplyr' (just Google 'github dplyr'). To install this version from GitHub, use

```
remotes::install_github('tidyverse/dplyr')
```

The safest way (that we know of) to update a package installed from GitHub is to just reinstall it using the above command.

1.5.4 Using packages

Once you have installed a package onto your computer it is not immediately available for you to use. To use a package you first need to load the package by using the `library()` function. For example, to load the `remotes` package you previously installed.

```
library(remotes)
```

The `library()` function will also load any additional packages required and may print out additional package information. It is important to realise that every time you start a new R session (or restore a previously saved session) you need to load the packages you will be using. We tend to put all our `library()` statements required for our analysis near the top of our R scripts to make them easily accessible and easy to add to as our code develops. If you try to use a function without first loading the relevant R package you will receive an error message that R could not find the function. For example, if you try to use the `install_github()` function without loading the `remotes` package first you will receive the following error.

```
install_github('tidyverse/dplyr')

# Error in install_github("tidyverse/dplyr") :
#   could not find function "install_github"
```

Sometimes it can be useful to use a function without first using the `library()` function. If, for example, you will only be using one or two functions in your script and don't want to load all of the other functions in a package then you can access the function directly by specifying the package name followed by two colons and then the function name.

```
remotes::install_github('tidyverse/dplyr')
```

This is how we were able to use the `install()` and `install_github()` functions above without first loading the packages `BiocManager` and `remotes`. Most of the time we recommend using the `library()` function.

1.6   Projects in RStudio

As with most things in life, when it comes to dealing with data and data analysis things are so much simpler if you're organised. Clear project organisation makes it easier for both you (especially the future you) and your collaborators to make sense of what you've done. There's nothing more frustrating than coming back to a project months (sometimes years) later and have to spend days (or weeks) figuring out where everything is, what you did and why you did it. A well documented project that has a consistent and logical structure increases the likelihood that you can pick up where you left off with minimal fuss no matter how much time has passed. In addition, it's much easier to write code to automate tasks when files are well organised and are sensibly named. This is even more relevant nowadays as it's never been easier to collect vast amounts of data which can be saved across 1000's or even 100,000's of separate data files. Lastly, having a well organised project reduces the risk of introducing bugs or errors into your workflow and if they do occur (which inevitably they will at some point), it makes it easier to track down these errors and deal with them efficiently.

Thankfully, there are some nice features in R and RStudio that make it quite easy to manage a project. There are also a few simple steps you can take right at the start of any project to help keep things shipshape.

A great way of keeping things organised is to use RStudio Projects. An RStudio Project keeps all of your R scripts, R markdown documents, R functions and data together in one place. The nice thing about RStudio Projects is that each project has its own directory, workspace, history and source documents so different analyses that you are working on are kept completely separate from each other. This means that you can have

multiple instances of RStudio open at the same time (if that's your thing) or you can very easily switch between projects without fear of them interfering with each other.

See this video for step-by-step instructions on how to create and work with RStudio projects

To create a project, open RStudio and select `File -> New Project...` from the menu. You can create either an entirely new project, a project from an existing directory or a version controlled project (see the GitHub Chapter for further details about this). In this Chapter we will create a project in a new directory.
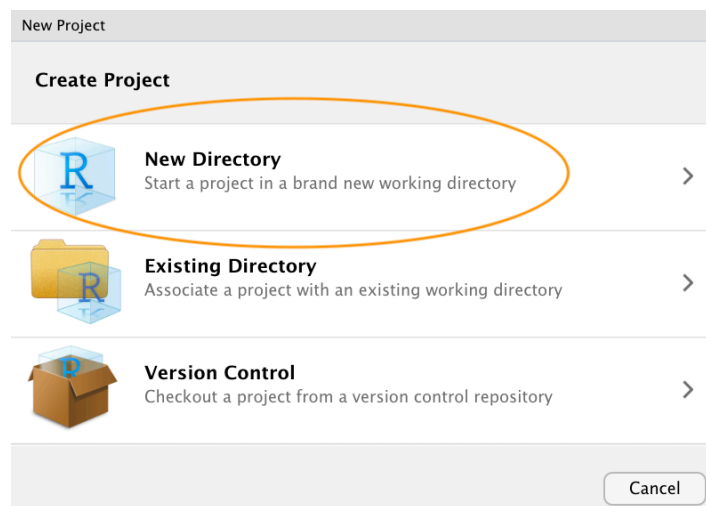
You can also create a new project by clicking on the 'Project' button in the top right of RStudio and selecting 'New Project...'
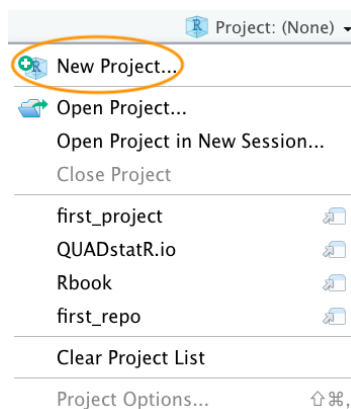
In the next window select 'New Project'.
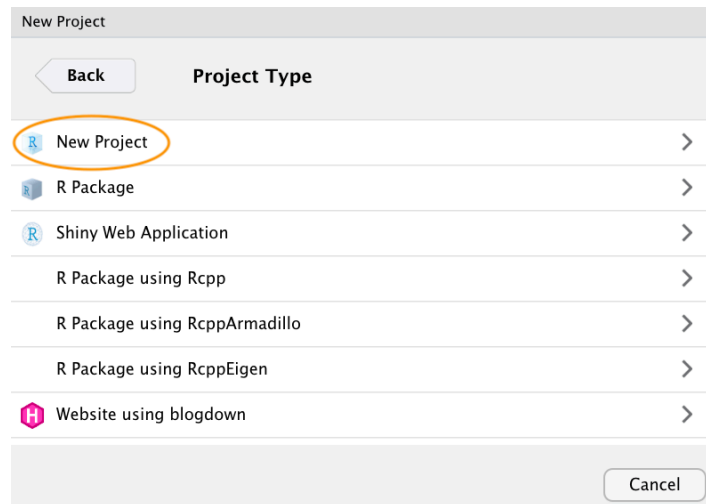
Now enter the name of the directory you want to create in the 'Directory name:' field (we'll call it `first_project` for this Chapter). If you want to change the location of the directory on your computer click the 'Browse...' button and navigate to where you would like to create the directory. We always tick the 'Open in new session' box as well. Finally, hit the 'Create Project' to create the new project.
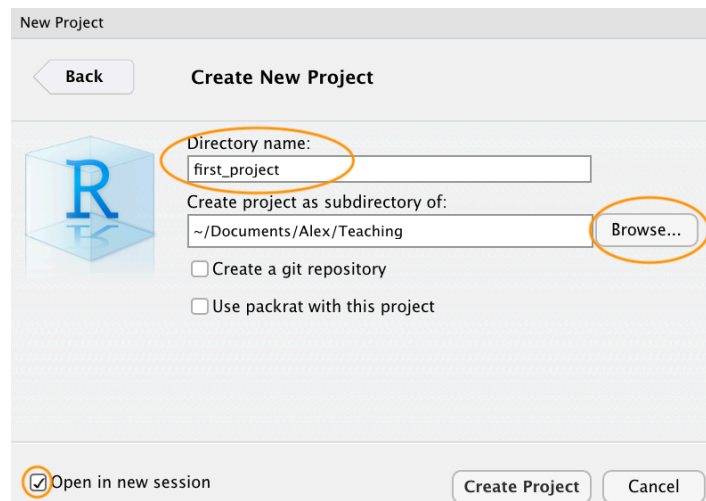
Once your new project has been created you will now have a new folder on your computer that contains an RStudio project file called `first_project.Rproj`. This `.Rproj` file contains various project options (but you shouldn't really interact with it) and can also

be used as a shortcut for opening the project directly from the file system (just double click on it). You can check this out in the 'Files' tab in RStudio (or in Finder if you're on a Mac or File Explorer in Windows).
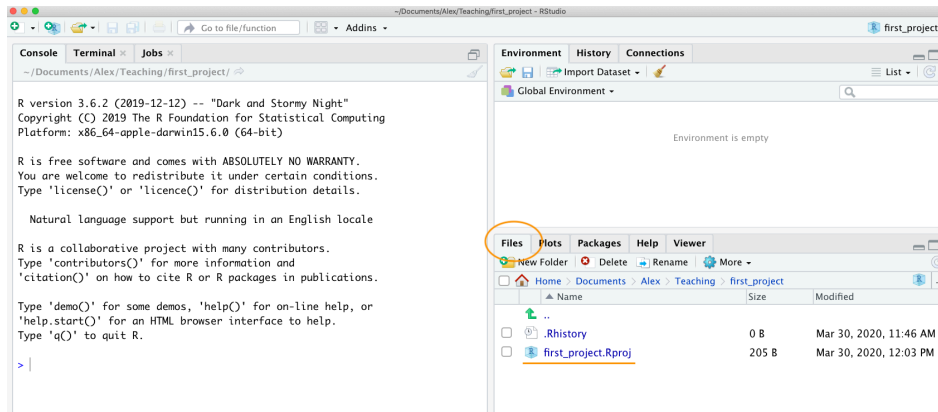
The last thing we suggest you do is select `Tools -> Project Options...` from the menu. Click on the 'General' tab on the left hand side and then change the values for 'Restore .RData into workspace at startup' and 'Save workspace to .RData on exit' from 'Default' to 'No'. This ensures that every time you open your project you start with a clean R session. You don't have to do this (many people don't) but we prefer to start with a completely clean workspace whenever we open our projects to avoid any potential conflicts with things we have done in previous sessions. The downside to this is that you will need to rerun your R code every time you open your project.
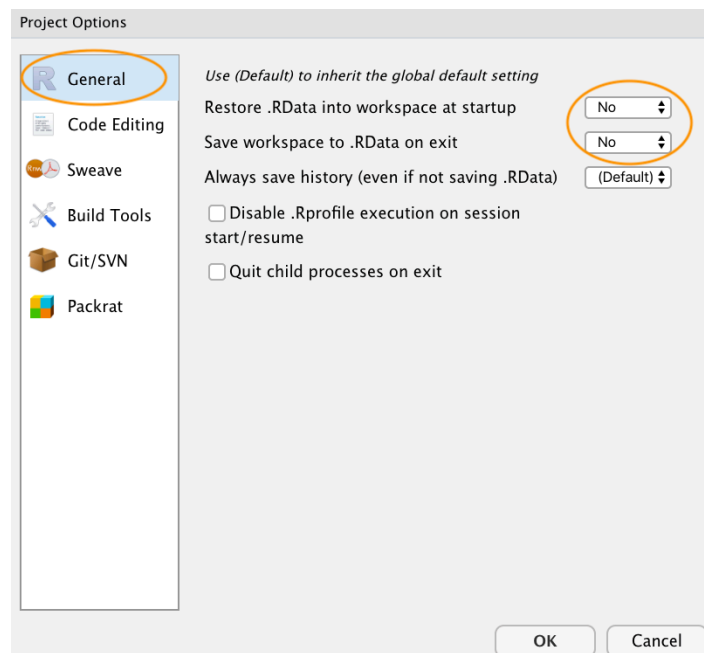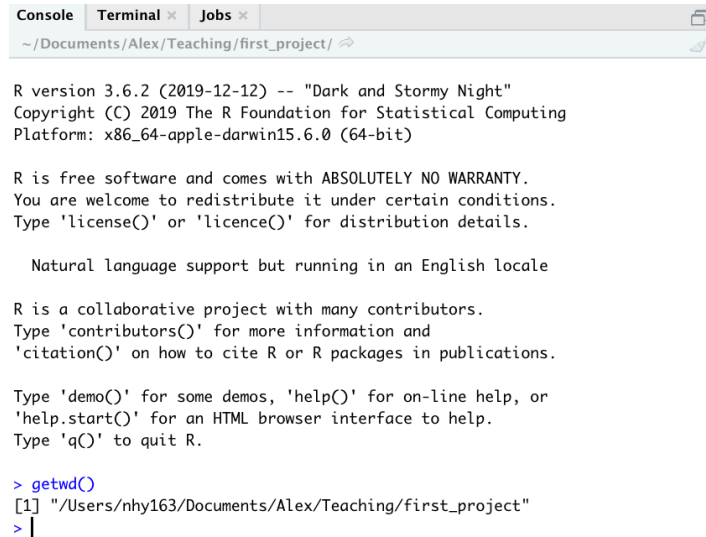
Now that you have an RStudio project set up you can start creating R scripts (or R markdown documents) or whatever you need to complete you project. All of the R scripts will now be contained within the RStudio project and saved in the project folder.

1.7   Working directories

The working directory is the default location where R will look for files you want to load and where it will put any files you save. One of the great things about using RStudio Projects is that when you open a project it will automatically set your working directory to the appropriate location. You can check the file path of your working directory by looking at bar at the top of the Console pane. Note: the ~ symbol above is shorthand for /Users/nhy163/ on a Mac computer (the same on Linux computers).
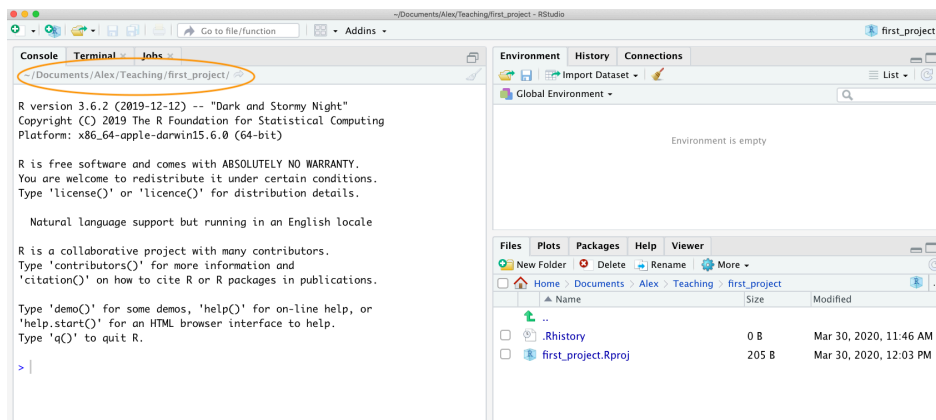
You can also use the getwd() function in the Console which returns the file path of the current working directory.

```
Console   Terminal ×   Jobs ×                                          ⏏
~/Documents/Alex/Teaching/first_project/ ⇗                              🗑

R version 3.6.2 (2019-12-12) -- "Dark and Stormy Night"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

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

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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

> getwd()
[1] "/Users/nhy163/Documents/Alex/Teaching/first_project"
> |
```

In the example above, the working directory is a folder called 'first_project' which is a subfolder of "Teaching" in the 'Alex' folder which in turn is in a 'Documents' folder located in the 'nhy163' folder which itself is in the 'Users' folder. On a Windows based computer our working directory would also include a drive letter (i.e. `C:/Users/nhy163/Documents/Alex/Teaching/first_project`).

If you weren't using an RStudio Project then you would have to set your working directory using the `setwd()` function at the start of every R script (something we did for many years).

```
setwd('/Users/nhy163/Documents/Alex/Teaching/first_project')
```

However, the problem with `setwd()` is that it uses an *absolute* file path which is specific to the computer you are working on. If you want to send your script to someone else (or if you're working on a different computer) this absolute file path is not going to work on your friend/colleagues computer as their directory configuration will be different (you are unlikely to have a directory structure `/Users/nhy163/Documents/Alex/Teaching/` on your computer). This results in a project that is not self-contained and not easily portable. RStudio solves this problem by allowing you to use *relative* file paths which are relative to the *Root* project directory. The Root project directory is just the directory that contains the `.Rproj` file (`first_project.Rproj` in our case). If you want to share your analysis with someone else, all you need to do is copy the entire project directory and send to your to your collaborator. They would then just need to open the project file and any R scripts that contain references to relative file paths will just work. For example, let's say that you've created a subdirectory called `raw_data` in your Root project directory that contains a tab delimited datafile called `mydata.txt` (we will cover directory structures below). To import this datafile in an RStudio project using the `read.table()` function (don't worry about this now, we will cover this in much more detail in Chapter 3) all you need to include in your R script is

```r
dataf <- read.table('raw_data/mydata.txt', header = TRUE,
                    sep = '\t')
```

Because the file path `raw_data/mydata.txt` is relative to the project directory it doesn't matter where you collaborator saves the project directory on their computer it will still work.

If you weren't using an RStudio project then you would have to use either of the options below neither of which would work on a different computer.

```r
setwd("/Users/nhy163/Documents/Alex/Teaching/first_project/")

dataf <- read.table("raw_data/mydata.txt", header = TRUE, sep = "\t")

# or

dataf <- read.table("/Users/nhy163/Documents/Alex/Teaching/first_project/raw_data/mydata.txt",
    header = TRUE, sep = "\t")
```

For those of you who want to take the notion of relative file paths a step further, take a look at the `here()` function in the `here` package. The `here()` function allows you to automagically build file paths for any file relative to the project root directory that are also operating system agnostic (works on a Mac, Windows or Linux machine). For example, to import our `mydata.txt` file from the `raw_data` directory just use

```r
library(here)  # you may need to install the here package first
dataf <- read.table(here("raw_data", "mydata.txt"),
                    header = TRUE, sep = '\t',
                    stringsAsFactors = TRUE)

# or without loading the here package

dataf <- read.table(here::here("raw_data", "mydata.txt"),
                    header = TRUE, sep = '\t',
                    stringsAsFactors = TRUE)
```

1.8   Directory structure

In addition to using RStudio Projects, it's also really good practice to structure your working directory in a consistent and logical way to help both you and your collaborators. We frequently use the following directory structure in our R based projects
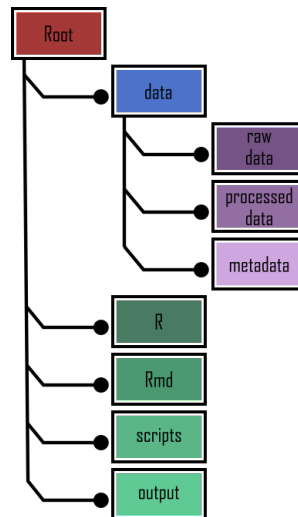
In our working directory we have the following directories:

- **Root** - This is your project directory containing your `.Rproj` file.

- **data** - We store all our data in this directory. The subdirectory called `raw_data`
  contains raw data files and only raw data files. These files should be treated
  as **read only** and should not be changed in any way. If you need to pro-
  cess/clean/modify your data do this in R (not MS Excel) as you can document
  (and justify) any changes made. Any processed data should be saved to a separate
  file and stored in the `processed_data` subdirectory. Information about data collec-
  tion methods, details of data download and any other useful metadata should be
  saved in a text document (see README text files below) in the `metadata` subdi-
  rectory.

- **R** - This is an optional directory where we save all of the custom R functions
  we've written for the current analysis. These can then be sourced into R using
  the `source()` function.

- **Rmd** - An optional directory where we save our R markdown documents.

- **scripts** - All of the main R scripts we have written for the current project are
  saved here.

- **output** - Outputs from our R scripts such as plots, HTML files and data sum-
  maries are saved in this directory. This helps us and our collaborators distinguish
  what files are outputs and which are source files.

Of course, the structure described above is just what works for us most of the time
and should be viewed as a starting point for your own needs. We tend to have a fairly
consistent directory structure across our projects as this allows us to quickly orientate
ourselves when we return to a project after a while. Having said that, different projects
will have different requirements so we happily add and remove directories as required.

You can create your directory structure using Windows Explorer (or Finder on a Mac) or within RStudio by clicking on the 'New folder' button in the 'Files' pane.
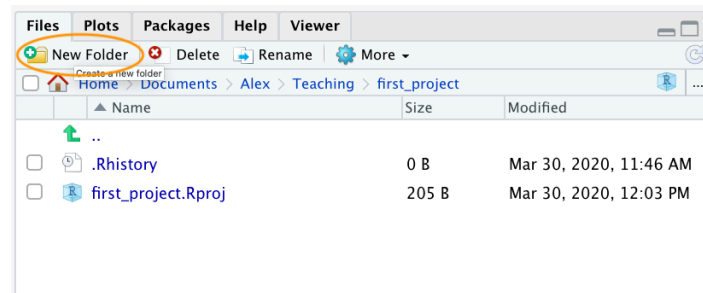
An alternative approach is to use the `dir.create()` and `list.files()` functions in the R Console.

```r
# create directory called 'data'
dir.create('data')

# create subdirectory raw_data in the data directory
dir.create('data/raw_data')

# list the files and directories
list.files(recursive = TRUE, include.dirs = TRUE)

# [1] "data"  "data/raw_data"   "first_project.Rproj"
```

1.9   File names

What you call your files matters more than you might think. Naming files is also more difficult than you think. The key requirement for a 'good' file name is that it's informative whilst also being relatively short. This is not always an easy compromise and often requires some thought. Ideally you should try to avoid the following!

Although there's not really a recognised standard approach to naming files (actually there is, just not everyone uses it), there are a couple of things to bear in mind.

- First, avoid using spaces in file names by replacing them with underscores or even hyphens. Why does this matter? One reason is that some command line software (especially many bioinformatic tools) won't recognise a file name with a space

Figure 1.1: source:https://xkcd.com/1459/
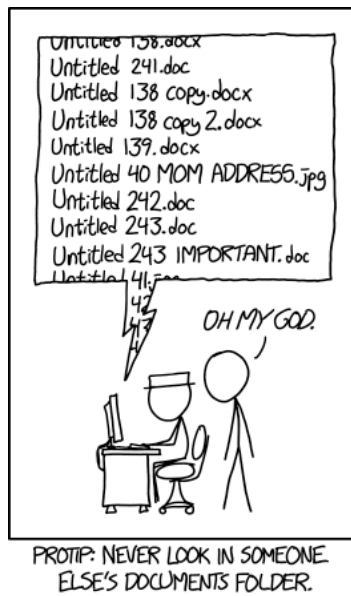
and you'll have to go through all sorts of shenanigans using escape characters to make sure spaces are handled correctly. Even if you don't think you will ever use command line software you may be doing so indirectly. Take R markdown for example, if you want to render an R markdown document to pdf using the `rmarkdown` package you will actually be using a command line LaTeX engine under the hood (called Pandoc). Another good reason not to use spaces in file names is that it makes searching for file names (or parts of file names) using regular expressions in R (or any other language) much more difficult.

- For the reasons given above, also avoid using special characters (i.e. @£$%^&*(:/) in your file names.

- If you are versioning your files with sequential numbers (i.e. file1, file2, file3 ...) and you have more than 9 files you should use 01, 02, 03 .. 10 as this will ensure the files are printed in the correct order (see what happens if you don't). If you have more than 99 files then use 001, 002, 003... etc.

- If your file names include dates, use the ISO 8601 format YYYY-MM-DD (or YYYYMMDD) to ensure your files are listed in proper chronological order.

- Never use the word *final* in any file name - it never is!

Whatever file naming convention you decide to use, try to adopt early, stick with it and be consistent. You'll thank us!

1.10 Project documentation

A quick note or two about writing R code and creating R scripts. Unless you're doing something really quick and dirty we suggest that you always write your R code as an

R script. R scripts are what make R so useful. Not only do you have a complete record of your analysis, from data manipulation, visualisation and statistical analysis, you can also share this code (and data) with friends, colleagues and importantly when you submit and publish your research to a journal. With this in mind, make sure you include in your R script all the information required to make your work reproducible (author names, dates, sampling design etc). This information could be included as a series of comments # or, even better, by mixing executable code with narrative into an R markdown document. It's also good practice to include the output of the sessionInfo() function at the end of any script which prints the R version, details of the operating system and also loaded packages. A really good alternative is to use the session_info() function from the xfun package for a more concise summary of our session environment.

Here's an example of including meta-information at the start of an R script.

```r
# Title: Time series analysis of snouters


# Purpose : This script performs a time series analyses on
#           snouter count data.
#           Data consists of counts of snouter species
#           collected from 18 islands in the Hy-yi-yi
#           archipelago between 1950 and 1957.
#           For details of snouter biology see:
#           https://en.wikipedia.org/wiki/Rhinogradentia


# Project number: #007


# DataFile:'data/snouter_pop.txt'


# Author: A. Nother
# Contact details: a.nother@uir.ac.uk


# Date script created: Mon Dec 2 16:06:44 2019 -----------
# Date script last modified: Thu Dec 12 16:07:12 2019 ----


# package dependencies
library(PopSnouter)
library(ggplot2)


print('put your lovely R code here')


# good practice to include session information


xfun::session_info()
```
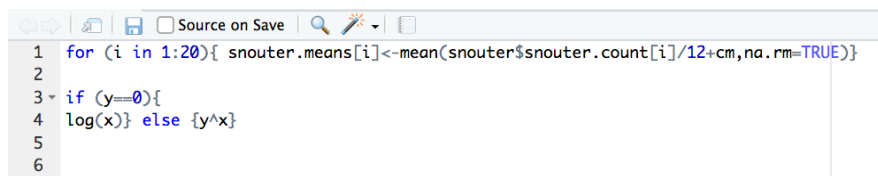
This is just one example and there are no hard and fast rules so feel free to develop a system that works for you. A really useful shortcut in RStudio is to automatically

include a time and date stamp in your R script. To do this, write `ts` where you want to insert your time stamp in your R script and then press the 'shift + tab' keys. RStudio will magically convert `ts` into the current date and time and also automatically comment out this line with a `#`. Another really useful RStudio shortcut is to comment out multiple lines in your script with a `#` symbol. To do this, highlight the lines of text you want to comment and then press 'ctrl + shift + c' (or 'cmd + shift + c' on a mac). To uncomment the lines just use 'ctrl + shift + c' again.

In addition to including metadata in your R scripts it's also common practice to create a separate text file to record important information. By convention these text files are named `README`. We often include a `README` file in the directory where we keep our raw data. In this file we include details about when data were collected (or downloaded), how data were collected, information about specialised equipment, preservation methods, type and version of any machines used (i.e. sequencing equipment) etc. You can create a README file for your project in RStudio by clicking on the `File -> New File -> Text File` menu.

1.11　R style guide

How you write your code is more or less up to you although your goal should be to make it as easy to read as possible (for you and others). Whilst there are no rules (and no code police), we encourage you to get into the habit of writing readable R code by adopting a particular style. We suggest that you follow Google's R style guide whenever possible. This style guide will help you decide where to use spaces, how to indent code and how to use square `[ ]` and curly `{ }` brackets amongst other things. If all that sounds like too much hard work you can install the `styler` package which includes an RStudio add-in to allow you to automatically restyle selected code (or entire files and projects) with the click of your mouse. You can find more information about the `styler` package including how to install here. Once installed, you can highlight the code you want to restyle, click on the 'Addins' button at the top of RStudio and select the 'Style Selection' option. Here is an example of poorly formatted R code.

```
   □  □ | □ | 🖫 □ Source on Save   🔍 🪄 ▾ | □
1  for (i in 1:20){ snouter.means[i]<-mean(snouter$snouter.count[i]/12+cm,na.rm=TRUE)}
2
3▾ if (y==0){
4  log(x)} else {y^x}
5
6
```

Now highlight the code and use the `styler` package to reformat
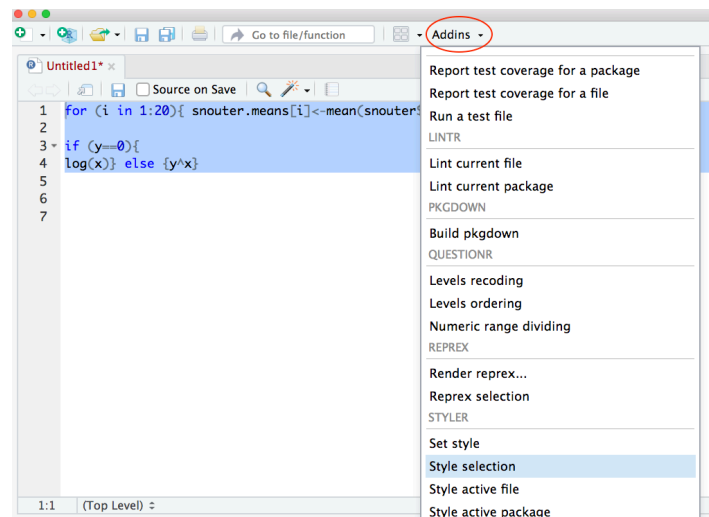
To produce some nicely formatted code
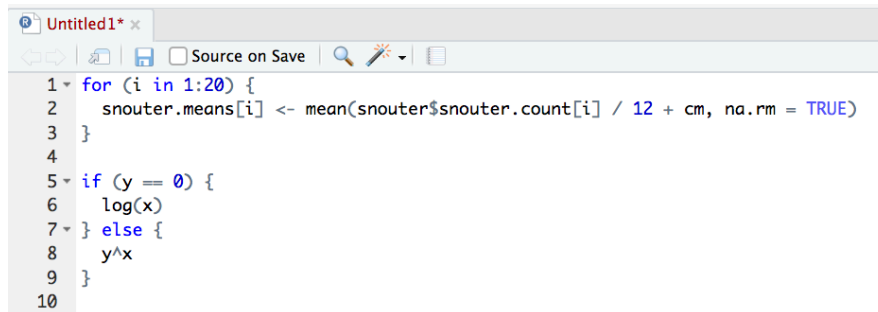
1.12   Backing up projects

Don't be that person who loses hard won (and often expensive) data and analyses. Don't be that person who thinks it'll never happen to me - it will! Always think of the absolute worst case scenario, something that makes you wake up in a cold sweat at night, and do all you can to make sure this never happens. Just to be clear, if you're relying on copying your precious files to an external hard disk or USB stick this is **NOT** an effective backup strategy. These things go wrong all the time as you lob them into your rucksack or 'bag for life' and then lug them between your office and home. Even if you do leave them plugged into your computer what happens when the building burns down (we did say worst case!)?

Ideally, your backups should be offsite and incremental. Happily there are numerous options for backing up your files. The first place to look is in your own institute. Most (all?) Universities have some form of network based storage that should be easily accessible and is also underpinned by a comprehensive disaster recovery plan. Other options include cloud based services such as Google Drive and Dropbox (to

name but a few), but make sure you're not storing sensitive data on these services and are comfortable with the often eye watering privacy policies.

Whilst these services are pretty good at storing files, they don't really help with incremental backups. Finding previous versions of files often involves spending inordinate amounts of time trawling through multiple files named *'final.doc'*, *'final_v2.doc'* and *'final_usethisone.doc'* etc until you find the one you were looking for. The best way we know for both backing up files and managing different versions of files is to use Git and GitHub. To find out more about how you can use RStudio, Git and GitHub together see the Git and GitHub Chapter.

1.13   Citing R

Many people have invested huge amounts of time and energy making R the great piece of software you're now using. If you use R in your work (and we hope you do) please remember to give appropriate credit by citing R. To get the most up to date citation for R you can use the `citation()` function.

```
citation()
## To cite R in publications use:
##
##   R Core Team (2023). _R: A Language and Environment for Statistical
##   Computing_. R Foundation for Statistical Computing, Vienna, Austria.
##   <https://www.R-project.org/>.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {R: A Language and Environment for Statistical Computing},
##     author = {{R Core Team}},
##     organization = {R Foundation for Statistical Computing},
##     address = {Vienna, Austria},
##     year = {2023},
##     url = {https://www.R-project.org/},
##   }
##
## We have invested a lot of time and effort in creating R, please cite it
## when using it for data analysis. See also 'citation("pkgname")' for
## citing R packages.
```

If you want to cite a particular package you've used for your data analysis.

```
citation(package = "here")
## To cite package 'here' in publications use:
##
##   Müller K (2020). _here: A Simpler Way to Find Your Files_. R package
##   version 1.0.1, <https://CRAN.R-project.org/package=here>.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {here: A Simpler Way to Find Your Files},
##     author = {Kirill Müller},
##     year = {2020},
##     note = {R package version 1.0.1},
##     url = {https://CRAN.R-project.org/package=here},
##   }
```

1.14   Exercise 1

Congratulations, you've reached the end of Chapter 1! Perhaps now's a good time to practice some of what you've learned. You can find an exercise we've prepared for you (and our solutions) on the course website.

Chapter 2

Some R basics

In this Chapter we'll introduce you to using R and RStudio to perform some basic R tasks such as creating objects and assigning values to objects, exploring different types of objects and how to perform some common operations on objects. We'll also learn how to get help in R and highlight some resources to help support your R learning. Finally, we'll cover how to save your work.

Before we continue, here are a few things to bear in mind as you work through this Chapter:

- R is case sensitive i.e. `A` is not the same as `a` and `anova` is not the same as `Anova`.

- Anything that follows a `#` symbol is interpreted as a comment and ignored by R. Comments should be used liberally throughout your code for both your own information and also to help your collaborators. Writing comments is a bit of an art and something that you will become more adept at as your experience grows.

- In R, commands are generally separated by a new line. You can also use a semicolon `;` to separate your commands but this is rarely used.

- If a continuation prompt `+` appears in the console after you execute your code this means that you haven't completed your code correctly. This often happens if you forget to close a bracket and is especially common when nested brackets are used (`((((some command))))`). Just finish the command on the new line and fix the typo or hit escape on your keyboard (see point below) and fix.

- In general, R is fairly tolerant of extra spaces inserted into your code, in fact using spaces is actively encouraged. However, spaces should not be inserted into operators i.e. `<-` should not read `< -` (note the space). See Google's style guide for advice on where to place spaces to make your code more readable.

- If your console 'hangs' and becomes unresponsive after running a command you can often get yourself out of trouble by pressing the escape key (esc) on your keyboard or clicking on the stop icon in the top right of your console. This will terminate most current operations.

2.1   Getting started

In Chapter 1 we learned about the R Console and creating scripts and Projects in
RStudio. We also saw how you write your R code in a script and then source this
code into the console to get it to run (if you've forgotten how to do this, pop back to
the console section to refresh your memory). Writing your code in a script means that
you'll always have a permanent record of everything you've done (provided you save
your script) and also allows you to make loads of comments to remind your future self
what you've done. So, while you're working through this Chapter we suggest that you
create a new script (or RStudio Project) to write your code as you follow along.

As we saw in the previous Chapter, at a basic level we can use R much as you would
use a calculator. We can type an arithmetic expression into our script, then source it
into the console and receive a result. For example, if we type the expression `2 + 2` and
then source this line of code we get the answer `4` (reassuringly!)

```
2 + 2
## [1] 4
```

The `[1]` in front of the result tells you that the observation number at the beginning
of the line is the first observation. This is not much help in this example, but can be
quite useful when printing results with multiple lines (we'll see an example below). The
other obvious arithmetic operators are `-`, `*`, `/` for subtraction, multiplication and division
respectively. R follows the usual mathematical convention of order of operations. For
example, the expression `2 + 3 * 4` is interpreted to have the value `2 + (3 * 4) = 14`, not
`(2 + 3) * 4 = 20`. There are a huge range of mathematical functions in R, some of the
most useful include; `log()`, `log10()`, `exp()`, `sqrt()`.

```
log(1)               # logarithm to base e
## [1] 0
log10(1)             # logarithm to base 10
## [1] 0
exp(1)               # natural antilog
## [1] 2.718282
sqrt(4)              # square root
## [1] 2
4^2                    # 4 to the power of 2
## [1] 16
pi                     # not a function but useful
## [1] 3.141593
```

It's important to realise that when you run code as we've done above, the result of the
code (or **value**) is only displayed in the console. Whilst this can sometimes be useful
it is usually much more practical to store the value(s) in a object.

2.2 Objects in R

At the heart of almost everything you will do (or ever likely to do) in R is the concept that everything in R is an object. These objects can be almost anything, from a single number or character string (like a word) to highly complex structures like the output of a plot, a summary of your statistical analysis or a set of R commands that perform a specific task. Understanding how you create objects and assign values to objects is key to understanding R.

2.2.1 Creating objects

See this video for an introduction to creating and managing objects in R

To create an object we simply give the object a name. We can then assign a value to this object using the *assignment operator* `<-` (sometimes called the *gets operator*). The assignment operator is a composite symbol comprised of a 'less than' symbol `<` and a hyphen `-` .

```
my_obj <- 48
```

In the code above, we created an object called `my_obj` and assigned it a value of the number `48` using the assignment operator (in our head we always read this as '*my_obj gets 48*'). You can also use `=` instead of `<-` to assign values but this is considered bad practice and we would discourage you from using this notation.

To view the value of the object you simply type the name of the object.

```
my_obj
## [1] 48
```

Now that we've created this object, R knows all about it and will keep track of it during this current R session. All of the objects you create will be stored in the current workspace and you can view all the objects in your workspace in RStudio by clicking on the 'Environment' tab in the top right hand pane.
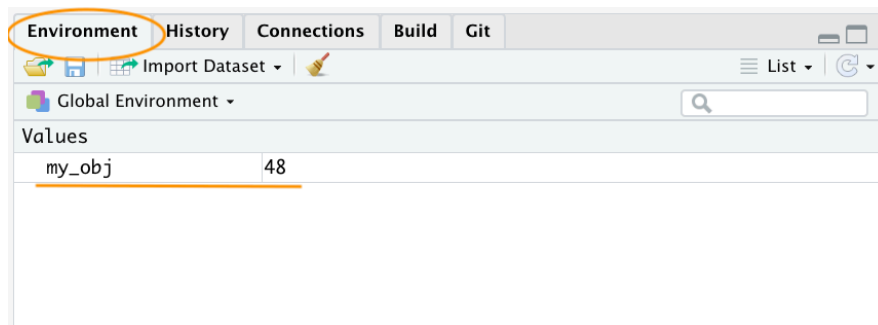
If you click on the down arrow on the 'List' icon in the same pane and change to 'Grid' view RStudio will show you a summary of the objects including the type (numeric - it's a number), the length (only one value in this object), its 'physical' size and its value (48 in this case).
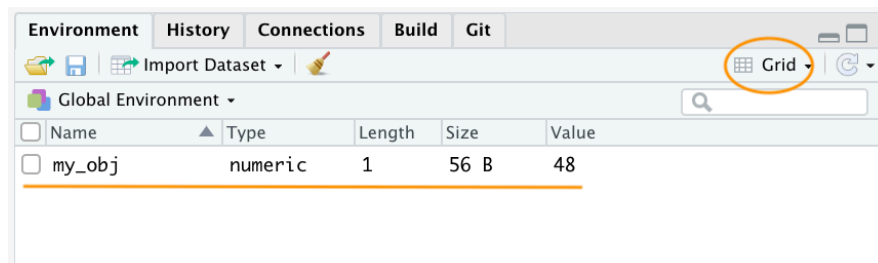

There are many different types of values that you can assign to an object. For example

```
my_obj2 <- "R is cool"
```

Here we have created an object called `my_obj2` and assigned it a value of `R is cool` which is a character string. Notice that we have enclosed the string in quotes. If you forget to use the quotes you will receive an error message.

```
my_obj2 <- R is cool
Error: unexpected symbol in "my_obj2 <- R is"
```

Our workspace now contains both objects we've created so far with `my_obj2` listed as type character.

To change the value of an existing object we simply reassign a new value to it. For example, to change the value of `my_obj2` from `"R is cool"` to the number `1024`.

```
my_obj2 <- 1024
```

Notice that the Type has changed to numeric and the value has changed to 1024 in the environment.

Once we have created a few objects, we can do stuff with our objects. For example, the following code creates a new object `my_obj3` and assigns it the value of `my_obj` added to `my_obj2` which is 1072 ($48 + 1024 = 1072$).

```
my_obj3 <- my_obj + my_obj2
my_obj3
```

```
## [1] 1072
```

Notice that to display the value of `my_obj3` we also need to write the object's name. The above code works because the values of both `my_obj` and `my_obj2` are numeric (i.e. a number). If you try to do this with objects with character values (**character class**) you will receive an error.

```
char_obj <- "hello"
char_obj2 <- "world!"
char_obj3 <- char_obj + char_obj2
Error in char_obj+char_obj2:non-numeric argument to binary operator
```

The error message is essentially telling you that either one or both of the objects `char_obj` and `char_obj2` is not a number and therefore cannot be added together.

When you first start learning R, dealing with errors and warnings can be frustrating as they're often difficult to understand (what's an *argument*? what's a *binary operator*?). One way to find out more information about a particular error is to Google a generalised version of the error message. For the above error try Googling *'non-numeric argument to binary operator error + r'* or even *'common r error messages'*.

Another error message that you'll get quite a lot when you first start using R is `Error: object 'XXX' not found`. As an example, take a look at the code below.

```
my_obj <- 48
my_obj4 <- my_obj + no_obj
Error: object 'no_obj' not found
```

R returns an error message because we haven't created (defined) the object `no_obj` yet. Another clue that there's a problem with this code is that, if you check your environment, you'll see that object `my_obj4` has not been created.

2.2.2   Naming objects

Naming your objects is one of the most difficult things you will do in R (honestly - we're serious). Ideally your object names should be kept both short and informative which is not always easy. If you need to create objects with multiple words in their name then use either an underscore or a dot between words or capitalise the different words. We prefer the underscore format (called *snake case*).

```
output_summary <- "my analysis"
output.summary <- "my analysis"
outputSummary <- "my analysis"
```

There are also a few limitations when it come to giving objects names. An object name cannot start with a number or a dot followed by a number (i.e. `2my_variable` or `.2my_variable`). You should also avoid using non-alphanumeric characters in your object names (i.e. &, ^, /, ! etc). In addition, make sure you don't name your objects with reserved words (i.e. `TRUE`, `NA`) and it's never a good idea to give your object the same name as a built-in function. One that crops up more times than we can remember is

```
data <- read.table("mydatafile", header = TRUE) #data is a
                                                # function!
```

2.3 Using functions in R

Up until now we've been creating simple objects by directly assigning a single value to an object. It's very likely that you'll soon want to progress to creating more complicated objects as your R experience grows and the complexity of your tasks increase. Happily, R has a multitude of functions to help you do this. You can think of a function as an object which contains a series of instructions to perform a specific task. The base installation of R comes with many functions already defined or you can increase the power of R by installing one of the 10000's of packages now available. Once you get a bit more experience with using R you may want to define your own functions to perform tasks that are specific to your goals (more about this in Chapter 7).

See this video for a general introduction to using functions in R and this video on how to create vectors in R

The first function we will learn about is the `c()` function. The `c()` function is short for concatenate and we use it to join together a series of values and store them in a data structure called a **vector** (more on vectors in Chapter 3).

```
my_vec <- c(2, 3, 1, 6, 4, 3, 3, 7)
```

In the code above we've created an object called `my_vec` and assigned it a value using the function `c()`. There are a couple of really important points to note here. Firstly, when you use a function in R, the function name is **always** followed by a pair of round brackets even if there's nothing contained between the brackets. Secondly, the argument(s) of a function are placed inside the round brackets and are separated by commas. You can think of an argument as way of customising the use or behaviour of a function. In the example above, the arguments are the numbers we want to concatenate. Finally, one of the tricky things when you first start using R is to know which function to use for a particular task and how to use it. Thankfully each function will always have a help document associated with it which will explain how to use the function (more on this later) and a quick Google search will also usually help you out.

To examine the value of our new object we can simply type out the name of the object as we did before.

```
my_vec
```

```
## [1] 2 3 1 6 4 3 3 7
```

Now that we've created a vector we can use other functions to do useful stuff with this object. For example, we can calculate the mean, variance, standard deviation and number of elements in our vector by using the `mean()`, `var()`, `sd()` and `length()` functions.

```
mean(my_vec)     # returns the mean of my_vec
## [1] 3.625
var(my_vec)      # returns the variance of my_vec
## [1] 3.982143
sd(my_vec)       # returns the standard deviation of my_vec
## [1] 1.995531
length(my_vec)  # returns the number of elements in my_vec
## [1] 8
```

If we wanted to use any of these values later on in our analysis we can just assign the resulting value to another object.

```
vec_mean <- mean(my_vec)     # returns the mean of my_vec
vec_mean
## [1] 3.625
```

Sometimes it can be useful to create a vector that contains a regular sequence of values in steps of one. Here we can make use of a shortcut using the : symbol.

```
my_seq <- 1:10      # create regular sequence
my_seq
##  [1]  1  2  3  4  5  6  7  8  9 10
my_seq2 <- 10:1    # in decending order
my_seq2
##  [1] 10  9  8  7  6  5  4  3  2  1
```

Other useful functions for generating vectors of sequences include the `seq()` and `rep()` functions. For example, to generate a sequence from 1 to 5 in steps of 0.5.

```
my_seq2 <- seq(from = 1, to = 5, by = 0.5)
my_seq2
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Here we've used the arguments `from =` and `to =` to define the limits of the sequence and the `by =` argument to specify the increment of the sequence. Play around with other values for these arguments to see their effect.

The `rep()` function allows you to replicate (repeat) values a specified number of times. To repeat the value 2, 10 times

```
my_seq3 <- rep(2, times = 10)   # repeats 2, 10 times
my_seq3
## [1] 2 2 2 2 2 2 2 2 2 2
```

You can also repeat non-numeric values

```
my_seq4 <- rep("abc", times = 3)    # repeats 'abc' 3 times
my_seq4
## [1] "abc" "abc" "abc"
```

or each element of a series

```
my_seq5 <- rep(1:5, times = 3)  # repeats the series 1 to
                                # 5, 3 times
my_seq5
## [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

or elements of a series.

```
my_seq6 <- rep(1:5, each = 3)   # repeats each element of the
                                #series 3 times
my_seq6
## [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

We can also repeat a non-sequential series.

```
my_seq7 <- rep(c(3, 1, 10, 7), each = 3) # repeats each
                                         # element of the
                                         # series 3 times
my_seq7
## [1]  3  3  3  1  1  1 10 10 10  7  7  7
```

Note in the code above how we've used the `c()` function inside the `rep()` function. Nesting functions allows us to build quite complex commands within a single line of code and is a very common practice when using R. However, care needs to be taken as too many nested functions can make your code quite difficult for others to understand (or yourself some time in the future!). We could rewrite the code above to explicitly separate the two different steps to generate our vector. Either approach will give the same result, you just need to use your own judgement as to which is more readable.

```
in_vec <- c(3, 1, 10, 7)
my_seq7 <- rep(in_vec, each = 3)     # repeats each element of
                                     # the series 3 times
my_seq7
## [1]  3  3  3  1  1  1 10 10 10  7  7  7
```

2.4   Working with vectors

Manipulating, summarising and sorting data using R is an important skill to master but one which many people find a little confusing at first. We'll go through a few simple examples here using vectors to illustrate some important concepts but will build on this in much more detail in Chapter 3 where we will look at more complicated (and useful) data structures.

> Take a look at this video for a quick introduction to working with vectors in R using positional and logical indexes

2.4.1   Extracting elements

To extract (also known as indexing or subscripting) one or more values (more generally known as elements) from a vector we use the square bracket `[ ]` notation. The general approach is to name the object you wish to extract from, then a set of square brackets with an index of the element you wish to extract contained within the square brackets. This index can be a position or the result of a logical test.

Positional index

To extract elements based on their position we simply write the position inside the `[ ]`. For example, to extract the 3rd value of `my_vec`

```
my_vec          # remind ourselves what my_vec looks like
## [1] 2 3 1 6 4 3 3 7
my_vec[3]       # extract the 3rd value
## [1] 1

# if you want to store this value in another object
val_3 <- my_vec[3]
val_3
## [1] 1
```

Note that the positional index starts at 1 rather than 0 like some other programming languages (i.e. Python).

We can also extract more than one value by using the `c()` function inside the square brackets. Here we extract the 1st, 5th, 6th and 8th element from the `my_vec` object

```
my_vec[c(1, 5, 6, 8)]
## [1] 2 4 3 7
```

Or we can extract a range of values using the `:` notation. To extract the values from the 3rd to the 8th elements.

```
my_vec[3:8]
## [1] 1 6 4 3 3 7
```

Logical index

Another really useful way to extract data from a vector is to use a logical expression as an index. For example, to extract all elements with a value greater than 4 in the vector `my_vec`

```
my_vec[my_vec > 4]
## [1] 6 7
```

Here, the logical expression is `my_vec > 4` and R will only extract those elements that satisfy this logical condition. So how does this actually work? If we look at the output of just the logical expression without the square brackets you can see that R returns a vector containing either `TRUE` or `FALSE` which correspond to whether the logical condition is satisfied for each element. In this case only the 4th and 8th elements return a `TRUE` as their value is greater than 4.

```
my_vec > 4
## [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

So what R is actually doing under the hood is equivalent to

```
my_vec[c(FALSE, FALSE, FALSE, TRUE, FALSE, FALSE, FALSE, TRUE)]
## [1] 6 7
```

and only those element that are TRUE will be extracted.

In addition to the < and > operators you can also use composite operators to increase the complexity of your expressions. For example the expression for 'greater or equal to' is >=. To test whether a value is equal to a value we need to use a double equals symbol == and for 'not equal to' we use != (the ! symbol means 'not').

```
my_vec[my_vec >= 4]        # values greater or equal to 4
## [1] 6 4 7
my_vec[my_vec < 4]         # values less than 4
## [1] 2 3 1 3 3
my_vec[my_vec <= 4]        # values less than or equal to 4
## [1] 2 3 1 4 3 3
my_vec[my_vec == 4]        # values equal to 4
## [1] 4
my_vec[my_vec != 4]        # values not equal to 4
## [1] 2 3 1 6 3 3 7
```

We can also combine multiple logical expressions using Boolean expressions. In R the & symbol means AND and the | symbol means OR. For example, to extract values in my_vec which are less than 6 AND greater than 2

```
val26 <- my_vec[my_vec < 6 & my_vec > 2]
val26
## [1] 3 4 3 3
```

or extract values in my_vec that are greater than 6 OR less than 3.

```
val63 <- my_vec[my_vec > 6 | my_vec < 3]
val63
## [1] 2 1 7
```

2.4.2 Replacing elements

We can change the values of some elements in a vector using our `[ ]` notation in combination with the assignment operator `<-`. For example, to replace the 4th value of our `my_vec` object from `6` to `500`

```
my_vec[4] <- 500
my_vec
## [1]   2   3   1 500   4   3   3   7
```

We can also replace more than one value or even replace values based on a logical expression.

```
# replace the 6th and 7th element with 100
my_vec[c(6, 7)] <- 100
my_vec
## [1]   2   3   1 500   4 100 100   7

# replace element that are less than or equal to 4 with 1000
my_vec[my_vec <= 4] <- 1000
my_vec
## [1] 1000 1000 1000  500 1000  100  100    7
```

2.4.3 Ordering elements

In addition to extracting particular elements from a vector we can also order the values contained in a vector. To sort the values from lowest to highest value we can use the `sort()` function.

```
vec_sort <- sort(my_vec)
vec_sort
## [1]    7  100  100  500 1000 1000 1000 1000
```

To reverse the sort, from highest to lowest, we can either include the `decreasing = TRUE` argument when using the `sort()` function

```
vec_sort2 <- sort(my_vec, decreasing = TRUE)
vec_sort2
## [1] 1000 1000 1000 1000  500  100  100    7
```

or first sort the vector using the `sort()` function and then reverse the sorted vector using the `rev()` function. This is another example of nesting one function inside another function.

```r
vec_sort3 <- rev(sort(my_vec))
vec_sort3
## [1] 1000 1000 1000 1000  500  100  100    7
```

Whilst sorting a single vector is fun, perhaps a more useful task would be to sort one vector according to the values of another vector. To do this we should use the `order()` function in combination with `[ ]`. To demonstrate this let's create a vector called `height` containing the height of 5 different people and another vector called `p.names` containing the names of these people (so Joanna is 180 cm, Charlotte is 155 cm etc).

```r
height <- c(180, 155, 160, 167, 181)
height
## [1] 180 155 160 167 181

p.names <- c("Joanna", "Charlotte", "Helen", "Karen", "Amy")
p.names
## [1] "Joanna"    "Charlotte" "Helen"     "Karen"     "Amy"
```

Our goal is to order the people in `p.names` in ascending order of their `height`. The first thing we'll do is use the `order()` function with the `height` variable to create a vector called `height_ord`.

```r
height_ord <- order(height)
height_ord
## [1] 2 3 4 1 5
```

OK, what's going on here? The first value, `2`, (remember ignore `[1]`) should be read as 'the smallest value of `height` is the second element of the `height` vector'. If we check this by looking at the `height` vector above, you can see that element 2 has a value of 155, which is the smallest value. The second smallest value in `height` is the 3rd element of `height`, which when we check is 160 and so on. The largest value of `height` is element `5` which is 181. Now that we have a vector of the positional indices of heights in ascending order (`height_ord`), we can extract these values from our `p.names` vector in this order.

```r
names_ord <- p.names[height_ord]
names_ord
## [1] "Charlotte" "Helen"     "Karen"     "Joanna"    "Amy"
```

You're probably thinking 'what's the use of this?' Well, imagine you have a dataset which contains two columns of data and you want to sort each column. If you just use `sort()` to sort each column separately, the values of each column will become uncoupled from each other. By using the `order()` on one column, a vector of positional indices is created of the values of the column in ascending order. This vector can be used on the second column, as the index of elements which will return a vector of values based on the first column.

2.4.4 Vectorisation

One of the great things about R functions is that most of them are vectorised. This means that the function will operate on all elements of a vector without needing to apply the function on each element separately. For example, to multiple each element of a vector by 5 we can simply use

```r
# create a vector
my_vec2 <- c(3, 5, 7, 1, 9, 20)

# multiply each element by 5
my_vec2 * 5
## [1]  15  25  35   5  45 100
```

Or we can add the elements of two or more vectors

```r
# create a second vector
my_vec3 <- c(17, 15, 13, 19, 11, 0)

# add both vectors
my_vec2 + my_vec3
## [1] 20 20 20 20 20 20

# multiply both vectors
my_vec2 * my_vec3
## [1] 51 75 91 19 99  0
```

However, you must be careful when using vectorisation with vectors of different lengths as R will quietly recycle the elements in the shorter vector rather than throw a wobbly (error).

```r
# create a third vector
my_vec4 <- c(1, 2)
```

```
# add both vectors - quiet recycling!
my_vec2 + my_vec4
## [1]  4  7  8  3 10 22
```

2.4.5   Missing data

In R, missing data is usually represented by an NA symbol meaning 'Not Available'. Data may be missing for a whole bunch of reasons, maybe your machine broke down, maybe you broke down, maybe the weather was too bad to collect data on a particular day etc etc. Missing data can be a pain in the proverbial both from an R perspective and also a statistical perspective. From an R perspective missing data can be problematic as different functions deal with missing data in different ways. For example, let's say we collected air temperature readings over 10 days, but our thermometer broke on day 2 and again on day 9 so we have no data for those days.

```
temp  <- c(7.2, NA, 7.1, 6.9, 6.5, 5.8, 5.8, 5.5, NA, 5.5)
temp
##  [1] 7.2  NA 7.1 6.9 6.5 5.8 5.8 5.5  NA 5.5
```

We now want to calculate the mean temperature over these days using the mean() function.

```
mean_temp <- mean(temp)
mean_temp
## [1] NA
```

Flippin heck, what's happened here? Why does the mean() function return an NA? Actually, R is doing something very sensible (at least in our opinion!). If a vector has a missing value then the only possible value to return when calculating a mean is NA. R doesn't know that you perhaps want to ignore the NA values (R can't read your mind - yet!). Happily, if we look at the help file (use help("mean") - see the next section for more details) associated with the mean() function we can see there is an argument na.rm = which is set to FALSE by default.

> na.rm - a logical value indicating whether NA values should be stripped before the computation proceeds.

If we change this argument to na.rm = TRUE when we use the mean() function this will allow us to ignore the NA values when calculating the mean.

```
mean_temp <- mean(temp, na.rm = TRUE)
mean_temp
## [1] 6.2875
```

It's important to note that the `NA` values have not been removed from our `temp` object (that would be bad practice), rather the `mean()` function has just ignored them. The point of the above is to highlight how we can change the default behaviour of a function using an appropriate argument. The problem is that not all functions will have an `na.rm =` argument, they might deal with `NA` values differently. However, the good news is that every help file associated with any function will **always** tell you how missing data are handled by default.

2.5   Getting help

This book is intended as a relatively brief introduction to R and as such you will soon be using functions and packages that go beyond this scope of this introductory text. Fortunately, one of the strengths of R is its comprehensive and easily accessible help system and wealth of online resources where you can obtain further information.
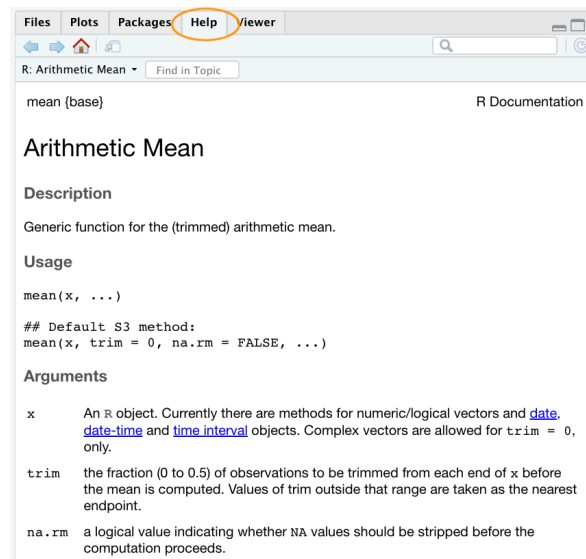
2.5.1   R help

To access R's built-in help facility to get information on any function simply use the `help()` function. For example, to open the help page for our friend the `mean()` function.

```
help("mean")
```

or you can use the equivalent shortcut.

```
?mean
```

After you run the code, the help page is displayed in the 'Help' tab in the Files pane (usually in the bottom right of RStudio).

Admittedly the help files can seem anything but helpful when you first start using R. This is probably because they're written in a very concise manner and the language used is often quite technical and full of jargon. Having said that, you do get used to this and will over time even come to appreciate a certain beauty in their brevity (honest!). One of the great things about the help files is that they all have a very similar structure regardless of the function. This makes it easy to navigate through the file to find exactly what you need.

The first line of the help document contains information such as the name of the function and the package where the function can be found. There are also other headings that provide more specific information such as

- **Description:** gives a brief description of the function and what it does.

- **Usage:** gives the name of the arguments associated with the function and possible default values.

- **Arguments:** provides more detail regarding each argument and what they do.

- **Details:** gives further details of the function if required.

- **Value:** if applicable, gives the type and structure of the object returned by the function or the operator.

- **See Also:** provides information on other help pages with similar or related content.

- **Examples:** gives some examples of using the function. These are really helpful, all you need to do is copy and paste them into the console to see what happens. You can also access examples at any time by using the `example()` function (i.e. `example("mean")`)

The `help()` function is useful if you know the name of the function. If you're not sure of the name, but can remember a key word then you can search R's help system using the `help.search()` function.

```r
help.search("mean")
```

Or you can use the equivalent shortcut.

```r
??mean
```

The results of the search will be displayed in RStudio under the 'Help' tab as before. The `help.search()` function searches through the help documentation, code demonstrations and package vignettes and displays the results as clickable links for further exploration.
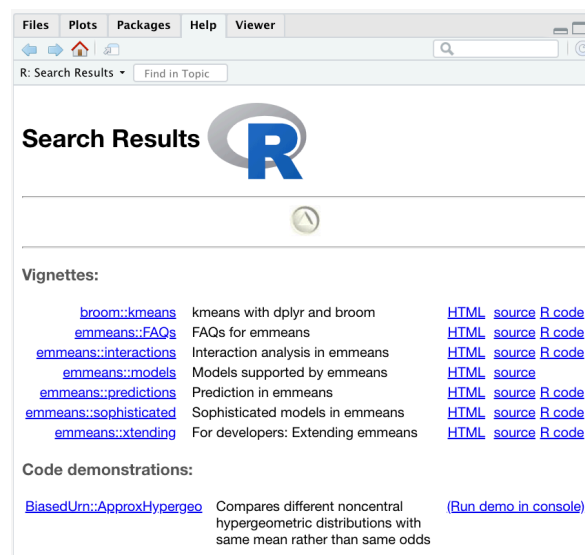
Another useful function is `apropos()`. This function can be used to list all functions containing a specified character string. For example, to find all functions with `mean` in their name

```r
apropos("mean")
##  [1] ".colMeans"     ".rowMeans"     "colMeans"       "kmeans"
##  [5] "mean"          "mean_temp"     "mean.Date"      "mean.default"
##  [9] "mean.difftime" "mean.POSIXct"  "mean.POSIXlt"   "rowMeans"
## [13] "vec_mean"      "weighted.mean"
```

You can then bring up the help file for the relevant function.

```
help("kmeans")
```

An extremely useful function is `RSiteSearch()` which enables you to search for keywords and phrases in function help pages and vignettes for all CRAN packages, and in CRAN task views. This function allows you to access the https://www.r-project.org/search.html search engine directly from the Console with the results displayed in your web browser.

```
RSiteSearch("regression")
```

2.5.2   Other sources of help

There really has never been a better time to start learning R. There are a plethora of freely available online resources ranging from whole courses to subject specific tutorials and mailing lists. There are also plenty of paid for options if that's your thing but unless you've money to burn there really is no need to part with your hard earned cash. Some resources we have found helpful are listed below.

General R resources

- R-Project: User contributed documentation
- The R Journal: Journal of the R project for statistical computing
- Swirl: An R package that teaches you R from within R
- RStudio's printable cheatsheets
- Rseek: A custom Google search for R-related sites

Getting help

- Google it!: Try Googling any error messages you get. It's not cheating and everyone does it! You'll be surprised how many other people have probably had the same problem and solved it.

- Stack Overflow: There are many thousands of questions relevant to R on Stack Overflow. Here are the most popular ones, ranked by vote. Make sure you search for similar questions before asking your own, and make sure you include a reproducible example to get the most useful advice. A reproducible example is a minimal example that lets others who are trying to help you to see the error themselves.

R markdown resources

- Basic markdown and R markdown reference
- A good markdown reference
- A good 10-minute markdown tutorial
- RStudio's R markdown cheatsheet
- R markdown reference sheet
- The R markdown documentation including a getting started guide, a gallery of demos, and several articles for more advanced usage.
- The knitr website has lots of useful reference material about how knitr works.

Git and GitHub resources

- Happy Git: Great resource for using Git and GitHub
- Version control with RStudio: RStudio document for using version control
- Using Git from RStudio: Good 10 minute guide
- The R Class: In depth guide to using Git and GitHub with RStudio

R programming

- R Programming for Data Science: In depth guide to R programming
- R for Data Science: Fantastic book, tidyverse orientated

2.6   Saving stuff in R

Your approach to saving work in R and RStudio depends on what you want to save. Most of the time the only thing you will need to save is the R code in your script(s). Remember your script is a reproducible record of everything you've done so all you need to do is open up your script in a new RStudio session and source it into the R Console and you're back to where you left off.

Unless you've followed our suggestion about changing the default settings for RStudio Projects you will be asked whether you want to save your workspace image every time you exit RStudio. We suggest that 99.9% of the time that you don't want to do this. By starting with a clean RStudio session each time we come back to our analysis we can be sure to avoid any potential conflicts with things we've done in previous sessions.

There are, however, some occasions when saving objects you've created in R is useful. For example, let's say you're creating an object that takes hours (even days) of computational time to generate. It would be extremely inconvenient to have to wait all this time each time you come back to your analysis (although we would suggest exporting this to an external file is a better solution). In this case we can save this object as an external `.RData` file which we can load back into RStudio the next time we want to use it. To save an object to an `.RData` file you can use the `save()` function (notice we don't need to use the assignment operator here)

```r
save(nameOfObject, file = "name_of_file.RData")
```

or if you want to save all of the objects in your workspace into a single .RData file use
the save.image() function.

```r
save.image(file = "name_of_file.RData")
```

To load your .RData file back into RStudio use the load() function.

```r
load(file = "name_of_file.RData")
```

2.7   Exercise 2

Congratulations, you've reached the end of Chapter 2! Perhaps now's a good
time to practice some of what you've learned. You can find an exercise we've
prepared for you (and our solutions) on the course website.

Chapter 3

Data in R

Until now, you've created fairly simple data in R and stored it as a vector. However, most (if not all) of you will have much more complicated datasets from your various experiments and surveys that go well beyond what a vector can handle. Learning how R deals with different types of data and data structures, how to import your data into R and how to manipulate and summarise your data are some of the most important skills you will need to master.

In this Chapter we'll go over the main data types in R and focus on some of the most common data structures. We will also cover how to import data into R from an external file, how to manipulate (wrangle) and summarise data and finally how to export data from R to an external file.

3.1 Data types

Understanding the different types of data and how R deals with these data is important. The temptation is to glaze over and skip these technical details, but beware, this can come back to bite you somewhere unpleasant if you don't pay attention. We've already seen an example of this when we tried (and failed) to add two character objects together using the + operator.

R has six basic types of data; numeric, integer, logical, complex and character. The keen eyed among you will notice we've only listed five data types here, the final data type is raw which we won't cover as it's not useful 99.99% of the time. We also won't cover complex numbers as we don't have the imagination!

- **Numeric** data are numbers that contain a decimal. Actually they can also be whole numbers but we'll gloss over that.

- **Integers** are whole numbers (those numbers without a decimal point).

- **Logical** data take on the value of either TRUE or FALSE. There's also another special type of logical called NA to represent missing values.

- **Character** data are used to represent string values. You can think of character strings as something like a word (or multiple words). A special type of character string is a *factor*, which is a string but with additional attributes (like levels or an order). We'll cover factors later.

R is (usually) able to automatically distinguish between different classes of data by their nature and the context in which they're used although you should bear in mind that R can't actually read your mind and you may have to explicitly tell R how you want to treat a data type. You can find out the type (or class) of any object using the `class()` function.

```
num <- 2.2
class(num)
## [1] "numeric"

char <- "hello"
class(char)
## [1] "character"

logi <- TRUE
class(logi)
## [1] "logical"
```

Alternatively, you can ask if an object is a specific class using using a logical test. The `is.[classOfData]()` family of functions will return either a `TRUE` or a `FALSE`.

```
is.numeric(num)
## [1] TRUE

is.character(num)
## [1] FALSE

is.character(char)
## [1] TRUE

is.logical(logi)
## [1] TRUE
```

It can sometimes be useful to be able to change the class of a variable using the `as.[className]()` family of coercion functions, although you need to be careful when doing this as you might receive some unexpected results (see what happens below when we try to convert a character string to a numeric).

```r
# coerce numeric to character
class(num)
## [1] "numeric"
num_char <-  as.character(num)
num_char
## [1] "2.2"
class(num_char)
## [1] "character"

# coerce character to numeric!
class(char)
## [1] "character"
char_num <- as.numeric(char)
## Warning: NAs introduced by coercion
```

Here's a summary table of some of the logical test and coercion functions available to you.

Type	Logical test	Coercing
Character	is.character	as.character
Numeric	is.numeric	as.numeric
Logical	is.logical	as.logical
Factor	is.factor	as.factor
Complex	is.complex	as.complex

3.2 Data structures

Now that you've been introduced to some of the most important classes of data in R, let's have a look at some of main structures that we have for storing these data.

3.2.1 Scalars and vectors

Perhaps the simplest type of data structure is the vector. You've already been introduced to vectors in Chapter 2 although some of the vectors you created only contained a single value. Vectors that have a single value (length 1) are called scalars. Vectors can contain numbers, characters, factors or logicals, but the key thing to remember is that all the elements inside a vector must be of the same class. In other words, vectors can contain either numbers, characters or logicals but not mixtures of these types of data. There is one important exception to this, you can include NA (remember this is special type of logical) to denote missing data in vectors with other data types.

scalar   vector
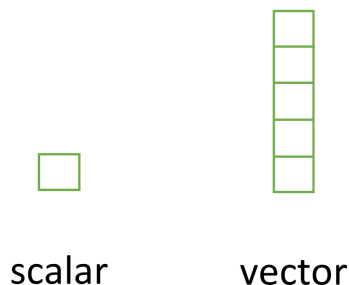
3.2.2 Matrices and arrays

Another useful data structure used in many disciplines such as population ecology, theoretical and applied statistics is the matrix. A matrix is simply a vector that has additional attributes called dimensions. Arrays are just multidimensional matrices. Again, matrices and arrays must contain elements all of the same data class.
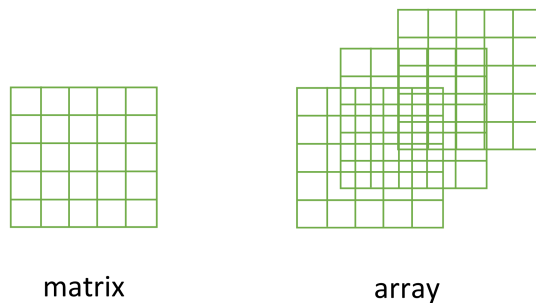
matrix   array

A convenient way to create a matrix or an array is to use the `matrix()` and `array()` functions respectively. Below, we will create a matrix from a sequence 1 to 16 in four rows (`nrow = 4`) and fill the matrix row-wise (`byrow = TRUE`) rather than the default column-wise. When using the `array()` function we define the dimensions using the `dim =` argument, in our case 2 rows, 4 columns in 2 different matrices.

```
my_mat <- matrix(1:16, nrow = 4, byrow = TRUE)
my_mat
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
```

```
## [3,]    9   10   11   12
## [4,]   13   14   15   16

my_array <- array(1:16, dim = c(2, 4, 2))
my_array
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    9   11   13   15
## [2,]   10   12   14   16
```

Sometimes it's also useful to define row and column names for your matrix but this is not a requirement. To do this use the `rownames()` and `colnames()` functions.

```
rownames(my_mat) <- c("A", "B", "C", "D")
colnames(my_mat) <- c("a", "b", "c", "d")
my_mat
##    a  b  c  d
## A  1  2  3  4
## B  5  6  7  8
## C  9 10 11 12
## D 13 14 15 16
```

Once you've created your matrices you can do useful stuff with them and as you'd expect, R has numerous built in functions to perform matrix operations. Some of the most common are given below. For example, to transpose a matrix we use the transposition function `t()`.

```
my_mat_t <- t(my_mat)
my_mat_t
##   A B  C  D
## a 1 5  9 13
## b 2 6 10 14
## c 3 7 11 15
## d 4 8 12 16
```

To extract the diagonal elements of a matrix and store them as a vector we can use the `diag()` function.

```
my_mat_diag <- diag(my_mat)
my_mat_diag
## [1]  1  6 11 16
```

The usual matrix addition, multiplication etc can be performed. Note the use of the
%*% operator to perform matrix multiplication.

```
mat.1 <- matrix(c(2, 0, 1, 1), nrow = 2)    # notice that the matrix has been filled
mat.1                                       # column-wise by default
##      [,1] [,2]
## [1,]    2    1
## [2,]    0    1

mat.2 <- matrix(c(1, 1, 0, 2), nrow = 2)
mat.2
##      [,1] [,2]
## [1,]    1    0
## [2,]    1    2

mat.1 + mat.2          # matrix addition
##      [,1] [,2]
## [1,]    3    1
## [2,]    1    3
mat.1 * mat.2          # element by element products
##      [,1] [,2]
## [1,]    2    0
## [2,]    0    2
mat.1 %*% mat.2          # matrix multiplication
##      [,1] [,2]
## [1,]    3    2
## [2,]    1    2
```

3.2.3 Lists

The next data structure we will quickly take a look at is a list. Whilst vectors and
matrices are constrained to contain data of the same type, lists are able to store mixtures
of data types. In fact we can even store other data structures such as vectors and
arrays within a list or even have a list of a list. This makes for a very flexible data
structure which is ideal for storing irregular or non-rectangular data (see Chapter 7 for
an example).

To create a list we can use the list() function. Note how each of the three list elements
are of different classes (character, logical, and numeric) and are of different lengths.

```r
list_1 <- list(c("black", "yellow", "orange"),
               c(TRUE, TRUE, FALSE, TRUE, FALSE, FALSE),
               matrix(1:6, nrow = 3))
list_1
## [[1]]
## [1] "black"  "yellow" "orange"
##
## [[2]]
## [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE
##
## [[3]]
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Elements of the list can be named during the construction of the list

```r
list_2 <- list(colours = c("black", "yellow", "orange"),
               evaluation = c(TRUE, TRUE, FALSE, TRUE, FALSE, FALSE),
               time = matrix(1:6, nrow = 3))
list_2
## $colours
## [1] "black"  "yellow" "orange"
##
## $evaluation
## [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE
##
## $time
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

or after the list has been created using the `names()` function.

```r
names(list_1) <- c("colours", "evaluation", "time")
list_1
## $colours
## [1] "black"  "yellow" "orange"
##
## $evaluation
```

```
## [1]   TRUE   TRUE FALSE   TRUE FALSE FALSE
##
## $time
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

3.2.4   Data frames

Take a look at this video for a quick introduction to data frame objects in R

By far the most commonly used data structure to store data in is the data frame. A data frame is a powerful two-dimensional object made up of rows and columns which looks superficially very similar to a matrix. However, whilst matrices are restricted to containing data all of the same type, data frames can contain a mixture of different types of data. Typically, in a data frame each row corresponds to an individual observation and each column corresponds to a different measured or recorded variable. This setup may be familiar to those of you who use LibreOffice Calc or Microsoft Excel to manage and store your data. Perhaps a useful way to think about data frames is that they are essentially made up of a bunch of vectors (columns) with each vector containing its own data type but the data type can be different between vectors.

As an example, the data frame below contains the results of an experiment to determine the effect of removing the tip of petunia plants (*Petunia sp.*) grown at 3 levels of nitrogen on various measures of growth (note: data shown below are a subset of the full dataset). The data frame has 8 variables (columns) and each row represents an individual plant. The variables `treat` and `nitrogen` are factors (categorical variables). The `treat` variable has 2 levels (`tip` and `notip`) and the `nitrogen` level variable has 3 levels (`low`, `medium` and `high`). The variables `height`, `weight`, `leafarea` and `shootarea` are numeric and the variable `flowers` is an integer representing the number of flowers. Although the variable `block` has numeric values, these do not really have any order and could also be treated as a factor (i.e. they could also have been called A and B).

There are a couple of important things to bear in mind about data frames. These types of objects are known as rectangular data (or tidy data) as each column must have the same number of observations. Also, any missing data should be recorded as an `NA` just as we did with our vectors.

treat	nitrogen	block	height	weight	leafarea	shootarea	flowers
tip	medium	1	7.5	7.62	11.7	31.9	1
tip	medium	1	10.7	12.14	14.1	46.0	10
tip	medium	1	11.2	12.76	7.1	66.7	10
tip	medium	1	10.4	8.78	11.9	20.3	1
tip	medium	1	10.4	13.58	14.5	26.9	4
tip	medium	1	9.8	10.08	12.2	72.7	9
notip	low	2	3.7	8.10	10.5	60.5	6
notip	low	2	3.2	7.45	14.1	38.1	4
notip	low	2	3.9	9.19	12.4	52.6	9
notip	low	2	3.3	8.92	11.6	55.2	6
notip	low	2	5.5	8.44	13.5	77.6	9
notip	low	2	4.4	10.60	16.2	63.3	6

We can construct a data frame from existing data objects such as vectors using the `data.frame()` function. As an example, let's create three vectors `p.height`, `p.weight` and `p.names` and include all of these vectors in a data frame object called `dataf`.

```
p.height <- c(180, 155, 160, 167, 181)
p.weight <- c(65, 50, 52, 58, 70)
p.names <- c("Joanna", "Charlotte", "Helen", "Karen", "Amy")

dataf <- data.frame(height = p.height, weight = p.weight, names = p.names)
dataf
##   height weight     names
## 1    180     65    Joanna
## 2    155     50 Charlotte
## 3    160     52     Helen
## 4    167     58     Karen
## 5    181     70       Amy
```

You'll notice that each of the columns are named with variable name we supplied when we used the `data.frame()` function. It also looks like the first column of the data frame is a series of numbers from one to five. Actually, this is not really a column but the name of each row. We can check this out by getting R to return the dimensions of the `dataf` object using the `dim()` function. We see that there are 5 rows and 3 columns.

```
dim(dataf)    # 5 rows and 3 columns
## [1] 5 3
```

Another really useful function which we use all the time is `str()` which will return a compact summary of the structure of the data frame object (or any object for that matter).

```
str(dataf)
## 'data.frame':    5 obs. of  3 variables:
##  $ height: num  180 155 160 167 181
##  $ weight: num  65 50 52 58 70
##  $ names : chr  "Joanna" "Charlotte" "Helen" "Karen" ...
```

The `str()` function gives us the data frame dimensions and also reminds us that `dataf` is a `data.frame` type object. It also lists all of the variables (columns) contained in the data frame, tells us what type of data the variables contain and prints out the first five values. We often copy this summary and place it in our R scripts with comments at the beginning of each line so we can easily refer back to it whilst writing our code. We showed you how to comment blocks in RStudio here.

Also notice that R has automatically decided that our `p.names` variable should be a character (`chr`) class variable when we first created the data frame. Whether this is a good idea or not will depend on how you want to use this variable in later analysis. If we decide that this wasn't such a good idea we can change the default behaviour of the `data.frame()` function by including the argument `stringsAsFactors = TRUE`. Now our strings are automatically converted to factors.

```
p.height <- c(180, 155, 160, 167, 181)
p.weight <- c(65, 50, 52, 58, 70)
p.names <- c("Joanna", "Charlotte", "Helen", "Karen", "Amy")

dataf <- data.frame(height = p.height, weight = p.weight, names = p.names,
                                    stringsAsFactors = TRUE)
str(dataf)
## 'data.frame':    5 obs. of  3 variables:
##  $ height: num  180 155 160 167 181
##  $ weight: num  65 50 52 58 70
##  $ names : Factor w/ 5 levels "Amy","Charlotte",..: 4 2 3 5 1
```

3.3 Importing data

Although creating data frames from existing data structures is extremely useful, by far the most common approach is to create a data frame by importing data from an external file. To do this, you'll need to have your data correctly formatted and saved in a file format that R is able to recognise. Fortunately for us, R is able to recognise a wide variety of file formats, although in reality you'll probably end up only using two or three regularly.

Take a look at this video for a quick introduction to importing data in R

3.3.1 Saving files to import

The easiest method of creating a data file to import into R is to enter your data into a spreadsheet using either Microsoft Excel or LibreOffice Calc and save the spreadsheet as a tab delimited file. We prefer LibreOffice Calc as it's open source, platform independent and free but MS Excel is OK too (but see here for some gotchas). Here's the data from the petunia experiment we dicussed previously displayed in LibreOffice. If you want to follow along you can download the data file (*'flower.xls'*) from the companion website here.

For those of you unfamiliar with the tab delimited file format it simply means that data in different columns are separated with a 'tab' character (yes, the same one as on your keyboard) and is usually saved as a file with a '.txt' extension (you might also see `.tsv` which is short for tab separated values).

To save a spreadsheet as a tab delimited file in LibreOffice Calc select `File -> Save as ...` from the main menu. You will need to specify the location you want to save your file in the 'Save in folder' option and the name of the file in the 'Name' option. In the drop down menu located above the 'Save' button change the default 'All formats' to 'Text CSV (.csv)'.
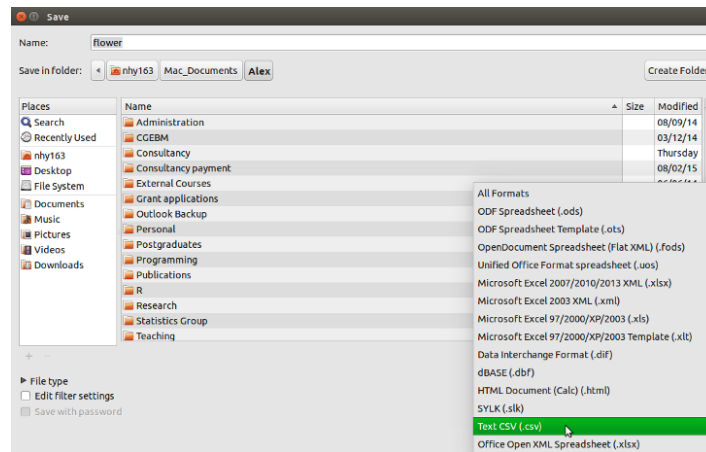
Click the Save button and then select the 'Use Text CSV Format' option. In the next pop-up window select {Tab} from the drop down menu in the 'Field delimiter' option. Click on OK to save the file.
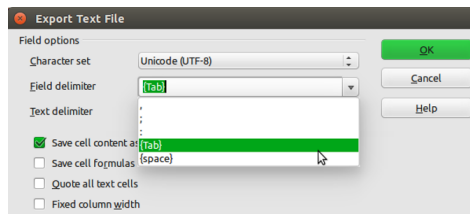
The resulting file will annoyingly have a '.csv' extension even though we've saved it as a tab delimited file. Either live with it or rename the file with a '.txt' extension instead.

In MS Excel, select File -> Save as ... from the main menu and navigate to the folder where you want to save the file. Enter the file name (keep it fairly short, no spaces!) in the 'Save as:' dialogue box. In the 'File format:' dialogue box click on the down arrow to open the drop down menu and select 'Text (Tab delimited)' as your file type. Select OK to save the file.
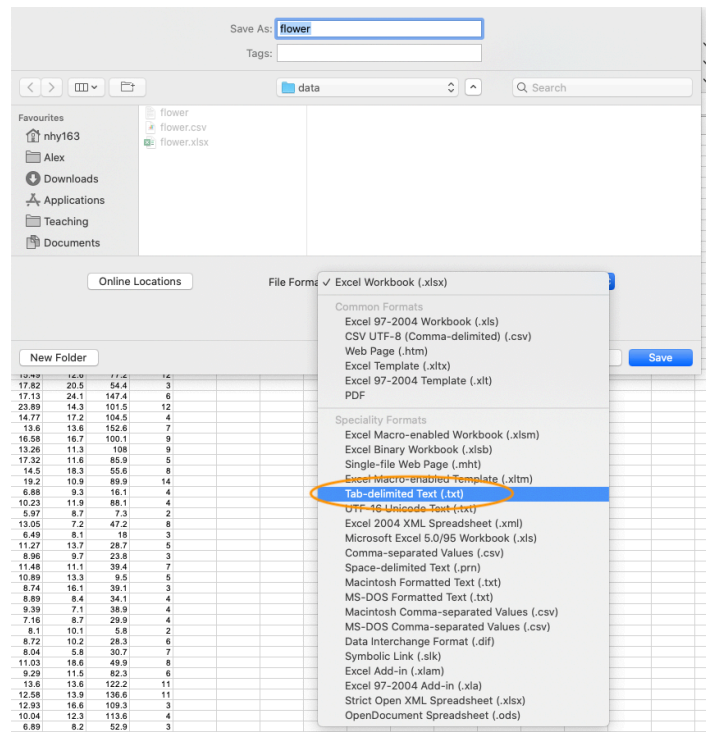
There are a couple of things to bear in mind when saving files to import into R which will make your life easier in the long run. Keep your column headings (if you have them) short and informative. Also avoid spaces in your column headings by replacing them with an underscore or a dot (i.e. replace `shoot height` with `shoot_height` or `shoot.height`) and avoid using special characters (i.e. `leaf area (mm^2)`). Remember, if you have missing data in your data frame (empty cells) you should use an `NA` to represent these missing values. This will keep the data frame tidy.

3.3.2 Import functions

Once you've saved your data file in a suitable format we can now read this file into R. The workhorse function for importing data into R is the `read.table()` function (we discuss some alternatives later in the chapter). The `read.table()` function is a very flexible function with a shed load of arguments (see `?read.table`) but it's quite simple to use. Let's import a tab delimited file called `flower.txt` which contains the data we saw previously in this Chapter and assign it to an object called `flowers`. The file is located in a `data` directory which itself is located in our root directory. The first row of the data contains the variable (column) names. To use the `read.table()` function to import this file.

```
flowers <- read.table(file = 'data/flower.txt', header = TRUE, sep = "\t",
                      stringsAsFactors = TRUE)
```

There are a few things to note about the above command. First, the file path and the filename (including the file extension) needs to be enclosed in either single or double quotes (i.e. the `data/flower.txt` bit) as the `read.table()` function expects this to be a character string. If your working directory is already set to the directory which contains the file, you don't need to include the entire file path just the filename. In the example above, the file path is separated with a single forward slash `/`. This will work regardless of the operating system you are using and we recommend you stick with this. However, Windows users may be more familiar with the single backslash notation and if you want to keep using this you will need to include them as double backslashes. Note though that the double backslash notation will **not** work on computers using Mac OSX or Linux operating systems.

```
flowers <- read.table(file = 'C:\\Documents\\Prog1\\data\\flower.txt',
                      header = TRUE, sep = "\t", stringsAsFactors = TRUE)
```

The `header = TRUE` argument specifies that the first row of your data contains the variable names (i.e. `nitrogen`, `block` etc). If this is not the case you can specify `header = FALSE` (actually, this is the default value so you can omit this argument entirely). The `sep = "\t"` argument tells R that the file delimiter is a tab (`\t`).

After importing our data into R it doesn't appear that R has done much, at least nothing appears in the R Console! To see the contents of the data frame we could just type the name of the object as we have done previously. **BUT** before you do that, think about why you're doing this. If your data frame is anything other than tiny, all you're going to do is fill up your Console with data. It's not like you can easily check whether there are any errors or that your data has been imported correctly. A much better solution is to use our old friend the `str()` function to return a compact and informative summary of your data frame.

```
str(flowers)
## 'data.frame':    96 obs. of  8 variables:
##  $ treat    : Factor w/ 2 levels "notip","tip": 2 2 2 2 2 2 2 2 2 2 ...
##  $ nitrogen : Factor w/ 3 levels "high","low","medium": 3 3 3 3 3 3 3 3 3 3 ...
##  $ block    : int  1 1 1 1 1 1 1 1 2 2 ...
##  $ height   : num  7.5 10.7 11.2 10.4 10.4 9.8 6.9 9.4 10.4 12.3 ...
##  $ weight   : num  7.62 12.14 12.76 8.78 13.58 ...
##  $ leafarea : num  11.7 14.1 7.1 11.9 14.5 12.2 13.2 14 10.5 16.1 ...
##  $ shootarea: num  31.9 46 66.7 20.3 26.9 72.7 43.1 28.5 57.8 36.9 ...
##  $ flowers  : int  1 10 10 1 4 9 7 6 5 8 ...
```

Here we see that `flowers` is a 'data.frame' object which contains 96 rows and 8 variables (columns). Each of the variables are listed along with their data class and the first 10 values. As we mentioned previously in this Chapter, it can be quite convenient to copy and paste this into your R script as a comment block for later reference.

Notice also that your character string variables (`treat` and `nitrogen`) have been imported as factors because we used the argument `stringsAsFactors = TRUE`. If this is not what you want you can prevent this by using the `stringsAsFactors = FALSE` or from R version 4.0.0 you can just leave out this argument as `stringsAsFactors = FALSE` is the default.

```
flowers <- read.table(file = 'data/flower.txt', header = TRUE,
                      sep = "\t", stringsAsFactors = FALSE)
str(flowers)
## 'data.frame':    96 obs. of  8 variables:
##  $ treat    : chr  "tip" "tip" "tip" "tip" ...
##  $ nitrogen : chr  "medium" "medium" "medium" "medium" ...
##  $ block    : int  1 1 1 1 1 1 1 1 2 2 ...
##  $ height   : num  7.5 10.7 11.2 10.4 10.4 9.8 6.9 9.4 10.4 12.3 ...
##  $ weight   : num  7.62 12.14 12.76 8.78 13.58 ...
##  $ leafarea : num  11.7 14.1 7.1 11.9 14.5 12.2 13.2 14 10.5 16.1 ...
##  $ shootarea: num  31.9 46 66.7 20.3 26.9 72.7 43.1 28.5 57.8 36.9 ...
##  $ flowers  : int  1 10 10 1 4 9 7 6 5 8 ...
```

Other useful arguments include `dec =` and `na.strings =`. The `dec =` argument allows you to change the default character (.) used for a decimal point. This is useful if you're in a country where decimal places are usually represented by a comma (i.e. `dec = ","`). The `na.strings =` argument allows you to import data where missing values are represented with a symbol other than NA. This can be quite common if you are importing data from other statistical software such as Minitab which represents missing values as a * (`na.strings = "*"`).

If we just wanted to see the names of our variables (columns) in the data frame we can use the `names()` function which will return a character vector of the variable names.

```
names(flowers)
## [1] "treat"     "nitrogen"  "block"     "height"    "weight"    "leafarea"
## [7] "shootarea" "flowers"
```

R has a number of variants of the `read.table()` function that you can use to import a variety of file formats. Actually, these variants just use the `read.table()` function but include different combinations of arguments by default to help import different file types. The most useful of these are the `read.csv()`, `read.csv2()` and `read.delim()` functions. The `read.csv()` function is used to import comma separated value (.csv) files and assumes that the data in columns are separated by a comma (it sets `sep = ","` by default). It also assumes that the first row of the data contains the variable names by default (it sets `header = TRUE` by default). The `read.csv2()` function assumes data are separated by semicolons and that a comma is used instead of a decimal point (as in many European countries). The `read.delim()` function is used to import tab delimited data and also assumes that the first row of the data contains the variable names by default.

```r
# import .csv file
flowers <- read.csv(file = 'data/flower.csv')


# import .csv file with dec = "," and sep = ";"
flowers <- read.csv2(file = 'data/flower.csv')


# import tab delim file with sep = "\t"
flowers <- read.delim(file = 'data/flower.txt')
```

You can even import spreadsheet files from MS Excel or other statistics software directly into R but our advice is that this should generally be avoided if possible as it just adds a layer of uncertainty between you and your data. In our opinion it's almost always better to export your spreadsheets as tab or comma delimited files and then import them into R using the `read.table()` function. If you're hell bent on directly importing data from other software you will need to install the `foreign` package which has functions for importing Minitab, SPSS, Stata and SAS files or the `xlsx` package to import Excel spreadsheets.

3.3.3 Common import frustrations

It's quite common to get a bunch of really frustrating error messages when you first start importing data into R. Perhaps the most common is

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

This error message is telling you that R cannot find the file you are trying to import. It usually rears its head for one of a couple of reasons (or all of them!). The first is that you've made a mistake in the spelling of either the filename or file path. Another common mistake is that you have forgotten to include the file extension in the filename (i.e. `.txt`). Lastly, the file is not where you say it is or you've used an incorrect file path. Using RStudio Projects and having a logical directory structure goes a long way to avoiding these types of errors.

Another really common mistake is to forget to include the `header = TRUE` argument when the first row of the data contains variable names. For example, if we omit this argument when we import our `flowers.txt` file everything looks OK at first (no error message at least)

```r
flowers_bad <- read.table(file = 'data/flower.txt', sep = "\t")
```

but when we take a look at our data frame using `str()`

```
str(flowers_bad)
## 'data.frame':    97 obs. of  8 variables:
##  $ V1: chr  "treat" "tip" "tip" "tip" ...
##  $ V2: chr  "nitrogen" "medium" "medium" "medium" ...
##  $ V3: chr  "block" "1" "1" "1" ...
##  $ V4: chr  "height" "7.5" "10.7" "11.2" ...
##  $ V5: chr  "weight" "7.62" "12.14" "12.76" ...
##  $ V6: chr  "leafarea" "11.7" "14.1" "7.1" ...
##  $ V7: chr  "shootarea" "31.9" "46" "66.7" ...
##  $ V8: chr  "flowers" "1" "10" "10" ...
```

We can see an obvious problem, all of our variables have been imported as factors and our variables are named V1, V2, V3 ... V8. The problem happens because we haven't told the `read.table()` function that the first row contains the variable names and so it treats them as data. As soon as we have a single character string in any of our data vectors, R treats the vectors as character type data (remember all elements in a vector must contain the same type of data).

3.3.4 Other import options

There are numerous other functions to import data from a variety of sources and formats. Most of these functions are contained in packages that you will need to install before using them. We list a couple of the more useful packages and functions below.

The `fread()` function from the `data.table` package is great for importing large data files quickly and efficiently (much faster than the `read.table()` function). One of the great things about the `fread()` function is that it will automatically detect many of the arguments you would normally need to specify (like `sep =` etc). One thing you might need to consider is that the `fread()` function will return a `data.table` object by default not a `data.frame` as would be the case with the `read.table()` function. This is usually not a problem and you can change this default behaviour by using the argument `data.table = FALSE` when you use `fread()` if you prefer a `data.frame` object. To learn more about the differences between `data.table` and `data.frame` objects see here.

```
library(data.table)
all_data <- fread(file = 'data/flower.txt')
```

Various functions from the `readr` package are also very efficient at reading in large data files. The `readr` package is part of the 'tidyverse' collection of packages and provides many equivalent functions to base R for importing data. The `readr` functions are used in a similar way to the `read.table()` or `read.csv()` functions and many of the arguments are the same (see `?readr::read_table` for more details). There are however some differences.

For example, when using the `read_table()` function the `header = TRUE` argument is replaced by `col_names = TRUE` and the function returns a `tibble` class object which is the tidyverse equivalent of a `data.frame` object (see here for differences).

```
library(readr)
# import white space delimited files
all_data <- read_table(file = 'data/flower.txt', col_names = TRUE)

# import comma delimited files
all_data <- read_csv(file = 'data/flower.txt')

# import tab delimited files
all_data <- read_delim(file = 'data/flower.txt', delim = "\t")

# or use
all_data <- read_tsv(file = 'data/flower.txt')
```

If your data file is ginormous, then the `ff` and `bigmemory` packages may be useful as they both contain import functions that are able to store large data in a memory efficient manner. You can find out more about these functions here and here.

3.4   Wrangling data frames

Now that you're able to successfully import your data from an external file into R our next task is to do something useful with our data. Working with data is a fundamental skill which you'll need to develop and get comfortable with as you'll likely do a lot of it during any project. The good news is that R is especially good at manipulating, summarising and visualising data. Manipulating data (often known as data wrangling or munging) in R can at first seem a little daunting for the new user but if you follow a few simple logical rules then you'll quickly get the hang of it, especially with some practice.

See this video for a general overview on how to use positional and logical indexes to extract data from a data frame object in R

Let's remind ourselves of the structure of the `flowers` data frame we imported in the previous section.

```
flowers <- read.table(file = 'data/flower.txt', header = TRUE, sep = "\t")
str(flowers)
## 'data.frame':    96 obs. of  8 variables:
##  $ treat    : chr  "tip" "tip" "tip" "tip" ...
##  $ nitrogen : chr  "medium" "medium" "medium" "medium" ...
##  $ block    : int  1 1 1 1 1 1 1 1 2 2 ...
##  $ height   : num  7.5 10.7 11.2 10.4 10.4 9.8 6.9 9.4 10.4 12.3 ...
##  $ weight   : num  7.62 12.14 12.76 8.78 13.58 ...
##  $ leafarea : num  11.7 14.1 7.1 11.9 14.5 12.2 13.2 14 10.5 16.1 ...
##  $ shootarea: num  31.9 46 66.7 20.3 26.9 72.7 43.1 28.5 57.8 36.9 ...
##  $ flowers  : int  1 10 10 1 4 9 7 6 5 8 ...
```

To access the data in any of the variables (columns) in our data frame we can use the `$` notation. For example, to access the `height` variable in our `flowers` data frame we can use `flowers$height`. This tells R that the `height` variable is contained within the data frame `flowers`.

```
flowers$height
##  [1]  7.5 10.7 11.2 10.4 10.4  9.8  6.9  9.4 10.4 12.3 10.4 11.0  7.1  6.0  9.0
## [16]  4.5 12.6 10.0 10.0  8.5 14.1 10.1  8.5  6.5 11.5  7.7  6.4  8.8  9.2  6.2
## [31]  6.3 17.2  8.0  8.0  6.4  7.6  9.7 12.3  9.1  8.9  7.4  3.1  7.9  8.8  8.5
## [46]  5.6 11.5  5.8  5.6  5.3  7.5  4.1  3.5  8.5  4.9  2.5  5.4  3.9  5.8  4.5
## [61]  8.0  1.8  2.2  3.9  8.5  8.5  6.4  1.2  2.6 10.9  7.2  2.1  4.7  5.0  6.5
## [76]  2.6  6.0  9.3  4.6  5.2  3.9  2.3  5.2  2.2  4.5  1.8  3.0  3.7  2.4  5.7
## [91]  3.7  3.2  3.9  3.3  5.5  4.4
```

This will return a vector of the `height` data. If we want we can assign this vector to another object and do stuff with it, like calculate a mean or get a summary of the variable using the `summary()` function.

```
f_height <- flowers$height
mean(f_height)
## [1] 6.839583
summary(f_height)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.200   4.475   6.450   6.840   9.025  17.200
```

Or if we don't want to create an additional object we can use functions 'on-the-fly' to only display the value in the console.

```
mean(flowers$height)
## [1] 6.839583
summary(flowers$height)
##    Min. 1st Qu.  Median   Mean 3rd Qu.    Max.
##   1.200   4.475   6.450   6.840   9.025  17.200
```

Just as we did with vectors, we also can access data in data frames using the square bracket [ ] notation. However, instead of just using a single index, we now need to use two indexes, one to specify the rows and one for the columns. To do this, we can use the notation my_data[rows, columns] where rows and columns are indexes and my_data is the name of the data frame. Again, just like with our vectors our indexes can be positional or the result of a logical test.

3.4.1   Positional indexes

To use positional indexes we simple have to write the position of the rows and columns we want to extract inside the [ ]. For example, if for some reason we wanted to extract the first value (1st row) of the height variable (4th column).

```
flowers[1, 4]
## [1] 7.5


# this would give you the same
flowers$height[1]
## [1] 7.5
```

We can also extract values from multiple rows or columns by specifying these indexes as vectors inside the [ ]. To extract the first 10 rows and the first 4 columns we simple supply a vector containing a sequence from 1 to 10 for the rows index (1:10) and a vector from 1 to 4 for the column index (1:4).

```
flowers[1:10, 1:4]
##     treat nitrogen block height
## 1    tip   medium     1    7.5
## 2    tip   medium     1   10.7
## 3    tip   medium     1   11.2
## 4    tip   medium     1   10.4
## 5    tip   medium     1   10.4
## 6    tip   medium     1    9.8
## 7    tip   medium     1    6.9
## 8    tip   medium     1    9.4
## 9    tip   medium     2   10.4
## 10   tip   medium     2   12.3
```

Or for non sequential rows and columns then we can supply vectors of positions using the `c()` function. To extract the 1st, 5th, 12th, 30th rows from the 1st, 3rd, 6th and 8th columns.

```
flowers[c(1, 5, 12, 30), c(1, 3, 6, 8)]
##    treat block leafarea flowers
## 1    tip     1     11.7       1
## 5    tip     1     14.5       4
## 12   tip     2     12.6       6
## 30   tip     2     11.6       5
```

All we are doing in the two examples above is creating vectors of positions for the rows and columns that we want to extract. We have done this by using the skills we developed in Chapter 2 when we generated vectors using the `c()` function or using the `:` notation.

But what if we want to extract either all of the rows or all of the columns? It would be extremely tedious to have to generate vectors for all rows or for all columns. Thankfully R has a shortcut. If you don't specify either a row or column index in the `[ ]` then R interprets it to mean you want all rows or all columns. For example, to extract the first 8 rows and all of the columns in the `flower` data frame

```
flowers[1:8, ]
##    treat nitrogen block height weight leafarea shootarea flowers
## 1    tip   medium     1    7.5   7.62     11.7      31.9       1
## 2    tip   medium     1   10.7  12.14     14.1      46.0      10
## 3    tip   medium     1   11.2  12.76      7.1      66.7      10
## 4    tip   medium     1   10.4   8.78     11.9      20.3       1
## 5    tip   medium     1   10.4  13.58     14.5      26.9       4
## 6    tip   medium     1    9.8  10.08     12.2      72.7       9
## 7    tip   medium     1    6.9  10.11     13.2      43.1       7
## 8    tip   medium     1    9.4  10.28     14.0      28.5       6
```

or all of the rows and the first 3 columns. If you're reading the web version of this book scroll down in output panel to see all of the data. Note, if you're reading the pdf version of the book some of the output has been truncated to save some space.

```
flowers[, 1:3]
```

```
##    treat nitrogen block
## 1    tip   medium     1
## 2    tip   medium     1
## 3    tip   medium     1
```

```
## 4     tip   medium     1
## 5     tip   medium     1
## 6     tip   medium     1
## 7     tip   medium     1
## 8     tip   medium     1
## 9     tip   medium     2
## 10    tip   medium     2
## 87 notip      low      1
## 88 notip      low      1
## 89 notip      low      2
## 90 notip      low      2
## 91 notip      low      2
## 92 notip      low      2
## 93 notip      low      2
## 94 notip      low      2
## 95 notip      low      2
## 96 notip      low      2
```

We can even use negative positional indexes to exclude certain rows and columns. As an example, lets extract all of the rows except the first 85 rows and all columns except the 4th, 7th and 8th columns. Notice we need to use -() when we generate our row positional vectors. If we had just used -1:85 this would actually generate a regular sequence from -1 to 85 which is not what we want (we can of course use -1:-85).

```
flowers[-(1:85), -c(4, 7, 8)]
##     treat nitrogen block weight leafarea
## 86 notip      low     1   6.01     17.6
## 87 notip      low     1   9.93     12.0
## 88 notip      low     1   7.03      7.9
## 89 notip      low     2   9.10     14.5
## 90 notip      low     2   9.05      9.6
## 91 notip      low     2   8.10     10.5
## 92 notip      low     2   7.45     14.1
## 93 notip      low     2   9.19     12.4
## 94 notip      low     2   8.92     11.6
## 95 notip      low     2   8.44     13.5
## 96 notip      low     2  10.60     16.2
```

In addition to using a positional index for extracting particular columns (variables) we can also name the variables directly when using the square bracket [ ] notation. For example, let's extract the first 5 rows and the variables treat, nitrogen and leafarea. Instead of using flowers[1:5, c(1, 2, 6)] we can instead use

```
flowers[1:5, c("treat", "nitrogen", "leafarea")]
##   treat nitrogen leafarea
## 1   tip   medium     11.7
## 2   tip   medium     14.1
## 3   tip   medium      7.1
## 4   tip   medium     11.9
## 5   tip   medium     14.5
```

We often use this method in preference to the positional index for selecting columns as it will still give us what we want even if we've changed the order of the columns in our data frame for some reason.

3.4.2 Logical indexes

Just as we did with vectors, we can also extract data from our data frame based on a logical test. We can use all of the logical operators that we used for our vector examples so if these have slipped your mind maybe pop back and refresh your memory. Let's extract all rows where `height` is greater than 12 and extract all columns by default (remember, if you don't include a column index after the comma it means all columns).

```
big_flowers <- flowers[flowers$height > 12, ]
big_flowers
##     treat nitrogen block height weight leafarea shootarea flowers
## 10    tip   medium     2   12.3  13.48     16.1      36.9       8
## 17    tip     high     1   12.6  18.66     18.6      54.0       9
## 21    tip     high     1   14.1  19.12     13.1     113.2      13
## 32    tip     high     2   17.2  19.20     10.9      89.9      14
## 38    tip      low     1   12.3  11.27     13.7      28.7       5
```

Notice in the code above that we need to use the `flowers$height` notation for the logical test. If we just named the `height` variable without the name of the data frame we would receive an error telling us R couldn't find the variable `height`. The reason for this is that the `height` variable only exists inside the `flowers` data frame so you need to tell R exactly where it is.

```
big_flowers <- flowers[height > 12, ]
Error in `[.data.frame`(flowers, height > 12, ) :
  object 'height' not found
```

So how does this work? The logical test is `flowers$height > 12` and R will only extract those rows that satisfy this logical condition. If we look at the output of just the logical

condition you can see this returns a vector containing TRUE if height is greater than 12 and FALSE if height is not greater than 12.

```
flowers$height > 12
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## [37] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [73] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [85] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

So our row index is a vector containing either TRUE or FALSE values and only those rows that are TRUE are selected.

Other commonly used operators are shown below.

```
flowers[flowers$height >= 6, ]         # values greater or equal to 6

flowers[flowers$height <= 6, ]          # values less than or equal to 6

flowers[flowers$height == 8, ]         # values  equal to 8

flowers[flowers$height != 8, ]         # values  not equal to 8
```

We can also extract rows based on the value of a character string or factor level. Let's extract all rows where the nitrogen level is equal to high (again we will output all columns). Notice that the double equals == sign must be used for a logical test and that the character string must be enclosed in either single or double quotes (i.e. "high").

```
nit_high <- flowers[flowers$nitrogen == "high", ]
nit_high
```

```
##    treat nitrogen block height weight leafarea shootarea flowers
## 17   tip     high     1   12.6  18.66     18.6      54.0       9
## 18   tip     high     1   10.0  18.07     16.9      90.5       3
## 19   tip     high     1   10.0  13.29     15.8     142.7      12
## 20   tip     high     1    8.5  14.33     13.2      91.4       5
## 21   tip     high     1   14.1  19.12     13.1     113.2      13
## 22   tip     high     1   10.1  15.49     12.6      77.2      12
## 23   tip     high     1    8.5  17.82     20.5      54.4       3
## 24   tip     high     1    6.5  17.13     24.1     147.4       6
```

```
## 25    tip     high     2    11.5   23.89    14.3     101.5      12
## 26    tip     high     2     7.7   14.77    17.2     104.5       4
## 71  notip     high     1     7.2   15.21    15.9     135.0      14
## 72  notip     high     1     2.1   19.15    15.6     176.7       6
## 73  notip     high     2     4.7   13.42    19.8     124.7       5
## 74  notip     high     2     5.0   16.82    17.3     182.5      15
## 75  notip     high     2     6.5   14.00    10.1     126.5       7
## 76  notip     high     2     2.6   18.88    16.4     181.5      14
## 77  notip     high     2     6.0   13.68    16.2     133.7       2
## 78  notip     high     2     9.3   18.75    18.4     181.1      16
## 79  notip     high     2     4.6   14.65    16.7      91.7      11
## 80  notip     high     2     5.2   17.70    19.1     181.1       8
```

Or we can extract all rows where `nitrogen` level is not equal to `medium` (using `!=`) and only return columns 1 to 4.

```r
nit_not_medium <- flowers[flowers$nitrogen != "medium", 1:4]
nit_not_medium
```

```
##      treat nitrogen block height
## 17    tip     high     1   12.6
## 18    tip     high     1   10.0
## 19    tip     high     1   10.0
## 20    tip     high     1    8.5
## 21    tip     high     1   14.1
## 22    tip     high     1   10.1
## 23    tip     high     1    8.5
## 24    tip     high     1    6.5
## 25    tip     high     2   11.5
## 26    tip     high     2    7.7
## 87  notip      low     1    3.0
## 88  notip      low     1    3.7
## 89  notip      low     2    2.4
## 90  notip      low     2    5.7
## 91  notip      low     2    3.7
## 92  notip      low     2    3.2
## 93  notip      low     2    3.9
## 94  notip      low     2    3.3
## 95  notip      low     2    5.5
## 96  notip      low     2    4.4
```

We can increase the complexity of our logical tests by combining them with Boolean expressions just as we did for vector objects. For example, to extract all rows where `height` is greater or equal to `6` AND `nitrogen` is equal to `medium` AND `treat` is equal to `notip` we combine a series of logical expressions with the `&` symbol.

```
low_notip_heigh6 <- flowers[flowers$height >= 6 & flowers$nitrogen == "medium" &
                            flowers$treat == "notip", ]
low_notip_heigh6
##    treat nitrogen block height weight leafarea shootarea flowers
## 51 notip   medium     1    7.5  13.60     13.6     122.2      11
## 54 notip   medium     1    8.5  10.04     12.3     113.6       4
## 61 notip   medium     2    8.0  11.43     12.6      43.2      14
```

To extract rows based on an 'OR' Boolean expression we can use the | symbol. Let's
extract all rows where `height` is greater than 12.3 OR less than 2.2.

```
height2.2_12.3 <- flowers[flowers$height > 12.3 | flowers$height < 2.2, ]
height2.2_12.3
##    treat nitrogen block height weight leafarea shootarea flowers
## 17   tip     high     1   12.6  18.66     18.6      54.0       9
## 21   tip     high     1   14.1  19.12     13.1     113.2      13
## 32   tip     high     2   17.2  19.20     10.9      89.9      14
## 62 notip   medium     2    1.8  10.47     11.8     120.8       9
## 68 notip     high     1    1.2  18.24     16.6     148.1       7
## 72 notip     high     1    2.1  19.15     15.6     176.7       6
## 86 notip      low     1    1.8   6.01     17.6      46.2       4
```

An alternative method of selecting parts of a data frame based on a logical expression is
to use the `subset()` function instead of the `[ ]`. The advantage of using `subset()` is that
you no longer need to use the `$` notation when specifying variables inside the data frame
as the first argument to the function is the name of the data frame to be subsetted.
The disadvantage is that `subset()` is less flexible than the `[ ]` notation.

```
tip_med_2 <- subset(flowers, treat == "tip" & nitrogen == "medium" & block == 2)
tip_med_2
##    treat nitrogen block height weight leafarea shootarea flowers
## 9    tip   medium     2   10.4  10.48     10.5      57.8       5
## 10   tip   medium     2   12.3  13.48     16.1      36.9       8
## 11   tip   medium     2   10.4  13.18     11.1      56.8      12
## 12   tip   medium     2   11.0  11.56     12.6      31.3       6
## 13   tip   medium     2    7.1   8.16     29.6       9.7       2
## 14   tip   medium     2    6.0  11.22     13.0      16.4       3
## 15   tip   medium     2    9.0  10.20     10.8      90.1       6
## 16   tip   medium     2    4.5  12.55     13.4      14.4       6
```

And if you only want certain columns you can use the `select =` argument.

```
tipplants <- subset(flowers, treat == "tip" & nitrogen == "medium" & block == 2,
                        select = c("treat", "nitrogen", "leafarea"))
tipplants
##     treat nitrogen leafarea
## 9    tip    medium     10.5
## 10   tip    medium     16.1
## 11   tip    medium     11.1
## 12   tip    medium     12.6
## 13   tip    medium     29.6
## 14   tip    medium     13.0
## 15   tip    medium     10.8
## 16   tip    medium     13.4
```

3.4.3   Ordering data frames

Remember when we used the function `order()` to order one vector based on the order of another vector (way back in Chapter 2). This comes in very handy if you want to reorder rows in your data frame. For example, if we want all of the rows in the data frame `flowers` to be ordered in ascending value of `height` and output all columns by default. If you're reading this section of the book on the web you can scroll down in the output panels to see the entire ordered data frame. If you're reading the pdf version of the book, note that some of the output from the code chunks has been truncated to save some space.

```
height_ord <- flowers[order(flowers$height), ]
height_ord
```

```
##     treat nitrogen block height weight leafarea shootarea flowers
## 68  notip     high     1    1.2  18.24     16.6     148.1       7
## 62  notip   medium     2    1.8  10.47     11.8     120.8       9
## 86  notip      low     1    1.8   6.01     17.6      46.2       4
## 72  notip     high     1    2.1  19.15     15.6     176.7       6
## 63  notip   medium     2    2.2  10.70     15.3      97.1       7
## 84  notip      low     1    2.2   9.97      9.6      63.1       2
## 82  notip      low     1    2.3   7.28     13.8      32.8       6
## 89  notip      low     2    2.4   9.10     14.5      78.7       8
## 56  notip   medium     1    2.5  14.85     17.5      77.8      10
## 69  notip     high     1    2.6  16.57     17.1     141.1       3
## 76  notip     high     2    2.6  18.88     16.4     181.5      14
## 87  notip      low     1    3.0   9.93     12.0      56.6       6
## 42    tip      low     2    3.1   8.74     16.1      39.1       3
## 92  notip      low     2    3.2   7.45     14.1      38.1       4
## 94  notip      low     2    3.3   8.92     11.6      55.2       6
```

We can also order by descending order of a variable (i.e. `leafarea`) using the `decreasing`
`= TRUE` argument.

```
leafarea_ord <- flowers[order(flowers$leafarea, decreasing = TRUE), ]
leafarea_ord
```

```
##      treat nitrogen block height weight leafarea shootarea flowers
## 70 notip     high     1   10.9  17.22     49.2     189.6      17
## 13   tip   medium     2    7.1   8.16     29.6       9.7       2
## 24   tip     high     1    6.5  17.13     24.1     147.4       6
## 65 notip     high     1    8.5  22.53     20.8     166.9      16
## 23   tip     high     1    8.5  17.82     20.5      54.4       3
## 66 notip     high     1    8.5  17.33     19.8     184.4      12
## 73 notip     high     2    4.7  13.42     19.8     124.7       5
## 80 notip     high     2    5.2  17.70     19.1     181.1       8
## 17   tip     high     1   12.6  18.66     18.6      54.0       9
## 49 notip   medium     1    5.6  11.03     18.6      49.9       8
## 78 notip     high     2    9.3  18.75     18.4     181.1      16
## 31   tip     high     2    6.3  14.50     18.3      55.6       8
## 57 notip   medium     2    5.4  11.36     17.8     104.6      12
## 86 notip      low     1    1.8   6.01     17.6      46.2       4
## 56 notip   medium     1    2.5  14.85     17.5      77.8      10
```

We can even order data frames based on multiple variables. For example, to order the
data frame `flowers` in ascending order of both `block` and `height`.

```
block_height_ord <- flowers[order(flowers$block, flowers$height), ]
block_height_ord
```

```
##      treat nitrogen block height weight leafarea shootarea flowers
## 68 notip     high     1    1.2  18.24     16.6     148.1       7
## 86 notip      low     1    1.8   6.01     17.6      46.2       4
## 72 notip     high     1    2.1  19.15     15.6     176.7       6
## 84 notip      low     1    2.2   9.97      9.6      63.1       2
## 82 notip      low     1    2.3   7.28     13.8      32.8       6
## 56 notip   medium     1    2.5  14.85     17.5      77.8      10
## 69 notip     high     1    2.6  16.57     17.1     141.1       3
## 87 notip      low     1    3.0   9.93     12.0      56.6       6
## 53 notip   medium     1    3.5  12.93     16.6     109.3       3
## 88 notip      low     1    3.7   7.03      7.9      36.7       5
## 81 notip      low     1    3.9   7.17     13.5      52.8       6
## 52 notip   medium     1    4.1  12.58     13.9     136.6      11
## 85 notip      low     1    4.5   8.60      9.4     113.5       7
```

```
## 55 notip    medium     1    4.9   6.89      8.2     52.9        3
## 83 notip       low     1    5.2   5.79     11.0     67.4        5
## 50 notip    medium     1    5.3   9.29     11.5     82.3        6
## 49 notip    medium     1    5.6  11.03     18.6     49.9        8
## 35   tip       low     1    6.4   5.97      8.7      7.3        2
## 67 notip      high     1    6.4  11.52     12.1    140.5        7
## 24   tip      high     1    6.5  17.13     24.1    147.4        6
```

What if we wanted to order `flowers` by ascending order of `block` but descending order of `height`? We can use a simple trick by adding a - symbol before the `flowers$height` variable when we use the `order()` function. This will essentially turn all of the `height` values negative which will result in reversing the order. Note, that this trick will only work with numeric variables.

```
block_revheight_ord <- flowers[order(flowers$block, -flowers$height), ]
block_revheight_ord
```

```
##      treat nitrogen block height weight leafarea shootarea flowers
## 21    tip     high     1   14.1  19.12     13.1    113.2       13
## 17    tip     high     1   12.6  18.66     18.6     54.0        9
## 38    tip      low     1   12.3  11.27     13.7     28.7        5
## 3     tip   medium     1   11.2  12.76      7.1     66.7       10
## 70 notip     high     1   10.9  17.22     49.2    189.6       17
## 2     tip   medium     1   10.7  12.14     14.1     46.0       10
## 4     tip   medium     1   10.4   8.78     11.9     20.3        1
## 5     tip   medium     1   10.4  13.58     14.5     26.9        4
## 22    tip     high     1   10.1  15.49     12.6     77.2       12
## 18    tip     high     1   10.0  18.07     16.9     90.5        3
## 19    tip     high     1   10.0  13.29     15.8    142.7       12
## 6     tip   medium     1    9.8  10.08     12.2     72.7        9
## 37    tip      low     1    9.7   6.49      8.1     18.0        3
## 8     tip   medium     1    9.4  10.28     14.0     28.5        6
## 39    tip      low     1    9.1   8.96      9.7     23.8        3
## 79 notip     high     2    4.6  14.65     16.7     91.7       11
## 16    tip   medium     2    4.5  12.55     13.4     14.4        6
## 60 notip   medium     2    4.5  13.68     14.8    125.5        9
## 96 notip      low     2    4.4  10.60     16.2     63.3        6
## 58 notip   medium     2    3.9   9.07      9.6     90.4        7
## 64 notip   medium     2    3.9  12.97     17.0     97.5        5
## 93 notip      low     2    3.9   9.19     12.4     52.6        9
## 91 notip      low     2    3.7   8.10     10.5     60.5        6
## 94 notip      low     2    3.3   8.92     11.6     55.2        6
## 92 notip      low     2    3.2   7.45     14.1     38.1        4
## 42    tip      low     2    3.1   8.74     16.1     39.1        3
## 76 notip     high     2    2.6  18.88     16.4    181.5       14
```

```
## 89 notip      low     2    2.4   9.10     14.5      78.7        8
## 63 notip   medium     2    2.2  10.70     15.3      97.1        7
## 62 notip   medium     2    1.8  10.47     11.8     120.8        9
```

If we wanted to do the same thing with a factor (or character) variable like `nitrogen` we would need to use the function `xtfrm()` for this variable inside our `order()` function.

```
block_revheight_ord <- flowers[order(-xtfrm(flowers$nitrogen), flowers$height), ]
block_revheight_ord
```

```
##      treat nitrogen block height weight leafarea shootarea flowers
## 62 notip   medium     2    1.8  10.47     11.8     120.8        9
## 63 notip   medium     2    2.2  10.70     15.3      97.1        7
## 56 notip   medium     1    2.5  14.85     17.5      77.8       10
## 53 notip   medium     1    3.5  12.93     16.6     109.3        3
## 58 notip   medium     2    3.9   9.07      9.6      90.4        7
## 64 notip   medium     2    3.9  12.97     17.0      97.5        5
## 52 notip   medium     1    4.1  12.58     13.9     136.6       11
## 16    tip   medium     2    4.5  12.55     13.4      14.4        6
## 60 notip   medium     2    4.5  13.68     14.8     125.5        9
## 55 notip   medium     1    4.9   6.89      8.2      52.9        3
## 50 notip   medium     1    5.3   9.29     11.5      82.3        6
## 57 notip   medium     2    5.4  11.36     17.8     104.6       12
## 49 notip   medium     1    5.6  11.03     18.6      49.9        8
## 59 notip   medium     2    5.8  10.18     15.7      88.8        6
## 14    tip   medium     2    6.0  11.22     13.0      16.4        3
## 20    tip     high     1    8.5  14.33     13.2      91.4        5
## 23    tip     high     1    8.5  17.82     20.5      54.4        3
## 65 notip     high     1    8.5  22.53     20.8     166.9       16
## 66 notip     high     1    8.5  17.33     19.8     184.4       12
## 28    tip     high     2    8.8  16.58     16.7     100.1        9
## 29    tip     high     2    9.2  13.26     11.3     108.0        9
## 78 notip     high     2    9.3  18.75     18.4     181.1       16
## 18    tip     high     1   10.0  18.07     16.9      90.5        3
## 19    tip     high     1   10.0  13.29     15.8     142.7       12
## 22    tip     high     1   10.1  15.49     12.6      77.2       12
## 70 notip     high     1   10.9  17.22     49.2     189.6       17
## 25    tip     high     2   11.5  23.89     14.3     101.5       12
## 17    tip     high     1   12.6  18.66     18.6      54.0        9
## 21    tip     high     1   14.1  19.12     13.1     113.2       13
## 32    tip     high     2   17.2  19.20     10.9      89.9       14
```

Notice that the `nitrogen` variable has been reverse ordered alphabetically and `height` has been ordered by increasing values within each level of `nitrogen`.

If we wanted to order the data frame by `nitrogen` but this time order it from `low ->`
`medium -> high` instead of the default alphabetically (`high`, `low`, `medium`), we need to first
change the order of our levels of the `nitrogen` factor in our data frame using the `factor()`
function. Once we've done this we can then use the `order()` function as usual. Note,
if you're reading the pdf version of this book, the output has been truncated to save
space.

```
flowers$nitrogen <- factor(flowers$nitrogen,
                           levels = c("low", "medium", "high"))
nit_ord <- flowers[order(flowers$nitrogen),]
nit_ord
```

```
##       treat nitrogen block height weight leafarea shootarea flowers
## 33    tip       low     1    8.0   6.88      9.3      16.1       4
## 34    tip       low     1    8.0  10.23     11.9      88.1       4
## 35    tip       low     1    6.4   5.97      8.7       7.3       2
## 36    tip       low     1    7.6  13.05      7.2      47.2       8
## 37    tip       low     1    9.7   6.49      8.1      18.0       3
## 38    tip       low     1   12.3  11.27     13.7      28.7       5
## 39    tip       low     1    9.1   8.96      9.7      23.8       3
## 40    tip       low     1    8.9  11.48     11.1      39.4       7
## 41    tip       low     2    7.4  10.89     13.3       9.5       5
## 42    tip       low     2    3.1   8.74     16.1      39.1       3
## 43    tip       low     2    7.9   8.89      8.4      34.1       4
## 44    tip       low     2    8.8   9.39      7.1      38.9       4
## 45    tip       low     2    8.5   7.16      8.7      29.9       4
## 46    tip       low     2    5.6   8.10     10.1       5.8       2
## 47    tip       low     2   11.5   8.72     10.2      28.3       6
## 66  notip      high     1    8.5  17.33     19.8     184.4      12
## 67  notip      high     1    6.4  11.52     12.1     140.5       7
## 68  notip      high     1    1.2  18.24     16.6     148.1       7
## 69  notip      high     1    2.6  16.57     17.1     141.1       3
## 70  notip      high     1   10.9  17.22     49.2     189.6      17
## 71  notip      high     1    7.2  15.21     15.9     135.0      14
## 72  notip      high     1    2.1  19.15     15.6     176.7       6
## 73  notip      high     2    4.7  13.42     19.8     124.7       5
## 74  notip      high     2    5.0  16.82     17.3     182.5      15
## 75  notip      high     2    6.5  14.00     10.1     126.5       7
## 76  notip      high     2    2.6  18.88     16.4     181.5      14
## 77  notip      high     2    6.0  13.68     16.2     133.7       2
## 78  notip      high     2    9.3  18.75     18.4     181.1      16
## 79  notip      high     2    4.6  14.65     16.7      91.7      11
## 80  notip      high     2    5.2  17.70     19.1     181.1       8
```

3.4.4   Adding columns and rows

Sometimes it's useful to be able to add extra rows and columns of data to our data
frames. There are multiple ways to achieve this (as there always is in R!) depending
on your circumstances. To simply append additional rows to an existing data frame we
can use the `rbind()` function and to append columns the `cbind()` function. Let's create
a couple of test data frames to see this in action using our old friend the `data.frame()`
function.

```
# rbind for rows
df1 <- data.frame(id = 1:4, height = c(120, 150, 132, 122),
                             weight = c(44, 56, 49, 45))
df1
##   id height weight
## 1  1    120     44
## 2  2    150     56
## 3  3    132     49
## 4  4    122     45


df2 <- data.frame(id = 5:6, height = c(119, 110),
                             weight = c(39, 35))
df2
##   id height weight
## 1  5    119     39
## 2  6    110     35


df3 <- data.frame(id = 1:4, height = c(120, 150, 132, 122),
                             weight = c(44, 56, 49, 45))
df3
##   id height weight
## 1  1    120     44
## 2  2    150     56
## 3  3    132     49
## 4  4    122     45


df4 <- data.frame(location = c("UK", "CZ", "CZ", "UK"))
df4
##   location
## 1       UK
## 2       CZ
## 3       CZ
## 4       UK
```

We can use the `rbind()` function to append the rows of data in `df2` to the rows in `df1`
and assign the new data frame to `df_rcomb`.

```
df_rcomb <- rbind(df1, df2)
df_rcomb
##   id height weight
## 1  1    120     44
## 2  2    150     56
## 3  3    132     49
## 4  4    122     45
## 5  5    119     39
## 6  6    110     35
```

And `cbind` to append the column in `df4` to the `df3` data frame and assign to `df_ccomb`.

```
df_ccomb <- cbind(df3, df4)
df_ccomb
##   id height weight location
## 1  1    120     44       UK
## 2  2    150     56       CZ
## 3  3    132     49       CZ
## 4  4    122     45       UK
```

Another situation when adding a new column to a data frame is useful is when you want to perform some kind of transformation on an existing variable. For example, say we wanted to apply a $\log_{10}$ transformation on the height variable in the `df_rcomb` data frame we created above. We could just create a separate variable to contains these values but it's good practice to create this variable as a new column inside our existing data frame so we keep all of our data together. Let's call this new variable `height_log10`.

```
# log10 transformation
df_rcomb$height_log10 <- log10(df_rcomb$height)
df_rcomb
##   id height weight height_log10
## 1  1    120     44     2.079181
## 2  2    150     56     2.176091
## 3  3    132     49     2.120574
## 4  4    122     45     2.086360
## 5  5    119     39     2.075547
## 6  6    110     35     2.041393
```

This situation also crops up when we want to convert an existing variable in a data frame from one data class to another data class. For example, the `id` variable in the `df_rcomb` data frame is numeric type data (use the `str()` or `class()` functions to check for yourself). If we wanted to convert the `id` variable to a factor to use later in our analysis we can create a new variable called `Fid` in our data frame and use the `factor()` function to convert the `id` variable.

```
# convert to a factor
df_rcomb$Fid <- factor(df_rcomb$id)
df_rcomb
##   id height weight height_log10 Fid
## 1  1    120     44     2.079181   1
## 2  2    150     56     2.176091   2
## 3  3    132     49     2.120574   3
## 4  4    122     45     2.086360   4
## 5  5    119     39     2.075547   5
## 6  6    110     35     2.041393   6
str(df_rcomb)
## 'data.frame':    6 obs. of  5 variables:
##  $ id         : int  1 2 3 4 5 6
##  $ height     : num  120 150 132 122 119 110
##  $ weight     : num  44 56 49 45 39 35
##  $ height_log10: num  2.08 2.18 2.12 2.09 2.08 ...
##  $ Fid        : Factor w/ 6 levels "1","2","3","4",..: 1 2 3 4 5 6
```

3.4.5   Merging data frames

Instead of just appending either rows or columns to a data frame we can also merge
two data frames together. Let's say we have one data frame that contains taxonomic
information on some common UK rocky shore invertebrates (called `taxa`) and another
data frame that contains information on where they are usually found on the rocky
shore (called `zone`). We can merge these two data frames together to produce a single
data frame with both taxonomic and location information. Let's first create both of
these data frames (in reality you would probably just import your different datasets).

```
taxa <- data.frame(GENUS = c("Patella", "Littorina", "Halichondria", "Semibalanus"),
       species = c("vulgata", "littoria", "panacea", "balanoides"),
       family = c("patellidae", "Littorinidae", "Halichondriidae", "Archaeobalanidae"))
taxa
##           GENUS     species           family
## 1       Patella     vulgata       patellidae
## 2     Littorina    littoria     Littorinidae
## 3  Halichondria     panacea  Halichondriidae
## 4   Semibalanus  balanoides Archaeobalanidae

zone <- data.frame(genus = c("Laminaria", "Halichondria", "Xanthoria", "Littorina",
                    "Semibalanus", "Fucus"),
                 species = c("digitata", "panacea", "parietina", "littoria",
                     "balanoides", "serratus"),
                 zone = c( "v_low", "low", "v_high", "low_mid", "high", "low_mid"))
```

```
zone
## 	       genus	   species	  zone
## 1	   Laminaria	  digitata	 v_low
## 2 Halichondria	   panacea	   low
## 3	   Xanthoria	 parietina	v_high
## 4	   Littorina	  littoria	low_mid
## 5	 Semibalanus balanoides	  high
## 6	       Fucus	  serratus	low_mid
```

Because both of our data frames contains at least one variable in common (`species` in our case) we can simply use the `merge()` function to create a new data frame called `taxa_zone`.

```
taxa_zone <- merge(x = taxa, y = zone)
taxa_zone
## 	    species	       GENUS	          family	     genus	   zone
## 1 balanoides	 Semibalanus Archaeobalanidae	 Semibalanus	   high
## 2	  littoria	   Littorina	   Littorinidae	   Littorina low_mid
## 3	   panacea Halichondria	 Halichondriidae Halichondria	    low
```

Notice that the merged data frame contains only the rows that have `species` information in **both** data frames. There are also two columns called `GENUS` and `genus` because the `merge()` function treats these as two different variables that originate from the two data frames.

If we want to include all data from both data frames then we will need to use the `all = TRUE` argument. The missing values will be included as `NA`.

```
taxa_zone <- merge(x = taxa, y = zone, all = TRUE)
taxa_zone
## 	    species	       GENUS	          family	     genus	   zone
## 1 balanoides	 Semibalanus Archaeobalanidae	 Semibalanus	   high
## 2	   digitata	        <NA>	            <NA>	   Laminaria	  v_low
## 3	   littoria	   Littorina	   Littorinidae	   Littorina low_mid
## 4	    panacea Halichondria	 Halichondriidae Halichondria	    low
## 5	  parietina	        <NA>	            <NA>	   Xanthoria	 v_high
## 6	   serratus	        <NA>	            <NA>	       Fucus low_mid
## 7	    vulgata	     Patella	       patellidae	        <NA>	   <NA>
```

If the variable names that you want to base the merge on are different in each data frame (for example `GENUS` and `genus`) you can specify the names in the first data frame (known as `x`) and the second data frame (known as `y`) using the `by.x =` and `by.y =` arguments.

```
taxa_zone <- merge(x = taxa, y = zone, by.x = "GENUS", by.y = "genus", all = TRUE)
taxa_zone
##           GENUS   species.x             family   species.y     zone
## 1        Fucus        <NA>               <NA>     serratus low_mid
## 2 Halichondria     panacea  Halichondriidae       panacea      low
## 3    Laminaria        <NA>               <NA>     digitata    v_low
## 4    Littorina     littoria     Littorinidae     littoria low_mid
## 5      Patella      vulgata        patellidae         <NA>     <NA>
## 6   Semibalanus  balanoides Archaeobalanidae   balanoides     high
## 7    Xanthoria        <NA>               <NA>     parietina   v_high
```

Or using multiple variable names.

```
taxa_zone <- merge(x = taxa, y = zone, by.x = c("species", "GENUS"),
                   by.y = c("species", "genus"), all = TRUE)
taxa_zone
##      species        GENUS             family     zone
## 1 balanoides  Semibalanus Archaeobalanidae     high
## 2   digitata    Laminaria             <NA>    v_low
## 3   littoria    Littorina     Littorinidae low_mid
## 4    panacea Halichondria  Halichondriidae      low
## 5  parietina    Xanthoria             <NA>   v_high
## 6    serratus        Fucus             <NA> low_mid
## 7     vulgata      Patella        patellidae     <NA>
```

3.4.6   Reshaping data frames

Reshaping data into different formats is a common task. With rectangular type data
(data frames have the same number of rows in each column) there are two main data
frame shapes that you will come across: the 'long' format (sometimes called stacked)
and the 'wide' format. An example of a long format data frame is given below. We can
see that each row is a single observation from an individual subject and each subject
can have multiple rows. This results in a single column of our measurement.

```
long_data <- data.frame(
            subject = rep(c("A", "B", "C", "D"), each = 3),
            sex = rep(c("M", "F", "F", "M"), each =3),
            condition = rep(c("control", "cond1", "cond2"), times = 4),
            measurement = c(12.9, 14.2, 8.7, 5.2, 12.6, 10.1, 8.9,
                            12.1, 14.2, 10.5, 12.9, 11.9))
long_data
##    subject sex condition measurement
```

```
## 1        A   M   control        12.9
## 2        A   M     cond1        14.2
## 3        A   M     cond2         8.7
## 4        B   F   control         5.2
## 5        B   F     cond1        12.6
## 6        B   F     cond2        10.1
## 7        C   F   control         8.9
## 8        C   F     cond1        12.1
## 9        C   F     cond2        14.2
## 10       D   M   control        10.5
## 11       D   M     cond1        12.9
## 12       D   M     cond2        11.9
```

We can also format the same data in the wide format as shown below. In this format we have multiple observations from each subject in a single row with measurements in different columns (`control`, `cond1` and `cond2`). This is a common format when you have repeated measurements from sampling units.

```
wide_data <- data.frame(subject = c("A", "B", "C", "D"),
                sex = c("M", "F", "F", "M"),
                control = c(12.9, 5.2, 8.9, 10.5),
                cond1 = c(14.2, 12.6, 12.1, 12.9),
                cond2 = c(8.7, 10.1, 14.2, 11.9))
wide_data
##   subject sex control cond1 cond2
## 1       A   M    12.9  14.2   8.7
## 2       B   F     5.2  12.6  10.1
## 3       C   F     8.9  12.1  14.2
## 4       D   M    10.5  12.9  11.9
```

Whilst there's no inherent problem with either of these formats we will sometimes need to convert between the two because some functions will require a specific format for them to work. The most common format is the long format.

There are many ways to convert between these two formats but we'll use the `melt()` and `dcast()` functions from the `reshape2` package (you will need to install this package first). The `melt()` function is used to convert from wide to long formats. The first argument for the `melt()` function is the data frame we want to melt (in our case `wide_data`). The `id.vars = c("subject", "sex")` argument is a vector of the variables you want to stack, the `measured.vars = c("control", "cond1", "cond2")` argument identifies the columns of the measurements in different conditions, the `variable.name = "condition"` argument specifies what you want to call the stacked column of your different conditions in your output

data frame and `value.name = "measurement"` is the name of the column of your stacked measurements in your output data frame.

```r
library(reshape2)
wide_data    # remind ourselves what the wide format looks like
##   subject sex control cond1 cond2
## 1       A   M    12.9  14.2   8.7
## 2       B   F     5.2  12.6  10.1
## 3       C   F     8.9  12.1  14.2
## 4       D   M    10.5  12.9  11.9

# convert wide to long
my_long_df <- melt(data = wide_data, id.vars = c("subject", "sex"),
                   measured.vars = c("control", "cond1", "cond2"),
                   variable.name = "condition", value.name = "measurement")
my_long_df
##    subject sex condition measurement
## 1        A   M   control        12.9
## 2        B   F   control         5.2
## 3        C   F   control         8.9
## 4        D   M   control        10.5
## 5        A   M     cond1        14.2
## 6        B   F     cond1        12.6
## 7        C   F     cond1        12.1
## 8        D   M     cond1        12.9
## 9        A   M     cond2         8.7
## 10       B   F     cond2        10.1
## 11       C   F     cond2        14.2
## 12       D   M     cond2        11.9
```

The `dcast()` function is used to convert from a long format data frame to a wide format data frame. The first argument is again is the data frame we want to cast (`long_data` for this example). The second argument is in formula syntax. The `subject + sex` bit of the formula means that we want to keep these columns separate, and the `~ condition` part is the column that contains the labels that we want to split into new columns in our new data frame. The `value.var = "measurement"` argument is the column that contains the measured data.

```r
long_data    # remind ourselves what the long format look like
##    subject sex condition measurement
## 1        A   M   control        12.9
## 2        A   M     cond1        14.2
## 3        A   M     cond2         8.7
## 4        B   F   control         5.2
```

```
## 5        B   F      cond1      12.6
## 6        B   F      cond2      10.1
## 7        C   F    control       8.9
## 8        C   F      cond1      12.1
## 9        C   F      cond2      14.2
## 10       D   M    control      10.5
## 11       D   M      cond1      12.9
## 12       D   M      cond2      11.9

# convert long to wide
my_wide_df <- dcast(data = long_data, subject + sex ~ condition,
                    value.var = "measurement")
my_wide_df
##   subject sex cond1 cond2 control
## 1       A   M  14.2   8.7    12.9
## 2       B   F  12.6  10.1     5.2
## 3       C   F  12.1  14.2     8.9
## 4       D   M  12.9  11.9    10.5
```

3.5 Summarising data frames

Now that we're able to manipulate and extract data from our data frames our next task is to start exploring and getting to know our data. In this section we'll start producing tables of useful summary statistics of the variables in our data frame and in the next two Chapters we'll cover visualising our data with base R graphics and using the ggplot2 package.

A really useful starting point is to produce some simple summary statistics of all of the variables in our flowers data frame using the summary() function.

```
summary(flowers)
##     treat              nitrogen      block         height          weight
##  Length:96           low   :32   Min.   :1.0   Min.   : 1.200   Min.   : 5.790
##  Class :character    medium:32   1st Qu.:1.0   1st Qu.: 4.475   1st Qu.: 9.027
##  Mode  :character    high  :32   Median :1.5   Median : 6.450   Median :11.395
##                                  Mean   :1.5   Mean   : 6.840   Mean   :12.155
##                                  3rd Qu.:2.0   3rd Qu.: 9.025   3rd Qu.:14.537
##                                  Max.   :2.0   Max.   :17.200   Max.   :23.890
##     leafarea         shootarea          flowers
##  Min.   : 5.80    Min.   :  5.80    Min.   : 1.000
##  1st Qu.:11.07    1st Qu.: 39.05    1st Qu.: 4.000
##  Median :13.45    Median : 70.05    Median : 6.000
##  Mean   :14.05    Mean   : 79.78    Mean   : 7.062
```

```
##   3rd Qu.:16.45     3rd Qu.:113.28    3rd Qu.: 9.000
##   Max.   :49.20     Max.   :189.60    Max.   :17.000
```

For numeric variables (i.e. `height`, `weight` etc) the mean, minimum, maximum, median, first (lower) quartile and third (upper) quartile are presented. For factor variables (i.e. `treat` and `nitrogen`) the number of observations in each of the factor levels is given. If a variable contains missing data then the number of `NA` values is also reported.

If we wanted to summarise a smaller subset of variables in our data frame we can use our indexing skills in combination with the `summary()` function. For example, to summarise only the `height`, `weight`, `leafarea` and `shootarea` variables we can include the appropriate column indexes when using the `[ ]`. Notice we include all rows by not specifying a row index.

```r
summary(flowers[, 4:7])
##      height              weight           leafarea        shootarea
##   Min.   : 1.200    Min.   : 5.790    Min.   : 5.80    Min.   :  5.80
##   1st Qu.: 4.475    1st Qu.: 9.027    1st Qu.:11.07    1st Qu.: 39.05
##   Median : 6.450    Median :11.395    Median :13.45    Median : 70.05
##   Mean   : 6.840    Mean   :12.155    Mean   :14.05    Mean   : 79.78
##   3rd Qu.: 9.025    3rd Qu.:14.537    3rd Qu.:16.45    3rd Qu.:113.28
##   Max.   :17.200    Max.   :23.890    Max.   :49.20    Max.   :189.60

# or equivalently
# summary(flowers[, c("height", "weight", "leafarea", "shootarea")])
```

And to summarise a single variable.

```r
summary(flowers$leafarea)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    5.80   11.07   13.45   14.05   16.45   49.20

# or equivalently
# summary(flowers[, 6])
```

As you've seen above, the `summary()` function reports the number of observations in each level of our factor variables. Another useful function for generating tables of counts is the `table()` function. The `table()` function can be used to build contingency tables of different combinations of factor levels. For example, to count the number of observations for each level of `nitrogen`

```
table(flowers$nitrogen)
##
##    low medium   high
##     32     32     32
```

We can extend this further by producing a table of counts for each combination of `nitrogen` and `treat` factor levels.

```
table(flowers$nitrogen, flowers$treat)
##
##          notip tip
##   low       16  16
##   medium    16  16
##   high      16  16
```

A more flexible version of the `table()` function is the `xtabs()` function. The `xtabs()` function uses a formula notation (~) to build contingency tables with the cross-classifying variables separated by a `+` symbol on the right hand side of the formula. `xtabs()` also has a useful `data =` argument so you don't have to include the data frame name when specifying each variable.

```
xtabs(~ nitrogen + treat, data = flowers)
##         treat
## nitrogen notip tip
##   low       16  16
##   medium    16  16
##   high      16  16
```

We can even build more complicated contingency tables using more variables. Note, in the example below the `xtabs()` function has quietly coerced our `block` variable to a factor.

```
xtabs(~ nitrogen + treat + block, data = flowers)
## , , block = 1
##
##         treat
## nitrogen notip tip
##   low        8   8
##   medium     8   8
##   high       8   8
##
```

```
## , , block = 2
##
##        treat
## nitrogen notip tip
##   low        8    8
##   medium     8    8
##   high       8    8
```

And for a nicer formatted table we can nest the `xtabs()` function inside the `ftable()` function to 'flatten' the table.

```
ftable(xtabs(~ nitrogen + treat + block, data = flowers))
##              block 1 2
## nitrogen treat
## low      notip      8 8
##          tip        8 8
## medium   notip      8 8
##          tip        8 8
## high     notip      8 8
##          tip        8 8
```

We can also summarise our data for each level of a factor variable. Let's say we want to calculate the mean value of `height` for each of our `low`, `meadium` and `high` levels of `nitrogen`. To do this we will use the `mean()` function and apply this to the `height` variable for each level of `nitrogen` using the `tapply()` function.

```
tapply(flowers$height, flowers$nitrogen, mean)
##      low   medium     high
## 5.853125 7.012500 7.653125
```

The `tapply()` function is not just restricted to calculating mean values, you can use it to apply many of the functions that come with R or even functions you've written yourself (see Chapter 7 for more details). For example, we can apply the `sd()` function to calculate the standard deviation for each level of `nitrogen` or even the `summary()` function.

```
tapply(flowers$height, flowers$nitrogen, sd)
##      low   medium     high
## 2.828425 3.005345 3.483323
tapply(flowers$height, flowers$nitrogen, summary)
## $low
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

```
##    1.800    3.600    5.550    5.853    8.000   12.300
##
## $medium
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.800    4.500    7.000    7.013    9.950   12.300
##
## $high
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.200    5.800    7.450    7.653    9.475   17.200
```

Note, if the variable you want to summarise contains missing values (NA) you will also need to include an argument specifying how you want the function to deal with the NA values. We saw an example if this in Chapter 2 where the mean() function returned an NA when we had missing data. To include the na.rm = TRUE argument we simply add this as another argument when using tapply().

```
tapply(flowers$height, flowers$nitrogen, mean, na.rm = TRUE)
##      low   medium     high
## 5.853125 7.012500 7.653125
```

We can also use tapply() to apply functions to more than one factor. The only thing to remember is that the factors need to be supplied to the tapply() function in the form of a list using the list() function. To calculate the mean height for each combination of nitrogen and treat factor levels we can use the list(flowers$nitrogen, flowers$treat) notation.

```
tapply(flowers$height, list(flowers$nitrogen, flowers$treat), mean)
##            notip     tip
## low      3.66875 8.0375
## medium   4.83750 9.1875
## high     5.70625 9.6000
```

And if you get a little fed up with having to write flowers$ for every variable you can nest the tapply() function inside the with() function. The with() function allows R to evaluate an R expression with respect to a named data object (in this case flowers).

```
with(flowers, tapply(height, list(nitrogen, treat), mean))
##            notip     tip
## low      3.66875 8.0375
## medium   4.83750 9.1875
## high     5.70625 9.6000
```

The `with()` function also works with many other functions and can save you alot of typing!

Another really useful function for summarising data is the `aggregate()` function. The `aggregate()` function works in a very similar way to `tapply()` but is a bit more flexible.

For example, to calculate the mean of the variables `height`, `weight`, `leafarea` and `shootarea` for each level of `nitrogen`.

```
aggregate(flowers[, 4:7], by = list(nitrogen = flowers$nitrogen), FUN = mean)
##   nitrogen   height    weight leafarea shootarea
## 1      low 5.853125  8.652812 11.14375   45.1000
## 2   medium 7.012500 11.164062 13.83125   67.5625
## 3     high 7.653125 16.646875 17.18125  126.6875
```

In the code above we have indexed the columns we want to summarise in the `flowers` data frame using `flowers[, 4:7]`. The `by =` argument specifies a list of factors (`list(nitrogen = flowers$nitrogen)`) and the `FUN =` argument names the function to apply (`mean` in this example).

Similar to the `tapply()` function we can include more than one factor to apply a function to. Here we calculate the mean values for each combination of `nitrogen` and `treat`

```
aggregate(flowers[, 4:7], by = list(nitrogen = flowers$nitrogen,
                                    treat = flowers$treat), FUN = mean)
##   nitrogen treat  height    weight leafarea shootarea
## 1      low notip 3.66875  8.289375 12.32500   59.89375
## 2   medium notip 4.83750 11.316875 14.17500   94.53125
## 3     high notip 5.70625 16.604375 18.81875  155.31875
## 4      low   tip 8.03750  9.016250  9.96250   30.30625
## 5   medium   tip 9.18750 11.011250 13.48750   40.59375
## 6     high   tip 9.60000 16.689375 15.54375   98.05625
```

We can also use the `aggregate()` function in a different way by using the formula method (as we did with `xtabs()`). On the left hand side of the formula (`~`) we specify the variable we want to apply the mean function on and to the right hand side our factors separated by a `+` symbol. The formula method also allows you to use the `data =` argument for convenience.

```
aggregate(height ~ nitrogen + treat, FUN = mean, data = flowers)
##   nitrogen treat  height
## 1      low notip 3.66875
## 2   medium notip 4.83750
## 3     high notip 5.70625
```

```
## 4      low   tip 8.03750
## 5   medium   tip 9.18750
## 6     high   tip 9.60000
```

One advantage of using the formula method is that we can also use the `subset =` argument to apply the function to subsets of the original data. For example, to calculate the mean `height` for each combination of the `nitrogen` and `treat` levels but only for those plants that have less than 7 `flowers`.

```
aggregate(height ~ nitrogen + treat, FUN = mean, subset = flowers < 7, data = flowers)
##   nitrogen treat   height
## 1      low notip 3.533333
## 2   medium notip 5.316667
## 3     high notip 3.850000
## 4      low   tip 8.176923
## 5   medium   tip 8.570000
## 6     high   tip 7.900000
```

Or for only those plants in `block` 1.

```
aggregate(height ~ nitrogen + treat, FUN = mean, subset = block == "1", data = flowers)
##   nitrogen treat  height
## 1      low notip  3.3250
## 2   medium notip  5.2375
## 3     high notip  5.9250
## 4      low   tip  8.7500
## 5   medium   tip  9.5375
## 6     high   tip 10.0375
```

3.6   Exporting data

By now we hope you're getting a feel for how powerful and useful R is for manipulating and summarising data (and we've only really scratched the surface). One of the great benefits of doing all your data wrangling in R is that you have a permanent record of all the things you've done to your data. Gone are the days of making undocumented changes in Excel or Calc! By treating your data as 'read only' and documenting all of your decisions in R you will have made great strides towards making your analysis more reproducible and transparent to others. It's important to realise, however, that any changes you've made to your data frame in R will not change the original data file you imported into R (and that's a good thing). Happily it's straightforward to export data frames to external files in a wide variety of formats.

3.6.1   Export functions

The main workhorse function for exporting data frames is the `write.table()` function. As with the `read.table()` function, the `write.table()` function is very flexible with lots of arguments to help customise it's behaviour. As an example, let's take our original `flowers` data frame, do some useful stuff to it and then export these changes to an external file.

Let's order the rows in the data frame in ascending order of `height` within each level `nitrogen`. We will also apply a square root transformation on the number of flowers variable (`flowers`) and a $\log_{10}$ transformation on the `height` variable and save these as additional columns in our data frame (hopefully this will be somewhat familiar to you!).
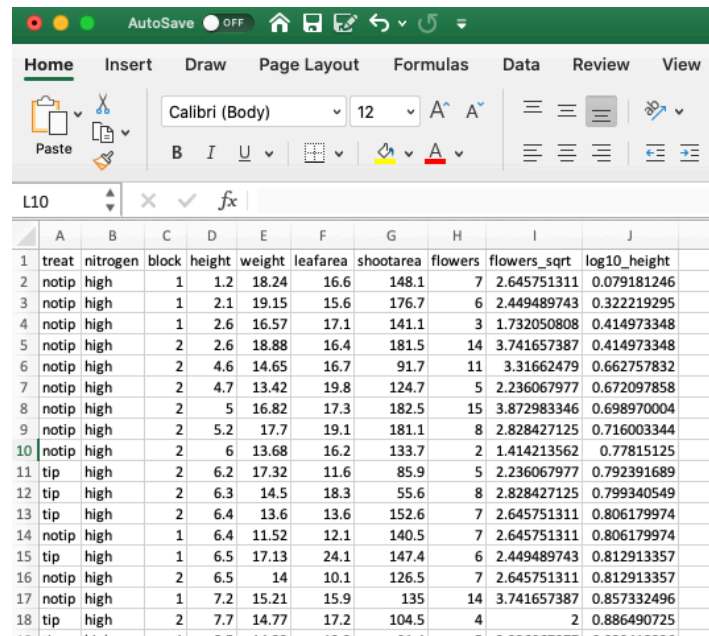
```
flowers_df2 <- flowers[order(flowers$nitrogen, flowers$height), ]
flowers_df2$flowers_sqrt <- sqrt(flowers_df2$flowers)
flowers_df2$log10_height <- log10(flowers_df2$height)
str(flowers_df2)
## 'data.frame':    96 obs. of  10 variables:
##  $ treat      : chr  "notip" "notip" "notip" "notip" ...
##  $ nitrogen   : Factor w/ 3 levels "low","medium",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ block      : int  1 1 1 2 1 2 2 2 1 2 ...
##  $ height     : num  1.8 2.2 2.3 2.4 3 3.1 3.2 3.3 3.7 3.7 ...
##  $ weight     : num  6.01 9.97 7.28 9.1 9.93 8.74 7.45 8.92 7.03 8.1 ...
##  $ leafarea   : num  17.6 9.6 13.8 14.5 12 16.1 14.1 11.6 7.9 10.5 ...
##  $ shootarea  : num  46.2 63.1 32.8 78.7 56.6 39.1 38.1 55.2 36.7 60.5 ...
##  $ flowers    : int  4 2 6 8 6 3 4 6 5 6 ...
##  $ flowers_sqrt: num  2 1.41 2.45 2.83 2.45 ...
##  $ log10_height: num  0.255 0.342 0.362 0.38 0.477 ...
```

Now we can export our new data frame `flowers_df2` using the `write.table()` function. The first argument is the data frame you want to export (`flowers_df2` in our example). We then give the filename (with file extension) and the file path in either single or double quotes using the `file =` argument. In this example we're exporting the data frame to a file called `flowers_04_12.txt` in the `data` directory. The `col.names = TRUE` argument indicates that the variable names should be written in the first row of the file and the `row.names = FALSE` argument stops R from including the row names in the first column of the file. Finally, the `sep = "\t"` argument indicates that a Tab should be used as the delimiter in the exported file.

```
write.table(flowers_df2, file = 'data/flowers_04_12.txt', col.names = TRUE,
            row.names = FALSE, sep = "\t")
```

As we saved the file as a tab delimited text file we could open this file in any text editor. Perhaps a more familiar option would be to open the file in Excel. First start Excel

and then select `File -> Open ..` in the main menu and then select our `flowers_04_12.txt` file to open. Next. choose the 'Tab' option to set the delimiter and then click on the 'Finish' button.

We can of course export our files in a variety of other formats. Another popular option is to export files in csv (comma separated values) format. We can do this using the `write.table()` function by setting the separator argument to `sep = ","`.

```
write.table(flowers_df2, file = 'data/flowers_04_12.csv', col.names = TRUE,
            row.names = FALSE, sep = ",")
```

Or alternatively by using the convenience function `write.csv()`. Notice that we don't need to set the `sep = ","` or `col.names = TRUE` arguments as these are the defaults when using the `read.csv()` function.

```
write.csv(flowers_df2, file = 'data/flowers_04_12.csv', row.names = FALSE)
```

3.6.2 Other export functions

As with importing data files into R, there are also many alternative functions for exporting data to external files beyond the `write.table()` function. If you followed the 'Other import functions' section of this Chapter you will already have the required packages installed.

The `fwrite()` function from the `data.table` package is very efficient at exporting large data objects and is much faster than the `write.table()` function. It's also quite simple to use as it has most of the same arguments as `write.table()`. To export a tab delimited text file we just need to specify the data frame name, the output file name and file path and the separator between columns.

```
library(data.table)
fwrite(flowers_df2, file = 'data/flowers_04_12.txt', sep = "\t")
```

To export a csv delimited file it's even easier as we don't even need to include the `sep =` argument.

```
library(data.table)
fwrite(flowers_df2, file = 'data/flowers_04_12.csv')
```

The `readr` package also comes with two useful functions for quickly writing data to external files: the `write_tsv()` function for writing tab delimited files and the `write_csv()` function for saving comma separated values (csv) files.

```
library(readr)
write_tsv(flowers_df2, file = 'data/flowers_04_12.txt')

write_csv(flowers_df2, file = 'data/flowers_04_12.csv')
```

You can even save data directly to an Excel spreadsheet using the `write_excel_csv()` function but again we don't recommend this!

3.7   Exercise 3

Congratulations, you've reached the end of Chapter 3! Perhaps now's a good time to practice some of what you've learned. You can find an exercise we've prepared for you (and our solutions) on the course website.

Chapter 4

Graphics with base R

Summarising your data, either numerically or graphically, is an important (if often overlooked) component of any data analysis. Fortunately, R has excellent graphics capabilities and can be used whether you want to produce plots for initial data exploration, model validation or highly complex publication quality figures. There are three main systems for producing graphics in R; base R graphics, lattice graphics and ggplot2. Each of these systems have their strengths and weaknesses and we often use them interchangeably. In this Chapter we'll focus mostly on base R graphics with a sprinkling of lattice graphics for added variety. In the next Chapter we'll introduce you to the `ggplot2` package.

The base R graphics system is the original plotting system that's been around (and has evolved) since the first days of R. When creating plots with base R we tend to use high level functions (like the `plot()` function) to first create our plot and then use one or more low level functions (like `lines()` and `text()` etc) to add additional information to these plots. This can seem a little weird (and time consuming) when you first start creating fancy plots in R, but it does allow you to customise almost every aspect of your plot and build complexity up in layers. The flip side to this flexibility is that you'll often need to make many decisions about how you want your plot to look rather than rely on the software to make these decisions for you. Having said that, it's generally very quick and easy to generate simple exploratory plots with base R graphics.

The lattice system is implemented in the `lattice()` package that comes pre-installed with the standard installation of R. However, it won't be loaded by default so you'll first need to use `library(lattice)` to access all the plotting functions. Unlike base R graphics, lattice plots are mostly generated all in one go using a single function so there's no need to use high and low level plotting functions to customise the look of a plot. This can be a real advantage as things like margin sizes and plot spacing are adjusted automatically. Lattice plots also make a few more decisions for you about how the plots will look but this comes with a slight cost as customising lattice plots to get them to look exactly how you want can become quite involved. Where lattice plots really shine is plotting complex multi-dimensional data using panel plots (also called trellis plots). We'll see a couple of examples of these types of plots later in the Chapter.

4.1   Getting started

When you create a plot in RStudio the plot will be displayed in the 'Plots' tab by default which is usually located in the bottom right pane in RStudio.
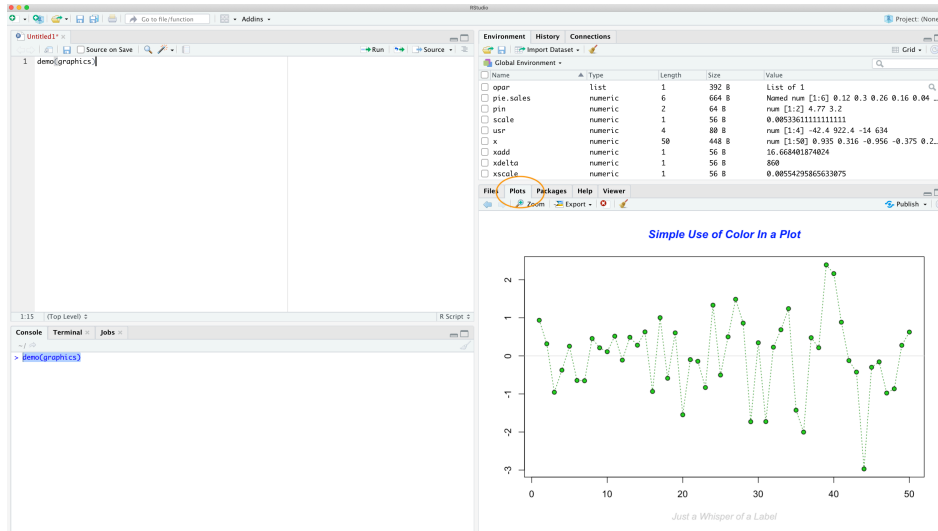
You can zoom in on a plot by clicking the 'Zoom' button which will display your plot in a separate window. This can be really useful if you have a particularly large or complex plot (we've noticed that RStudio sometimes fails to display a plot if it's 'big'). You can also scroll through plots you've previously created by clicking on one of the 'arrow' buttons.
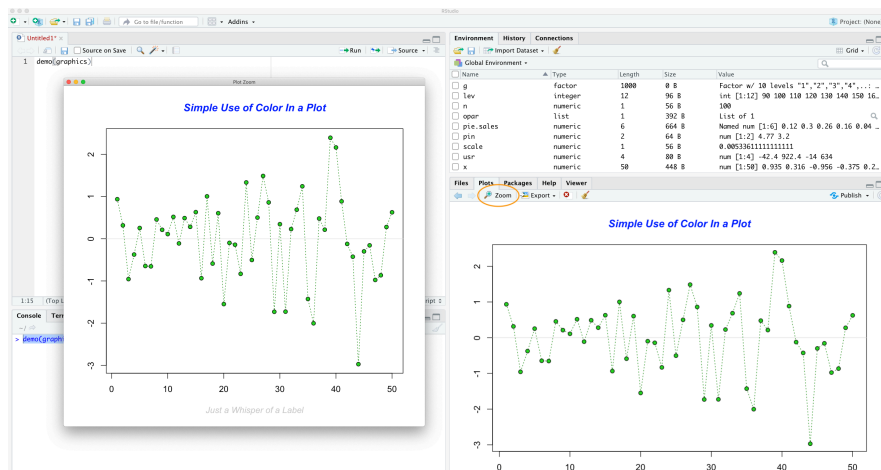
Plots can also be saved in a variety of formats (pdf, png, tiff, jpeg etc) by clicking on the 'Export' button and selecting your desired format. You can also redirect your plots to an external file using R code which we'll cover later in this Chapter.
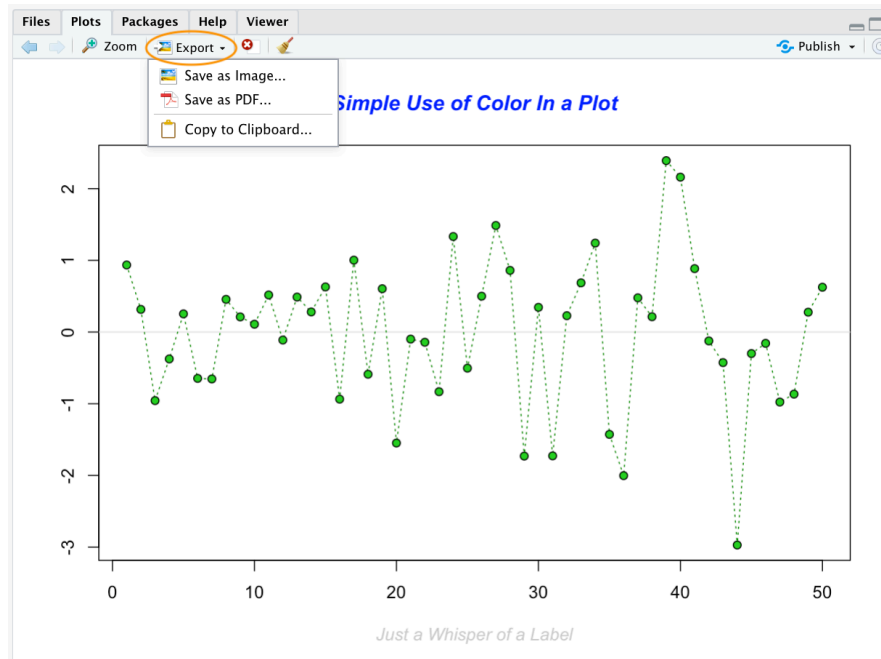
4.2 Simple base R plots

There are many functions in R to produce plots ranging from the very basic to the highly complex. It's impossible to cover every aspect of producing graphics in R in this introductory book so we'll introduce you to most of the common methods of graphing data and describe how to customise your graphs later on in this Chapter.

4.2.1 Scatterplots

The most common high level function used to produce plots in R is (rather unsurprisingly) the `plot()` function. For example, let's plot the `weight` of petunia plants from our `flowers` data frame which we imported in Chapter 3.

```
flowers <- read.table(file = 'data/flower.txt',
                      header = TRUE, sep = "\t",
                      stringsAsFactors = TRUE)


plot(flowers$weight)
```
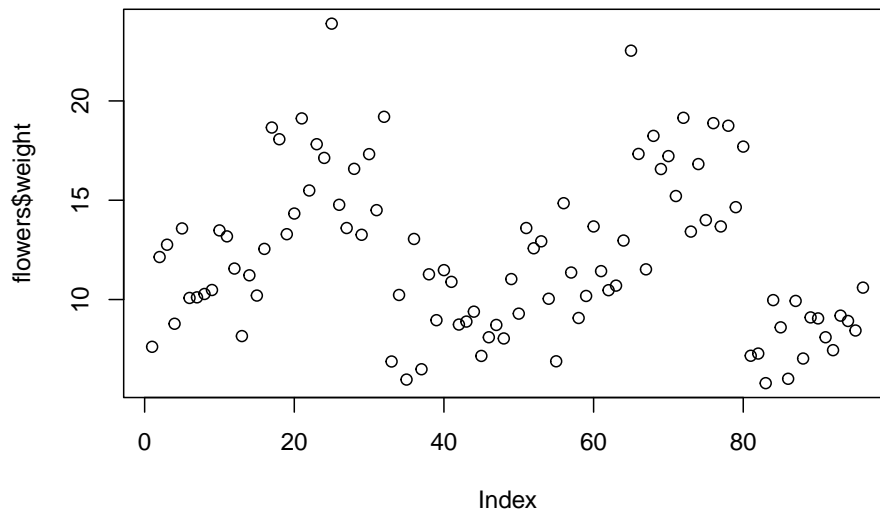
R has plotted the values of `weight` (on the y axis) against an index since we are only plotting one variable to plot. The index is just the order of the `weight` values in the data frame (1 first in the data frame and 97 last). The `weight` variable name has been automatically included as a y axis label and the axes scales have been automatically set.

If we'd only included the variable `weight` rather than `flowers$weight`, the `plot()` function will display an error as the variable `weight` only exists in the `flowers` data frame object.

```
plot(weight)

Error in plot(weight) : object 'weight' not found
```

As many of the base R plotting functions don't have a `data =` argument to specify the data frame name directly we can use the `with()` function in combination with `plot()` as a shortcut.

```
with(flowers, plot(weight))
```

To plot a scatterplot of one numeric variable against another numeric variable we just need to include both variables as arguments when using the `plot()` function. For example to plot `shootarea` on the y axis and `weight` of the x axis.

```
plot(x = flowers$weight, y = flowers$shootarea)
```
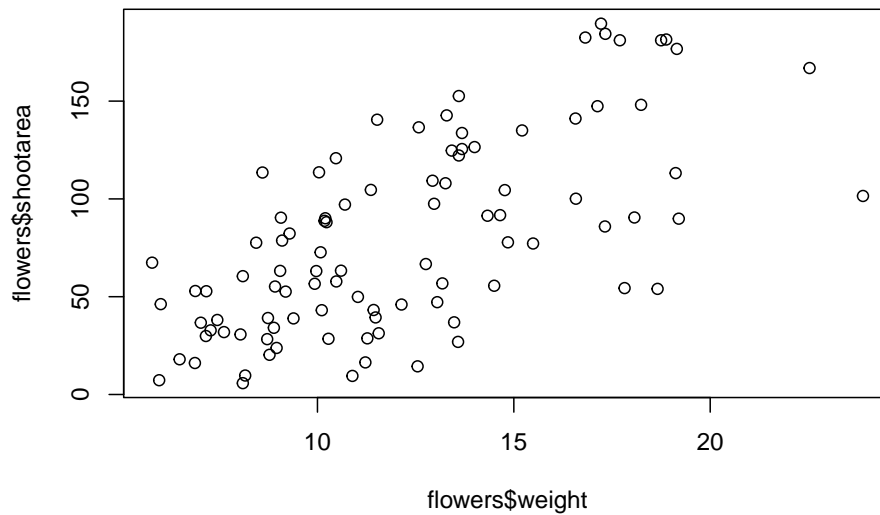
There is an equivalent approach for these types of plots which often causes some confusion at first. You can also use the formula notation when using the `plot()` function. However, in contrast to the previous method the formula method requires you to specify the y axis variable first, then a ~ and then our x axis variable.

```r
plot(flowers$shootarea ~ flowers$weight)
```
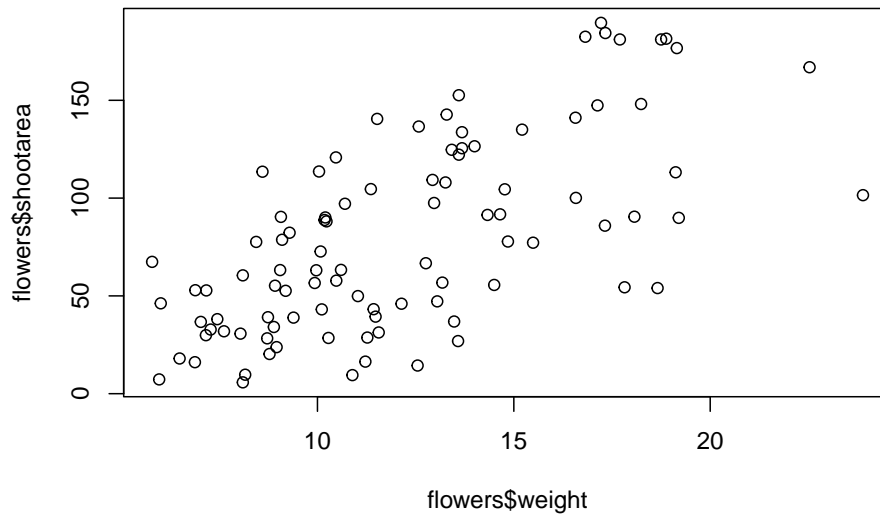

Both of these two approaches are equivalent so we suggest that you just choose the one you prefer and go with it.

You can also specify the type of graph you wish to plot using the argument `type =`. You can plot just the points (`type = "p"`, this is the default), just lines (`type = "l"`), both points and lines connected (`type = "b"`), both points and lines with the lines running through the points (`type = "o"`) and empty points joined by lines (`type = "c"`). For example, let's use our skills from Chapter 2 to generate two vectors of numbers (`my_x` and `my_y`) and then plot one against the other using different `type =` values to see what type of plots are produced. Don't worry about the `par(mfrow = c(2, 2))` line of code yet.

We're just using this to split the plotting device so we can fit all four plots on the same device to save some space. See later in the Chapter for more details about this. The top left plot is `type = "l"`, the top right `type = "b"`, bottom left `type = "o"` and bottom right is `type = "c"`.

```r
my_x <- 1:10
my_y <- seq(from = 1, to = 20, by = 2)

par(mfrow = c(2, 2))
plot(my_x, my_y, type = "l")
plot(my_x, my_y, type = "b")
plot(my_x, my_y, type = "o")
plot(my_x, my_y, type = "c")
```
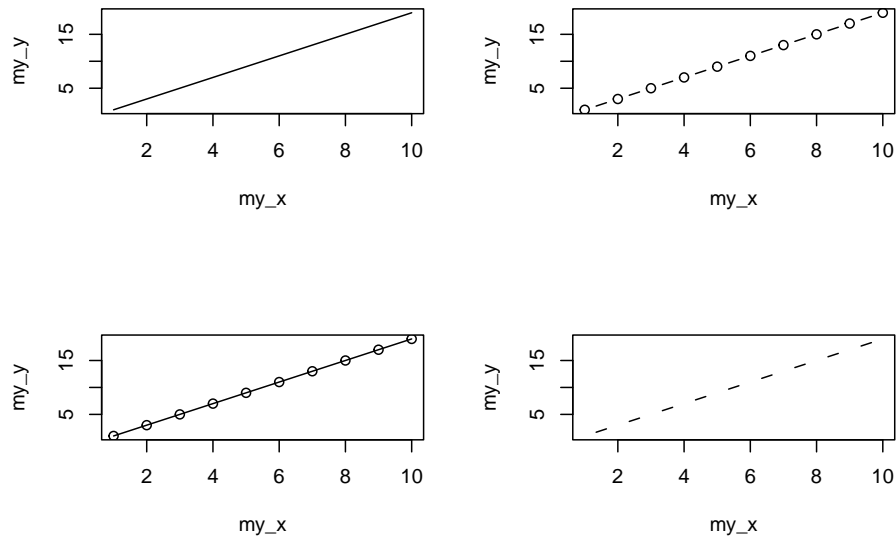

Play around: Try to create four more plots by using `type = 'p'`, `type = 'h'`, `type = 's'`, and `type = 'n'` arguments in `plot()` function.

Admittedly the plots we've produced so far don't look anything particularly special. However, the `plot()` function is incredibly versatile and can generate a large range of plots which you can customise to your own taste. We'll cover how to customise plots later in the Chapter. As a quick aside, the `plot()` function is also what's known as a generic function which means it can change its default behaviour depending on the type of object used as an argument. You will see an example of this in Chapter 6 where we use the `plot()` function to generate diagnostic plots of residuals from a linear model object (bet you can't wait!).

4.2.2   Histograms

Frequency histograms are useful when you want to get an idea about the distribution of values in a numeric variable. The `hist()` function takes a numeric vector as its main argument. Let's generate a histogram of the `height` values.

```r
hist(flowers$height)
```


Histogram of flowers$height
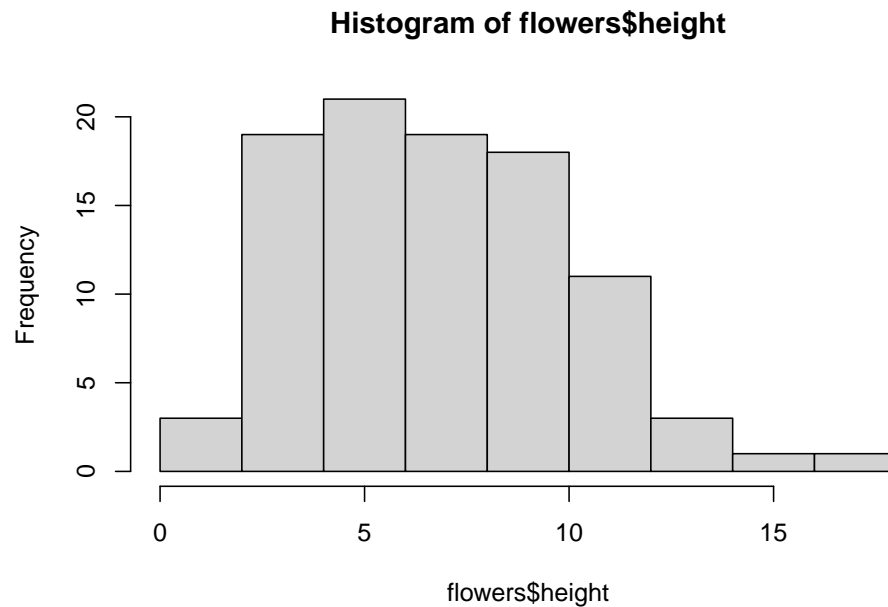
The `hist()` function automatically creates the breakpoints (or bins) in the histogram using the Sturges formula unless you specify otherwise by using the `break =` argument. For example, let's say we want to plot our histogram with breakpoints every 1 cm flower height. We first generate a sequence from zero to the maximum value of `height` (18 rounded up) in steps of 1 using the `seq()` function. We can then use this sequence with the `breaks =` argument. While we're at it, let's also replace the ugly title for something a little better using the `main =` argument

```r
brk <- seq(from = 0, to = 18, by = 1)
hist(flowers$height, breaks = brk, main = "petunia height")
```
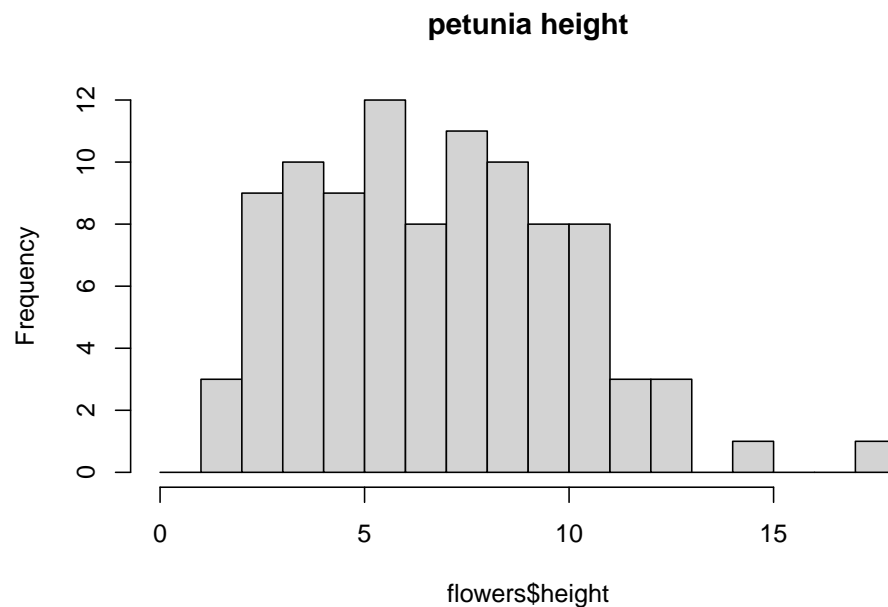

petunia height

You can also display the histogram as a proportion rather than a frequency by using the freq = FALSE argument.

```r
brk <- seq(from = 0, to = 18, by = 1)
hist(flowers$height, breaks = brk, main = "petunia height",
       freq = FALSE)
```
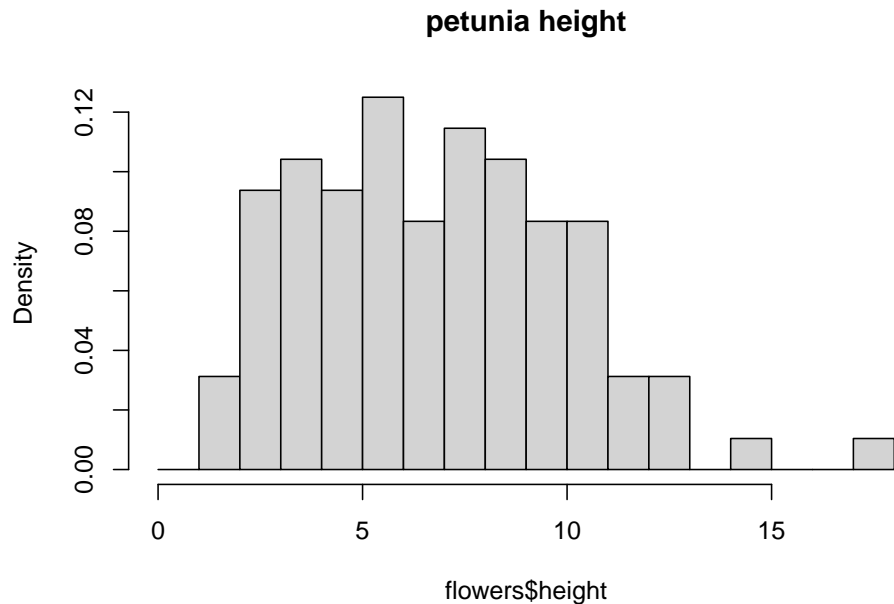

An alternative to plotting just a straight up histogram is to add a kernel density curve to the plot. You can superimpose a density curve onto the histogram by first using the density() function to compute the kernel density estimates and then use the low level function lines() to add these estimates onto the plot as a line.

```r
dens <- density(flowers$height)
hist(flowers$height, breaks = brk, main = "petunia height",
       freq = FALSE)
lines(dens)
```

petunia height
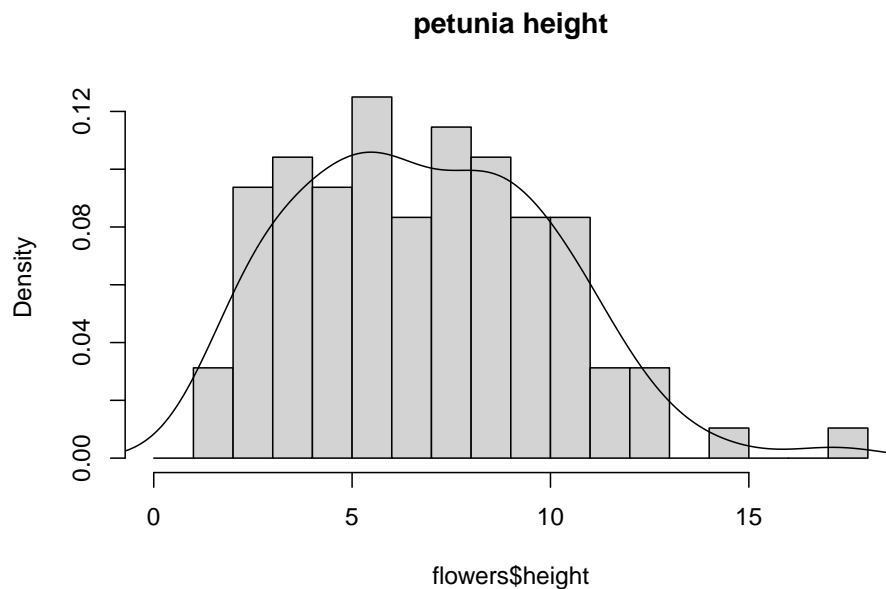
4.2.3 Box and violin plots

OK, we'll just come and out and say it, we love boxplots and their close relation the violin plot. Boxplots (or box-and-whisker plots to give them their full name) are very useful when you want to graphically summarise the distribution of a variable, identify potential unusual values and compare distributions between different groups. The reason we love them is their ease of interpretation, transparency and relatively high data-to-ink ratio (i.e. they convey lots of information efficiently). We suggest that you try to use boxplots as much as possible when exploring your data and avoid the temptation to use the more ubiquitous bar plot (even with standard error or 95% confidence intervals bars). The problem with bar plots (aka dynamite plots) is that they hide important information from the reader such as the distribution of the data and assume that the error bars (or confidence intervals) are symmetric around the mean. Of course, it's up to you what you do but if you're tempted to use bar plots just Google 'dynamite plots are evil' or see here or here for a fuller discussion.

To create a boxplot in R we use the `boxplot()` function. For example, let's create a boxplot of the variable `weight` from our `flowers` data frame. We can also include a y axis label using the `ylab =` argument.

```
boxplot(flowers$weight, ylab = "weight (g)")
```
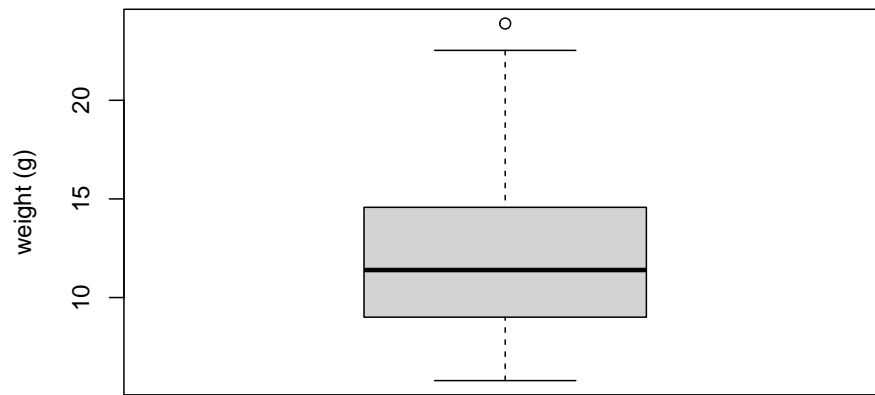
The thick horizontal line in the middle of the box is the median value of `weight` (around 11 g). The upper line of the box is the upper quartile (75th percentile) and the lower line is the lower quartile (25th percentile). The distance between the upper and lower quartiles is known as the inter quartile range and represents the values of `weight` for 50% of the data. The dotted vertical lines are called the whiskers and their length is determined as 1.5 x the inter quartile range. Data points that are plotted outside the the whiskers represent potential unusual observations. This doesn't mean they are unusual, just that they warrant a closer look. We recommend using boxplots in combination with Cleveland dotplots to identify potential unusual observations (see the next section of this Chapter for more details). The neat thing about boxplots is that they not only provide a measure of central tendency (the median value) they also give you an idea about the distribution of the data. If the median line is more or less in the middle of the box (between the upper and lower quartiles) and the whiskers are more or less the same length then you can be reasonably sure the distribution of your data is symmetrical.

If we want examine how the distribution of a variable changes between different levels of a factor we need to use the formula notation with the `boxplot()` function. For example, let's plot our `weight` variable again, but this time see how this changes with each level of `nitrogen`. When we use the formula notation with `boxplot()` we can use the `data =` argument to save some typing. We'll also introduce an x axis label using the `xlab =` argument.

```r
boxplot(weight ~ nitrogen, data = flowers,
        ylab = "weight (g)", xlab = "nitrogen level")
```
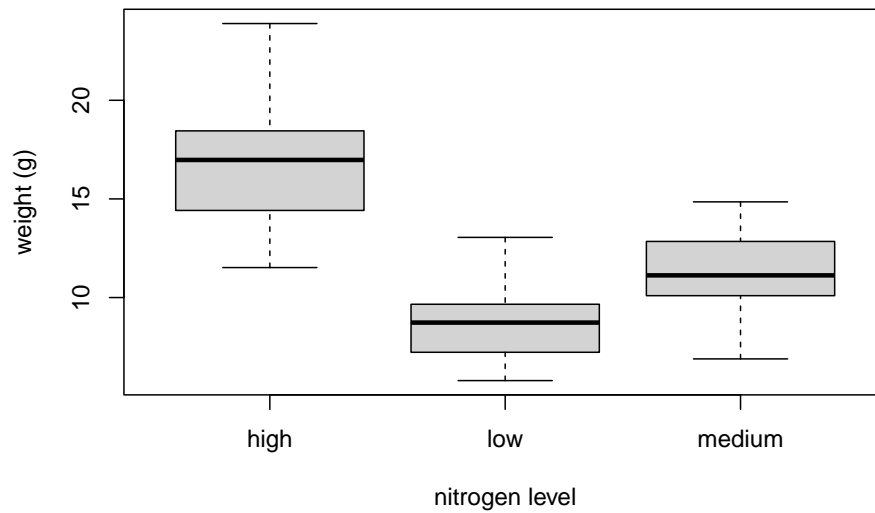
The factor levels are plotted in the same order defined by our factor variable `nitrogen` (often alphabetically). To change the order we need to change the order of our levels of the `nitrogen` factor in our data frame using the `factor()` function and then re-plot the graph. Let's plot our boxplot with our factor levels going from `low` to `high`.

```r
flowers$nitrogen <- factor(flowers$nitrogen,
                           levels = c("low", "medium", "high"))
boxplot(weight ~ nitrogen, data = flowers,
        ylab = "weight (g)", xlab = "nitrogen level")
```
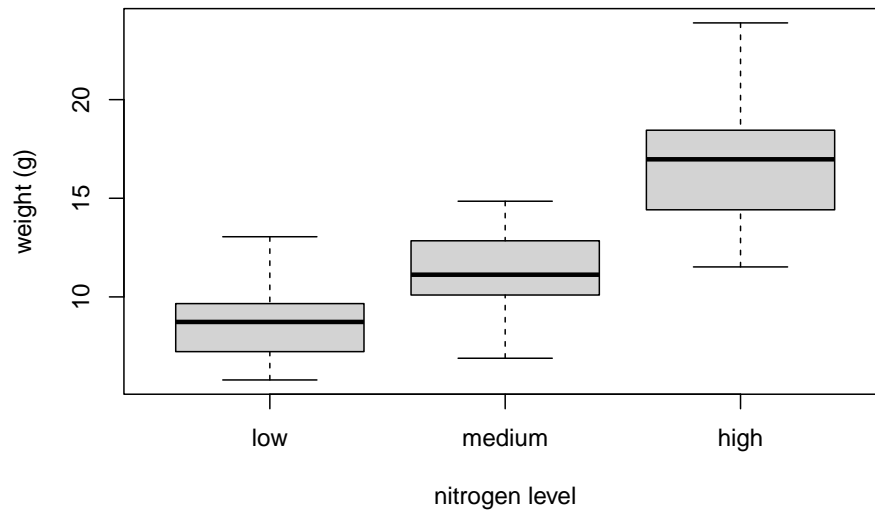

We can also group our variables by two factors in the same plot. Let's plot our `weight` variable but this time plot a separate box for each `nitrogen` and treatment (`treat`) combination.

```r
boxplot(weight ~ nitrogen * treat, data = flowers,
        ylab = "weight (g)", xlab = "nitrogen level")
```
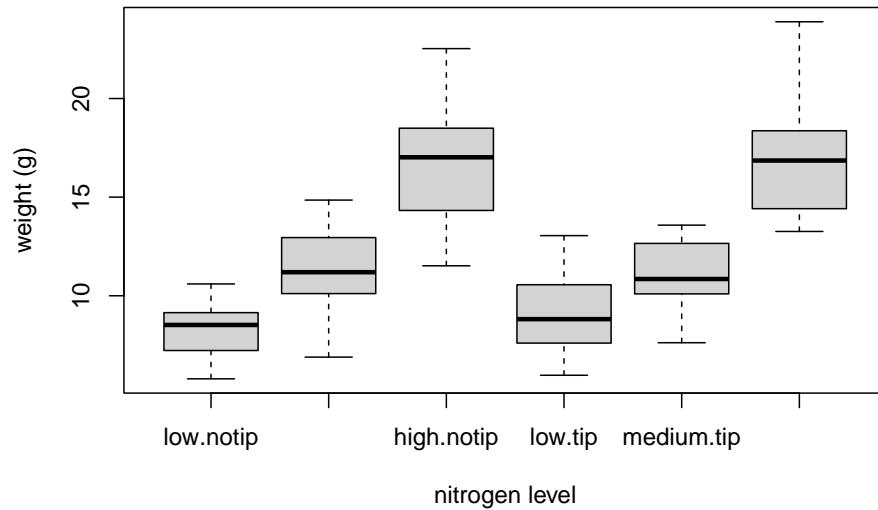

This plot looks OK, but some of the group labels are hidden as they're too long to fit on the plot. There are a couple of ways to deal with this. Perhaps the easiest is to reduce the font size of the tick mark labels in the plot so they all fit using the `cex.axis` = argument. Let's set the font size to be 30% smaller than the default with `cex.axis = 0.7`. We'll show you how to further customise plots later on in the Chapter.

```r
boxplot(weight ~ nitrogen * treat, data = flowers,
        ylab = "weight (g)", xlab = "nitrogen level",
        cex.axis = 0.7)
```
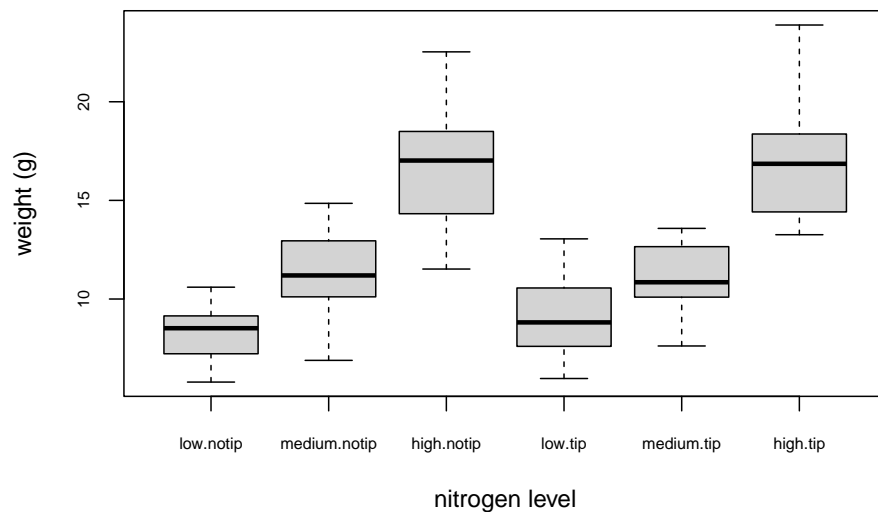

Violin plots are like a combination of a boxplot and a kernel density plot (you saw an example of a kernel density plot in the histogram section above) all rolled into one figure.

We can create a violin plot in R using the `vioplot()` function from the `vioplot` package. You'll need to first install this package using `install.packages('vioplot')` function as usual. The nice thing about the `vioplot()` function is that you use it in pretty much the same way you would use the `boxplot()` function. We'll also use the argument `col = "lightblue"` to change the fill colour to light blue.

```r
library(vioplot)
vioplot(weight ~ nitrogen, data = flowers,
        ylab = "weight (g)", xlab = "nitrogen level",
        col = "lightblue")
```
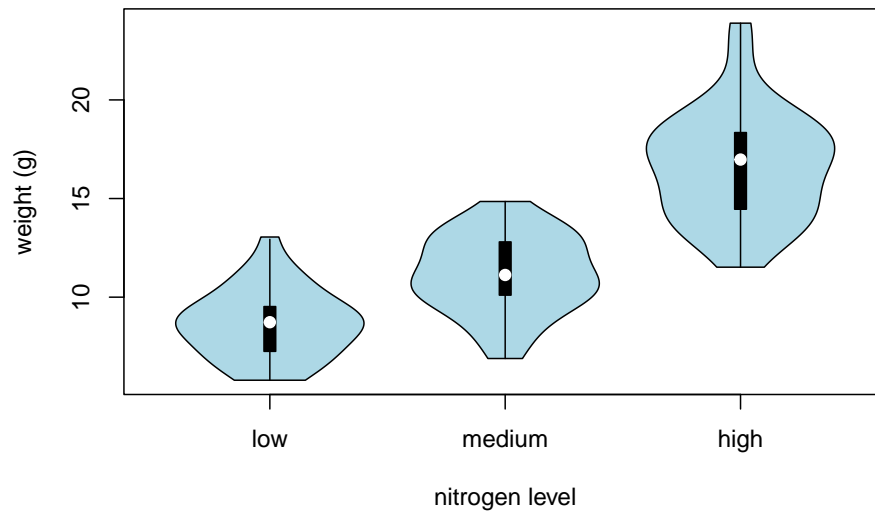

In the violin plot above we have our familiar boxplot for each `nitrogen` level but this time the median value is represented by a white circle. Plotted around each boxplot is the kernel density plot which represents the distribution of the data for each nitrogen level.

4.2.4 Dot charts

Identifying unusual observations (aka outliers) in numeric variables is extremely important as they may influence parameter estimates in your statistical model or indicate an error in your data. A really useful (if undervalued) plot to help identify outliers is the Cleveland dotplot. You can produce a dotplot in R very simply by using the `dotchart()` function.

```r
dotchart(flowers$height)
```
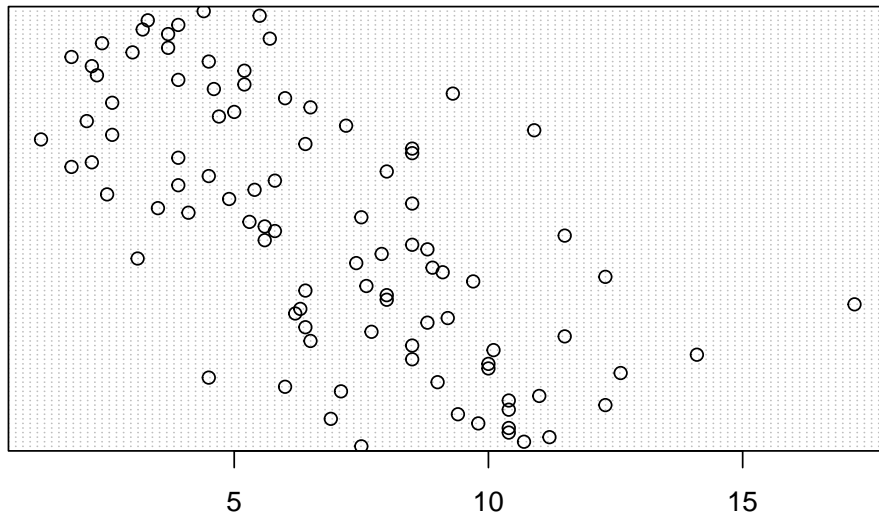
In the dotplot above the data from the `height` variable is plotted along the x axis and the data is plotted in the order it occurs in the `flowers` data frame on the y axis (values near the top of the y axis occur later in the data frame with those lower down occurring at the beginning of the data frame). In this plot we have a single value extending to the right at about 17 cm but it doesn't appear particularly large compared to the rest. An example of a dotplot with an unusual observation is given below.
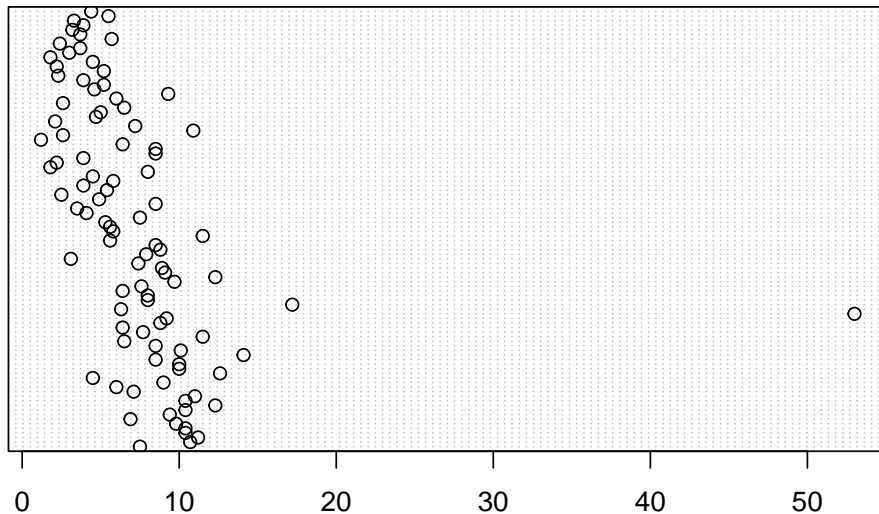
We can also group the values in our `height` variable by a factor variable such as `nitrogen` using the `groups =` argument. This is useful for identifying unusual observations within a factor level that might be obscured when looking at all the data together.

```r
dotchart(flowers$height, groups = flowers$nitrogen)
```
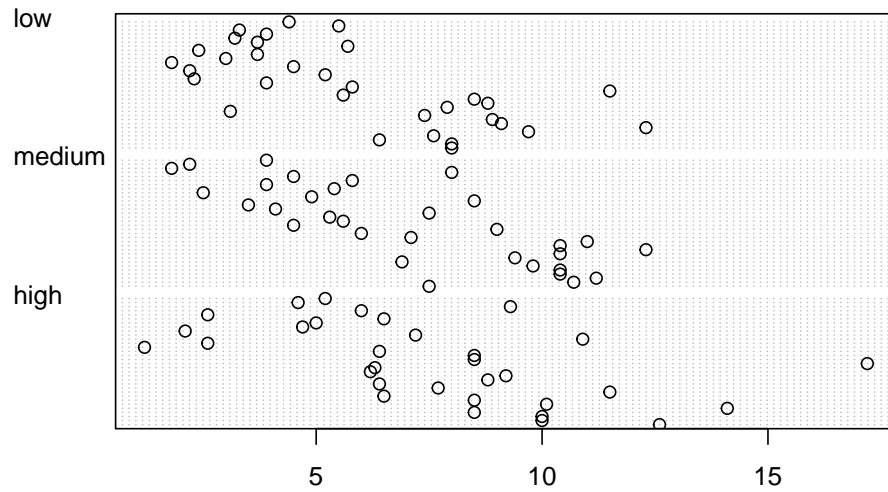
4.2.5 Pairs plots

Previously in this Chapter we used the `plot()` function to create a scatterplot to explore the relationship between two numeric variables. With datasets that contain many numeric variables, it's often handy to create multiple scatterplots to visualise relationships between all these variables. We could use the `plot()` function to create each of these plot individually, but a much easier way is to use the `pairs()` function. The `pairs()` function creates a multi-panel scatterplot (sometimes called a scatterplot matrix) which plots all combinations of variables. Let's create a multi-panel scatterplot of all of the numeric variables in our `flowers` data frame. Note, you may need to click on the 'Zoom' button in RStudio to display the plot clearly.

```
pairs(flowers[, c("height", "weight", "leafarea",
                "shootarea", "flowers")])
```
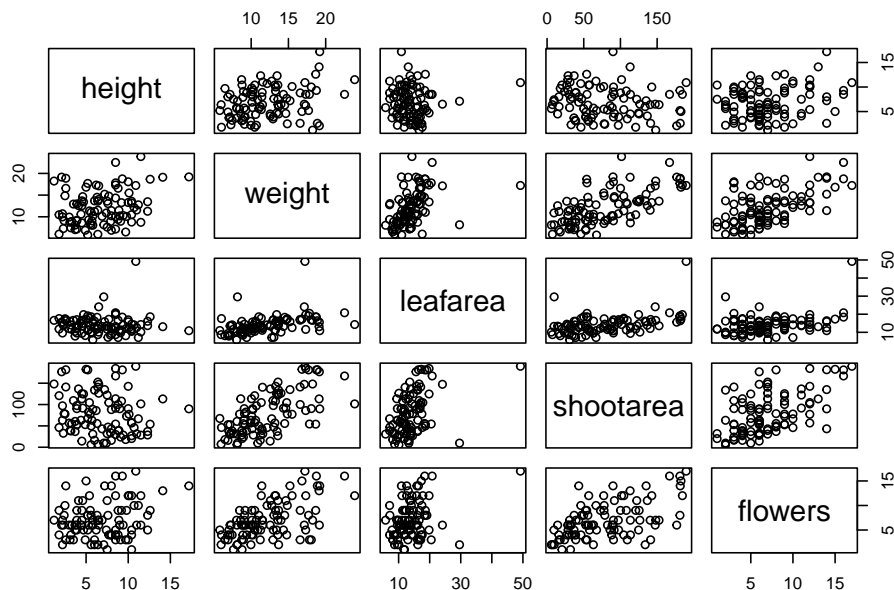
```
# or we could use the equivalent
# pairs(flowers[, 4:8])
```

Interpretation of the pairs plot takes a bit of getting used to. The panels on the diagonal give the variable names. The first row of plots displays the `height` variable on the y axis and the variables `weight`, `leafarea`, `shootarea` and `flowers` on the x axis for each of the four plots respectively. The next row of plots have `weight` on the y axis and `height`, `leafarea`, `shootarea` and `flowers` on the x axis. We interpret the rest of the rows in the same way with the last row displaying the `flowers` variable on the y axis and the other variables on the x axis. Hopefully you'll notice that the plots below the diagonal are the same plots as those above the diagonal just with the axis reversed.

We can also add additional information to each of our plots by including a panel function when we use the `pairs()` function. For example, to add a LOWESS (locally weighted scatterplot smoothing) smoother to each of the panels we just need to add the argument `panel = panel.smooth`.

```
pairs(flowers[, c("height", "weight", "leafarea",
               "shootarea", "flowers")],
      panel = panel.smooth)
```
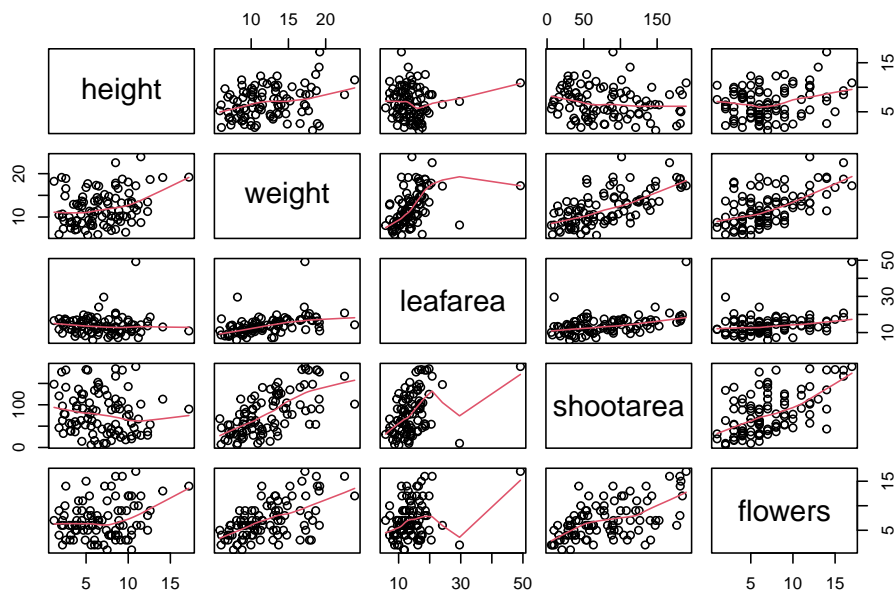

If you take a look at the help file for the `pairs()` function (`?pairs`) you'll find a few more useful panel functions in the 'Examples' section. To use these functions you'll first need to copy the code and paste it into the Console in RStudio (we go into more detail about defining functions in Chapter 7). For example, the `panel.cor()` function calculates the absolute correlation coefficient between two variables and adjusts the size of the displayed coefficient depending on the value (higher coefficient values are bigger). Don't worry if you don't understand this code just yet, for the moment we just need to know how to use it (it might be fun to try and figure it out though!).

```
panel.cor <- function(x, y, digits = 2, prefix = "", cex.cor, ...)
{
    usr <- par("usr")
    par(usr = c(0, 1, 0, 1))
    r <- abs(cor(x, y))
    txt <- format(c(r, 0.123456789), digits = digits)[1]
    txt <- paste0(prefix, txt)
    if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
    text(0.5, 0.5, txt, cex = cex.cor * r)
}
```

Once we've copied and pasted this code into the Console we can use the `panel.cor()` function to replace the plots below the diagonal with the correlation coefficient using the `lower.panel =` argument.

```
pairs(flowers[, c("height", "weight", "leafarea",
                  "shootarea", "flowers")],
              lower.panel = panel.cor)
```
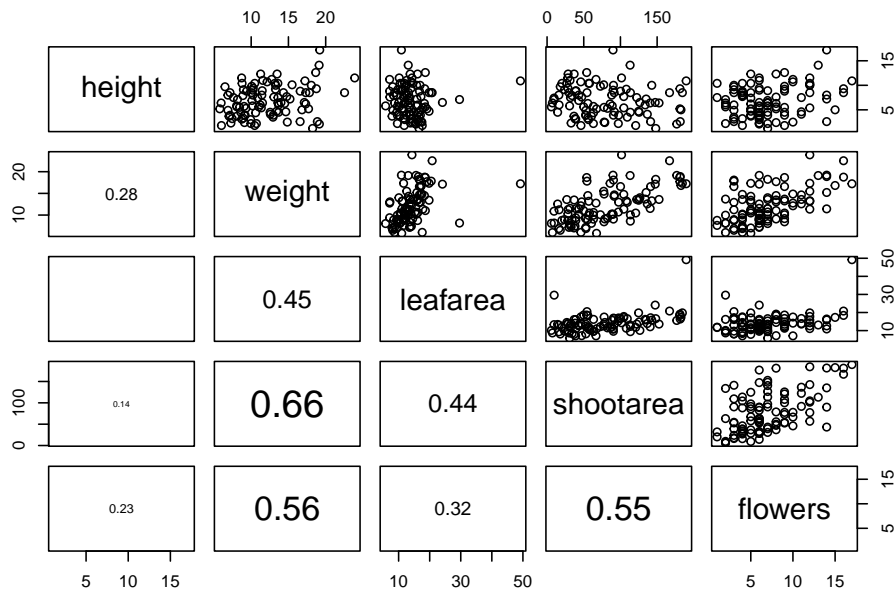

Another useful function in the 'Examples' section is the `panel.hist()` function. This function generates a histogram of each of the variables in the plot. To use it, we must again copy the code and paste it into the Console in RStudio.

```
panel.hist <- function(x, ...)
{
    usr <- par("usr")
```

```
    par(usr = c(usr[1:2], 0, 1.5) )
    h <- hist(x, plot = FALSE)
    breaks <- h$breaks; nB <- length(breaks)
    y <- h$counts; y <- y/max(y)
    rect(breaks[-nB], 0, breaks[-1], y, col = "cyan", ...)
}
```

We can then include it in our `pairs()` function to place the histograms on the diagonal set of panels using the `diag.panel = panel.hist` argument. Let's also apply the `panel.smooth()` function to the plots above the diagonal while we're at it using the `upper.panel = panel.smooth` argument.

```
pairs(flowers[, c("height", "weight", "leafarea",
                  "shootarea", "flowers")],
                lower.panel = panel.cor,
                diag.panel = panel.hist,
                upper.panel = panel.smooth)
```
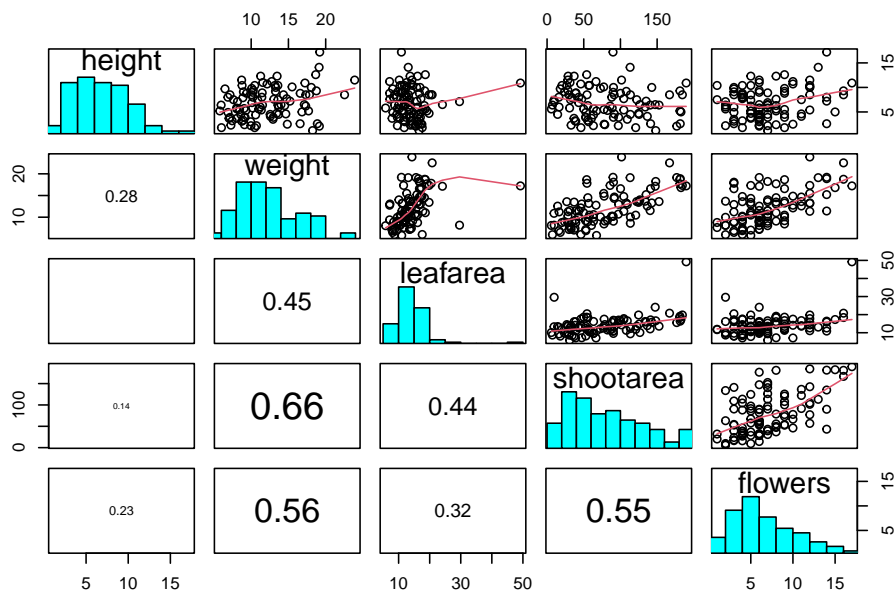

4.2.6   Coplots

When examining the relationship between two numeric variables, it is often useful to be able to determine whether a third variable is obscuring or changing any relationship. A really handy plot to use in these situations is a conditioning plot (also known as conditional scatterplot plot) which we can create in R by using the `coplot()` function. The `coplot()` function plots two variables but each plot is conditioned (`|`) by a third variable. This third variable can be either numeric or a factor. As an example, let's look at how the relationship between the number of flowers (`flowers` variable) and the

`weight` of petunia plants changes dependent on `leafarea`. Note the `coplot()` function has a `data =` argument so no need to use the `$` notation.

```r
coplot(flowers ~ weight|leafarea, data = flowers)
```
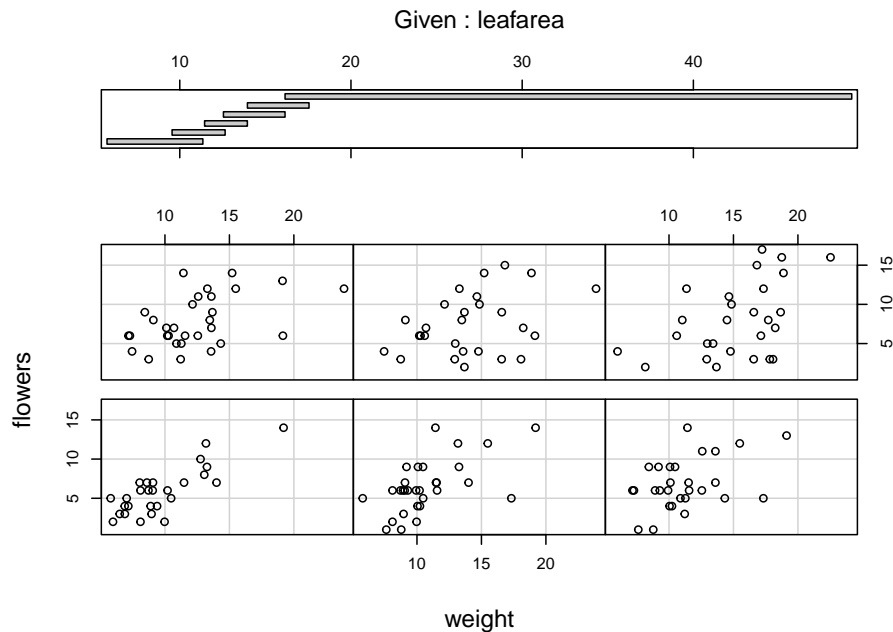

It takes a little practice to interpret coplots. The number of flowers is plotted on the y axis and the weight of plants on the x axis. The six plots show the relationship between these two variables for different ranges of leaf area. The bar plot at the top indicates the range of leaf area values for each of the plots. The panels are read from bottom left to top right along each row. For example, the bottom left panel shows the relationship between number of flowers and weight for plants with the lowest range of leaf area values (approximately 5 - 11 cm^2). The top right plot shows the relationship between flowers and weight for plants with a leaf area ranging from approximately 16 - 50 cm^2. Notice that the range of values for leaf area differs between panels and that the ranges overlap from panel to panel. The `coplot()` function does it's best to split the data up to ensure there are an adequate number of data points in each panel. If you don't want to produce plots with overlapping data in the panel you can set the `overlap =` argument to `overlap = 0`

```r
coplot(flowers ~ weight|leafarea, data = flowers, overlap = 0)
```
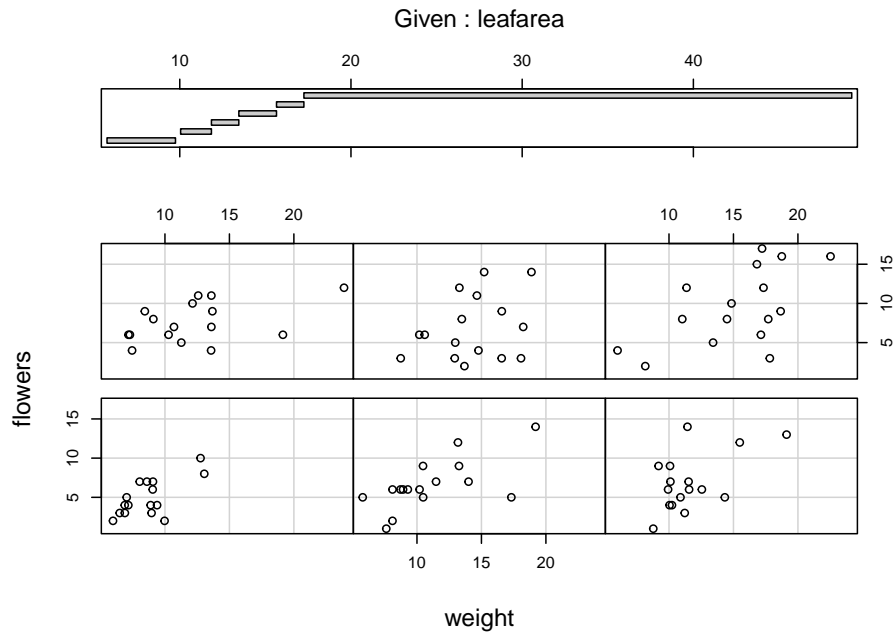
You can also use the `coplot()` function with factor conditioning variables. For example, we can examine the relationship between `flowers` and `weight` variables conditioned on the factor `nitrogen`. The bottom left plot is the relationship between `flowers` and `weight` for those plants in the `low` nitrogen treatment. The top left plot shows the same relationship but for plants in the `high` nitrogen treatment.

```
coplot(flowers ~ weight|nitrogen, data = flowers)
```
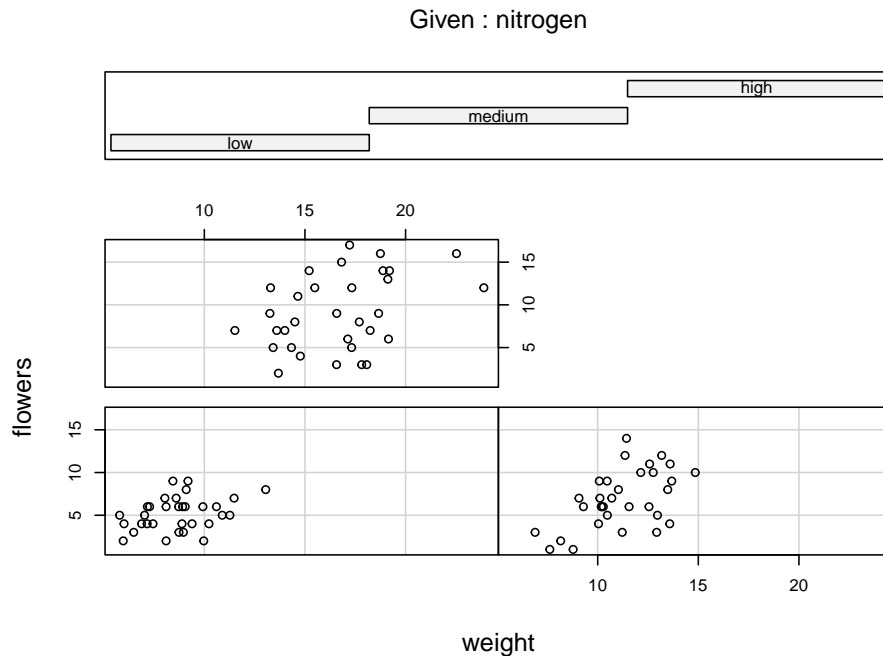

We can even use two conditioning variables (either numeric or factors). Let's look at the relationship between `flowers` and `height` but this time condition on both `nitrogen` and `treat`.

```r
coplot(flowers ~ weight|nitrogen * treat, data = flowers)
```
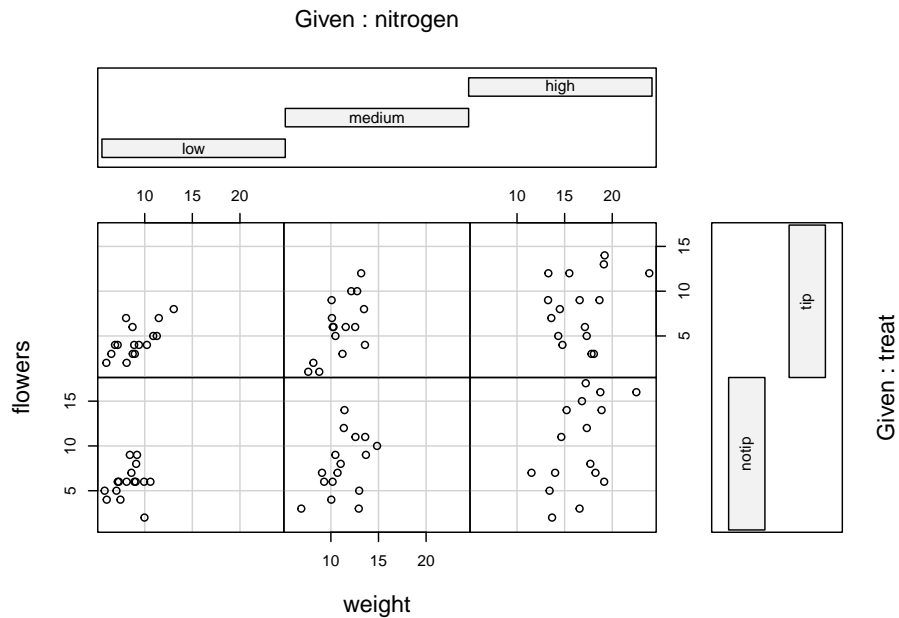

The bottom row of plots are for plants in the `notip` treatment and the top row for plants in the `tip` treatment. So the bottom left plot shows the relationship between `flowers` and `weight` for plants grown in `low` nitrogen with the `notip` treatment. The top right plot are those plants grown in `high` nitrogen with the `tip` treatment.

Similar to the `pairs()` function we can also apply functions to each of the panels using the `panel =` argument. For example, let's add a separate line of best fit from a linear model to each of the panels. Don't worry about the complicated looking code for now, this example is just included to give you an idea about some of the useful things you can do with base R plotting functions. We will introduce linear models in Chapter 6 and cover how to write functions in Chapter 7.

```r
coplot(flowers ~ weight|nitrogen * treat, data = flowers,
       panel = function(x, y, ...) {
       points(x, y, ...)
       abline(lm(y ~ x), col = "blue")})
```
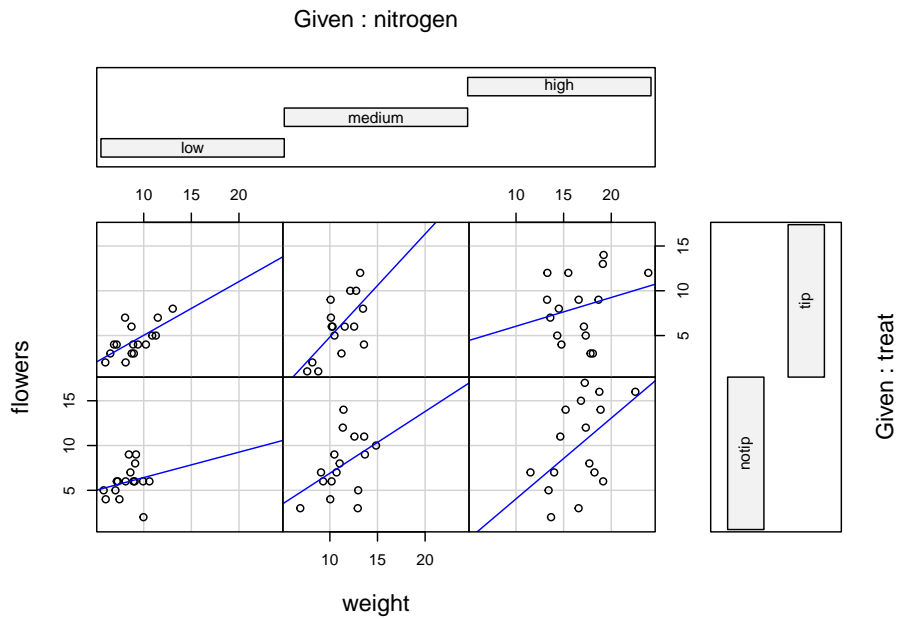
4.2.7   Lattice plots

Many of the plots we've previously created using base R graphics can also be created using functions from the `lattice` package. For example, we can recreate the frequency histogram of the `height` variable in our `flowers` data frame using the `histogram()` function. All of the plotting functions in the lattice package take the formula notation which is why we need to include `~ height` as our first argument. We also need to specify that we want a frequency histogram by using the argument `type = "count"`. Don't forget to first make the package available using `library(lattice)`.

```
library(lattice)
histogram(~ height, type = "count", data = flowers)
```
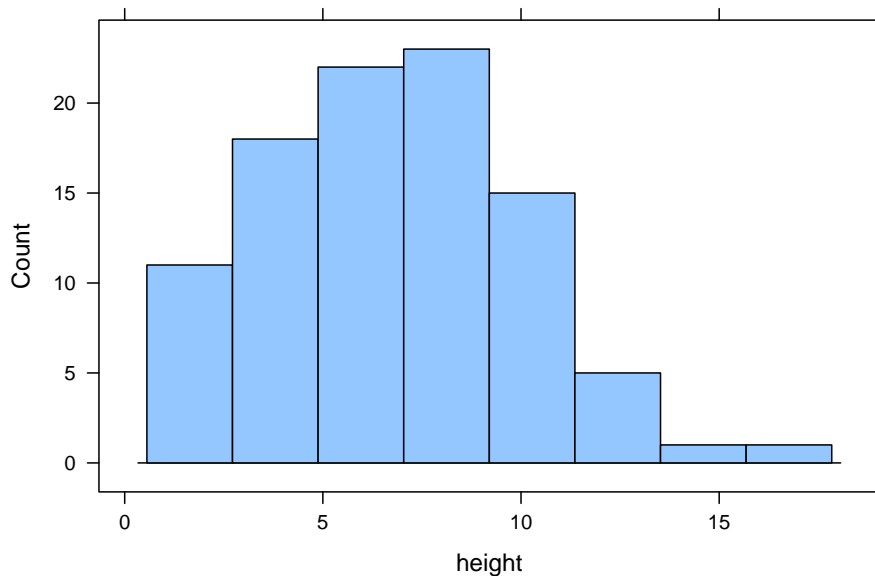
Or perhaps we would like a boxplot of our `weight` variable for each level of `nitrogen` using the `bwplot()` function.

```r
bwplot(weight ~ nitrogen, data = flowers)
```
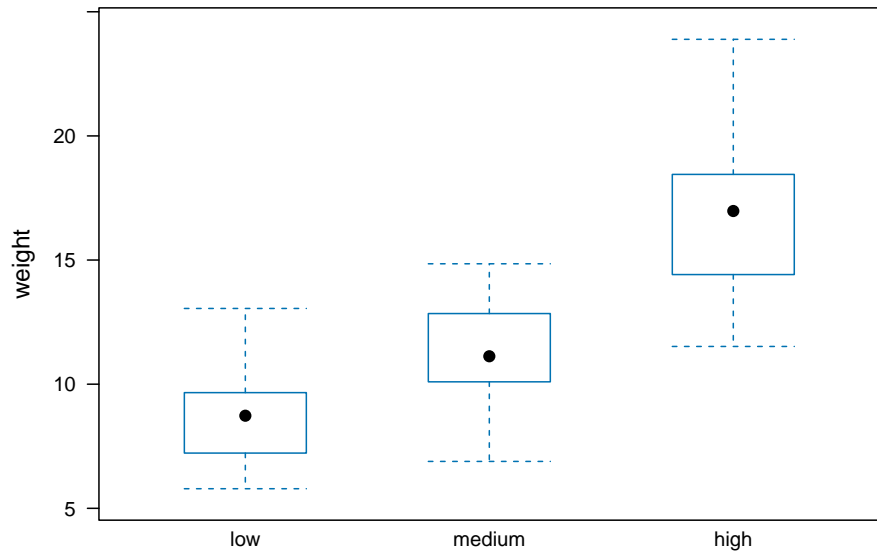

A (non-exhaustive) list of lattice functions and their base R equivalents is given in the table below.

Graph type	lattice function	Base R function
scatterplot	`xyplot()`	`plot()`
frequency histogram	`histogram(type = "count")`	`hist()`
boxplot	`bwplot()`	`boxplot()`
Cleveland dotplot	`dotplot()`	`dotchart()`
scatterplot matrix	`splom()`	`pairs()`
conditioning plot	`xyplot(y ~ x | z)`	`coplot()`

Where lattice plots really come into their own is when we want to plot graphs in multiple panels. For example, let's plot a histogram of our `height` variable again but this time create a separate histogram for each `nitrogen` level. We do this by including the `|` (pipe) symbol which we read as *'height conditional on nitrogen level'*. Also notice that the axis scales are the same for each of the panels to aid comparison which is the default for lattice plots.

```r
histogram(~ height | nitrogen, type = "count", data = flowers)
```
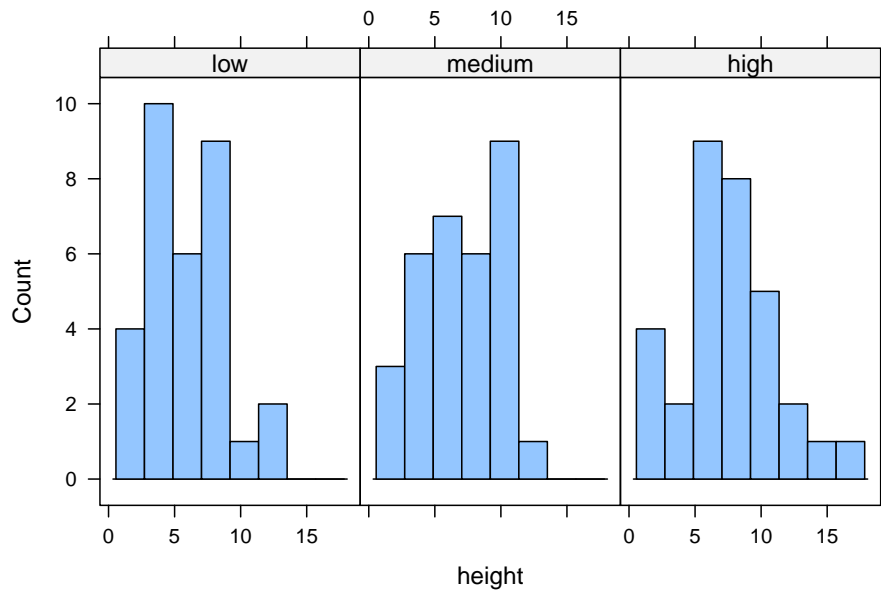
If we want to change the layout of our plots we can use the `layout =` argument. Perhaps we prefer all of the graphs to be stacked one on top of the other in which case we would use `layout = c(1, 3)` to specify 1 column and 3 rows of plots.

```
histogram(~ height | nitrogen, type = "count",
          layout = c(1, 3), data = flowers)
```
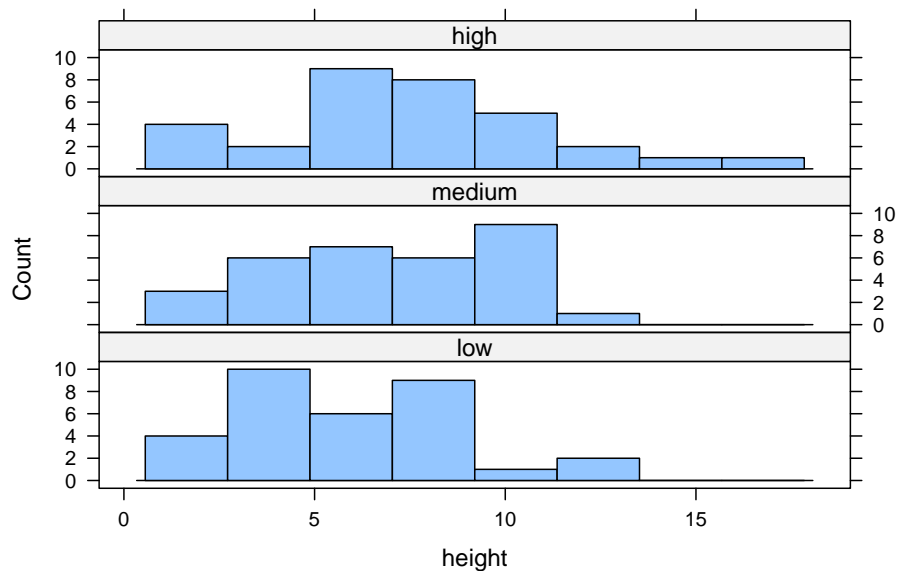

We can also easily create conditional boxplots using the same logic.

```
bwplot(weight ~ nitrogen | block, data = flowers)
```
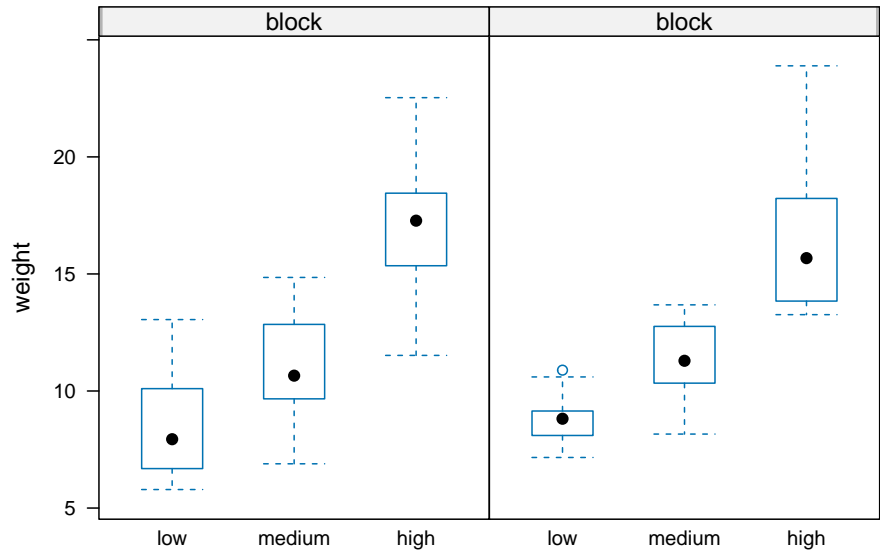
Notice in the plot above that the block names don't seem to be displayed (actually they are, the're just represented as orange vertical bars in the panel name). The reason for this is that our `block` variable is an integer variable (you can check this with `class(flowers$block)`) as the blocks were coded as either a 1 or a 2 in the original dataset that we imported into R. We can change this by creating a new variable in our data frame and use the `factor()` function to convert `block` to a factor (`flowers$Fblock <- factor(flowers$block)`) and then use the `Fblock` variable as the conditioning variable. Or we can just change `block` to be a factor *'on-the-fly'* when we use it in the `bwplot()` function. Note, this doesn't change the `block` variable in the `flowers` data frame, just temporarily when we use the `bwplot()` function.

```
bwplot(weight ~ nitrogen | factor(block), data = flowers)
```
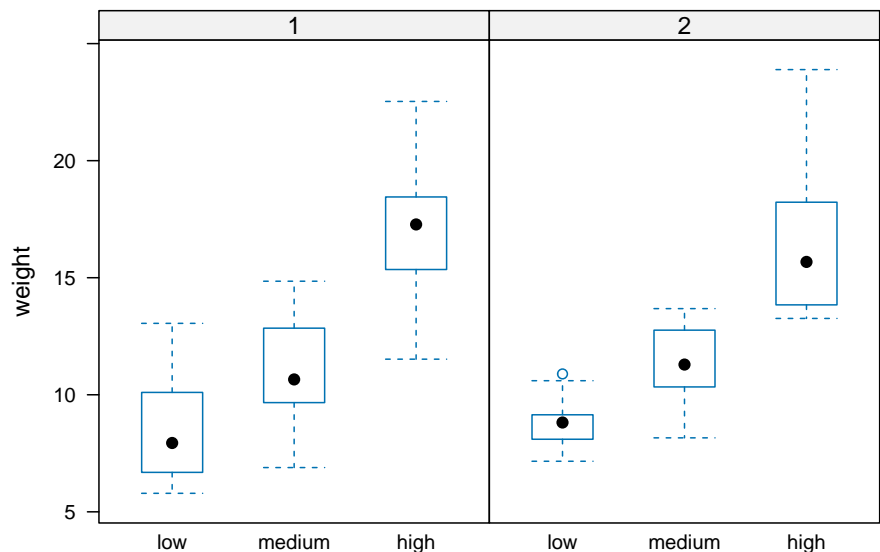

We can also include more than one conditioning variable. For example, let's create a scatter plot of `height` against `weight` for each level of `nitrogen` and `treat`. To do this we'll use the lattice `xyplot()` function.

```
xyplot(height ~ weight | nitrogen * treat, data = flowers)
```
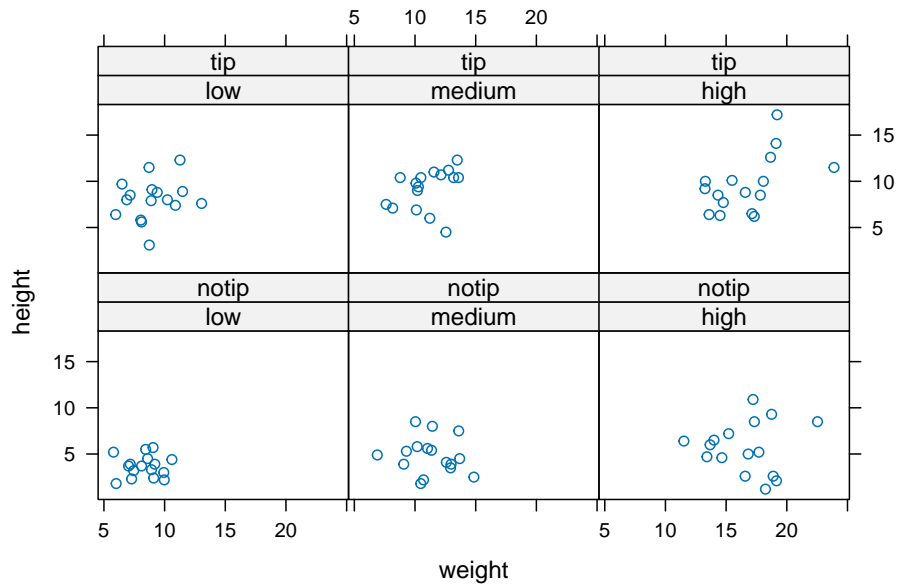

If we want to highlight which data points come from `block 1` and which from `block 2` by automatically changing the plotting symbols we can use the `groups =` argument. We'll also include the argument `auto.key = TRUE` to automatically generate a legend.

```
xyplot(flowers ~ shootarea | nitrogen * treat,
       groups = block, auto.key = TRUE, data = flowers)
```
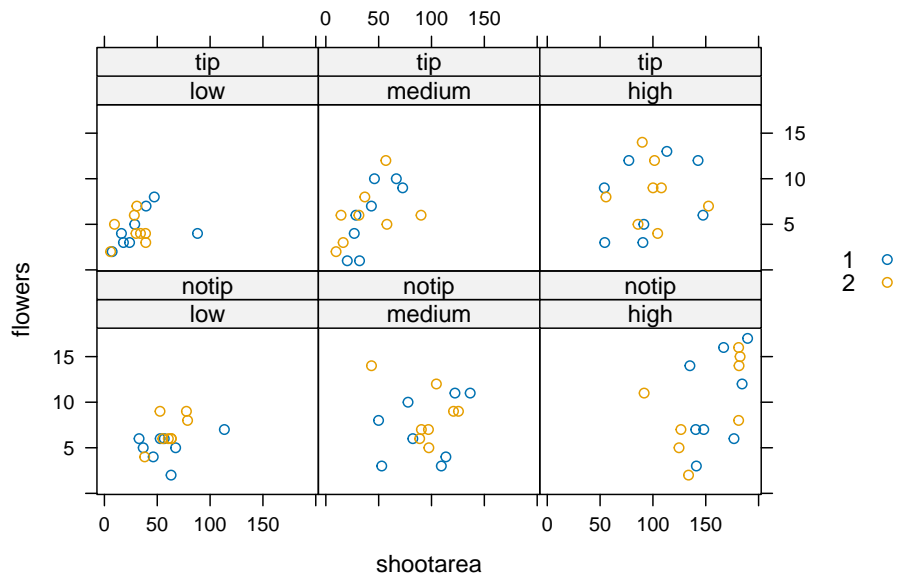

Hopefully, you're getting the idea that we can create really informative exploratory plots quite easily using either base R or lattice graphics. Which one you use is entirely up to you (that's the beauty of using R, you get to choose) and we happily mix and match to suit our needs. In the next section we cover how to customise your base R plots to get them to look exactly how you want.

4.3 Customising plots

All of the plots we've created so far in this Chapter are more than suitable for exploring your data. If however, you'd like to make them a little prettier (for your thesis, publication or even your own amusement) you'll need to invest some time learning how to customise your plots. The good news is that the base R graphics system allows you to change almost any aspect of your plot. There are however a couple of things to bear in mind. Firstly, although many of the approaches we introduce in this section will work with most base R plotting functions, there's no true consistency between functions. What works with the `plot()` function isn't guaranteed to necessarily work with the `boxplot()` function. This can be a little frustrating to begin with but gets easier the more experience you gain. If you crave a little more consistency take a look at Chapter 5 where we introduce the excellent `ggplot2` package. Secondly, when you start customising plots you're confronted with a huge number of options and arguments to try and remember. This isn't necessarily a bad thing as this is what makes base R graphics so flexible but it's a lot to take in. Often a quick Google or peek at the relevant help pages will jog your memory. Thirdly, learning how to customise plots in base R isn't just about what code you need to use, it's also about learning the process of building a plot. We often start with a basic layout of our plot and then add layers of complexity until we achieve the desired results. This requires a little experience (and trial and error), but again becomes easier with practice. Lastly, this section covers the basics of how to customise base R graphics and most (if not all) of these approaches will not work for plots created with the `lattice` graphics system.

4.3.1 Customising with arguments

Let's return to the basic plot we made previously in this Chapter. This was a simple scatterplot to examine the relationship between the `shootarea` and `weight` variables in the `flowers` data frame.

```
plot(flowers$weight, flowers$shootarea)
```
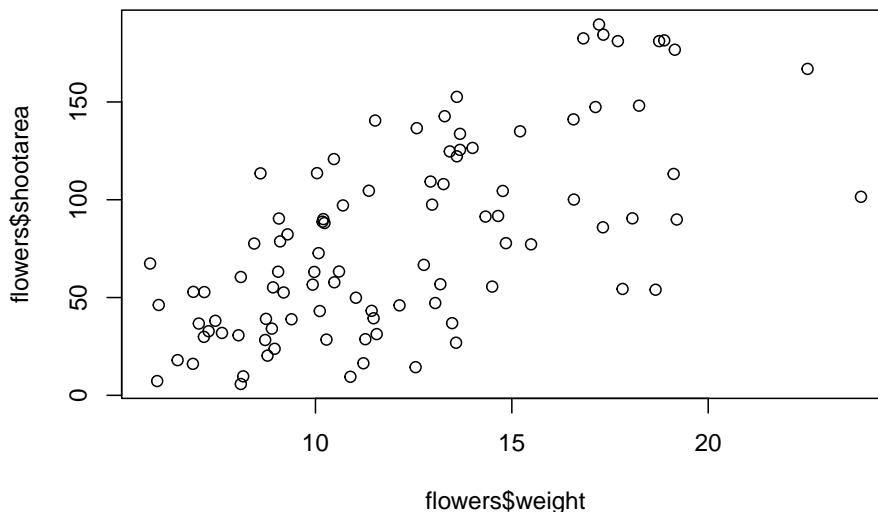
Whilst this plot is adequate for data exploration it's not going to cut the mustard if we want to share it with others. At the very least it could do with a better set of axes labels, more informative axes scales and some nicer plotting symbols.

Let's start with the axis labels. To add labels to the x and y axes we use the corresponding `ylab =` and `xlab =` arguments in the `plot()` function. Both of these arguments need character strings as values.

```r
plot(flowers$weight, flowers$shootarea,
     xlab = "weight (g)",
     ylab = "shoot area (cm2)")
```
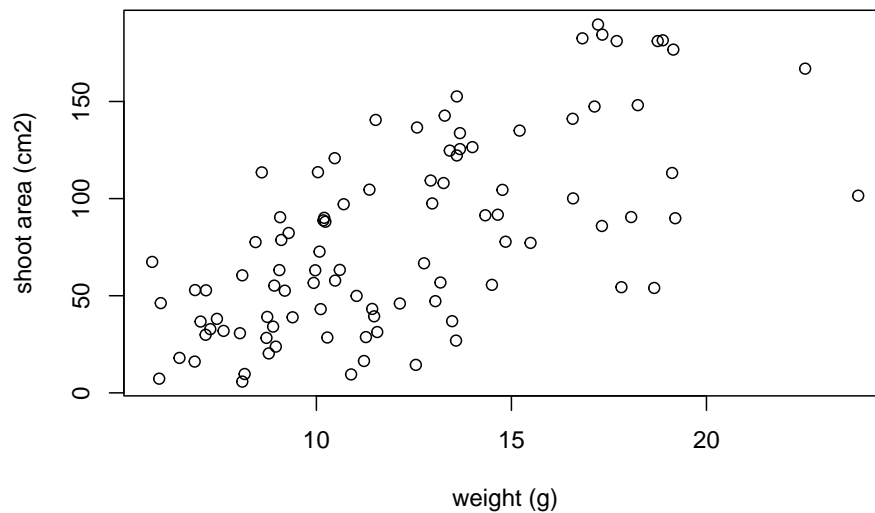

OK, that looks a little better but the units (`cm2`) looks a little ugly as we should format the `2` as a superscript. To convert to a superscript we need to use a combination of the `expression()` and `paste()` functions. The `expression()` function allows us to format the superscript (and other mathematical expressions - see `?plotmath` for more details) with the `^` symbol and the `paste()` function pastes together the elements `"shoot area (cm"^"2"` and `)` to create our axis label.

```r
plot(flowers$weight, flowers$shootarea,
     xlab = "weight (g)",
     ylab = expression(paste("shoot area (cm"^"2",")")))
```
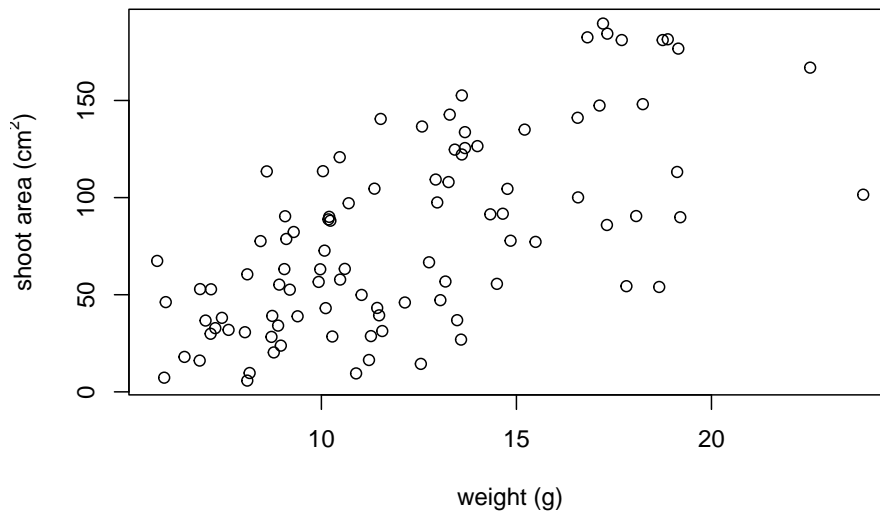
But now we have a new problem, the very top of the y axis label gets cut off. To remedy this we need to adjust the plot margins using the `par()` function and the `mar =` argument before we plot the graph. The `par()` function is the main function for setting graphical parameters in base R and the `mar =` argument sets the size of the margins that surround the plot. You can adjust the size of the margins using the notation `par(mar = c(bottom, left, top, right))` where the arguments `bottom`, `left`, `top` and `right` are the size of the corresponding margins. By default R sets these margins as `mar = c(5.1, 4.1, 4.1, 2.1)` with these numbers specifying the number of lines in each margin. Let's increase the size of the left margin a little bit and decrease the size of the right margin by a smidge.

```r
par(mar = c(4.1, 4.4, 4.1, 1.9))
plot(flowers$weight, flowers$shootarea,
      xlab = "weight (g)",
      ylab = expression(paste("shoot area (cm"^"2",")")))
```
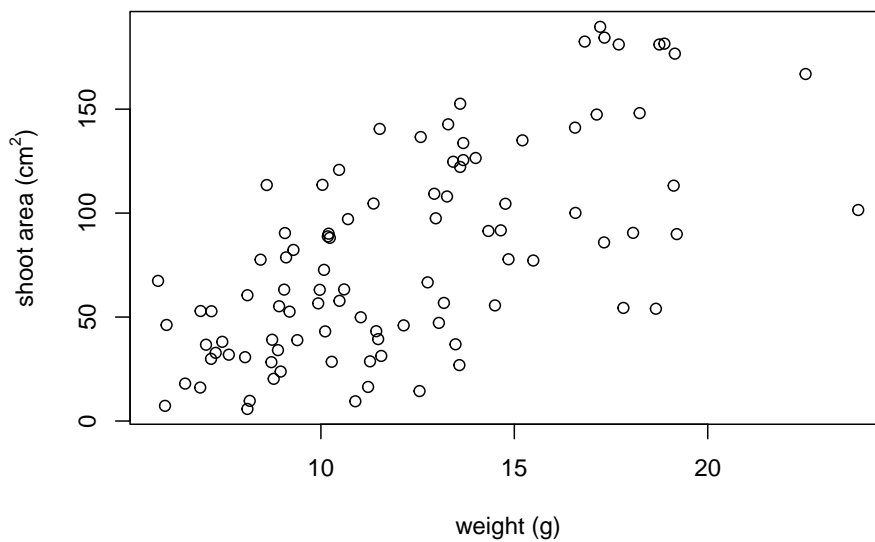

That looks better. Now let's increase the range of our axes scales so we have a bit of space above and to the right of the data points. To do this we need to supply a minimum

and maximum value using the `c()` function to the `xlim =` and `ylim =` arguments. We'll
set the x axis scale to run from 0 to 30 and the range of the y axis scale from 0 to 200.

```r
par(mar = c(4.1, 4.4, 4.1, 1.9))
plot(flowers$weight, flowers$shootarea,
     xlab = "weight (g)",
     ylab = expression(paste("shoot area (cm"^"2",")")),
     xlim = c(0, 30), ylim = c(0, 200))
```
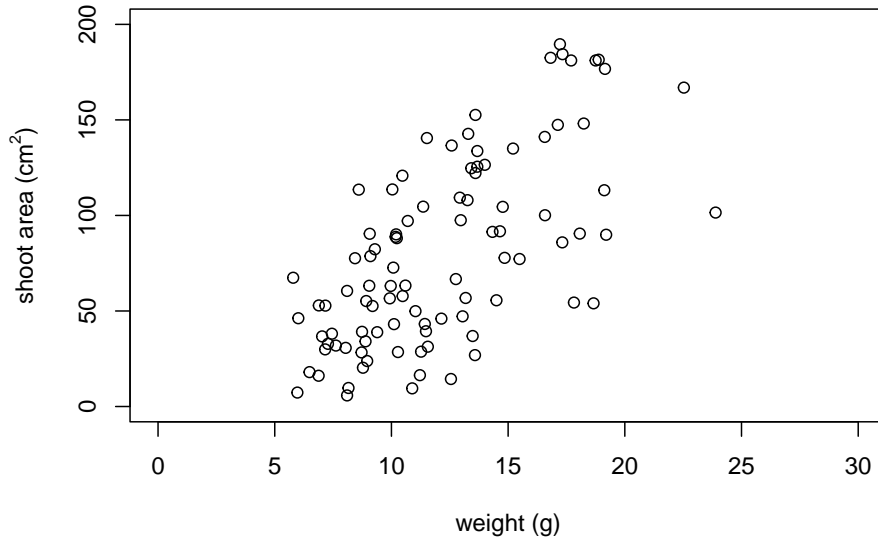

And while we're at it let's remove the annoying box all the way around the plot to just
leave the y and x axes using the `bty = "l"` argument.

```r
par(mar = c(4.1, 4.4, 4.1, 1.9))
plot(flowers$weight, flowers$shootarea,
     xlab = "weight (g)",
     ylab = expression(paste("shoot area (cm"^"2",")")),
     xlim = c(0, 30), ylim = c(0, 200), bty = "l")
```
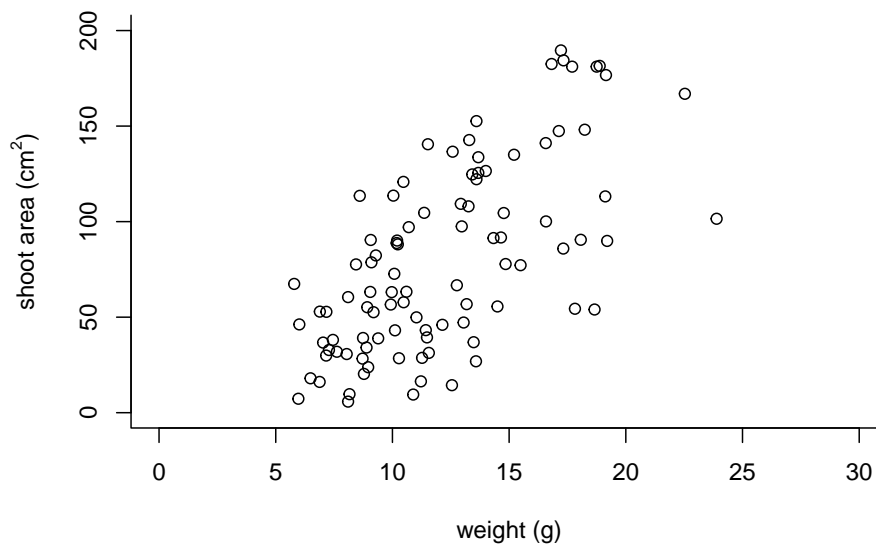
OK, that's looking a lot better already after only a few adjustments. One of the things that we still don't like is that by default the x and y axes do not intersect at the origin (0, 0) and both axes extend beyond the maximum value of the scale by a little bit. We can change this by setting the `xaxs = "i"` and `yaxs = "i"` arguments when we use the `par()` function. While we're about it let's also rotate the y axis tick mark labels so they read horizontally using by setting the `las = 1` argument in the `plot()` function and make them a tad smaller with the `cex.axis =` argument. The `cex.axis =` argument requires a number giving the amount by which the text will be magnified (or shrunk) relative to the default value of 1. We'll choose `0.8` making our text 20% smaller. We can also make the tick marks just a little shorter by setting `tcl = -0.2`. This value needs to be negative as we want the tick marks to be outside the plotting region (see what happens if you set it to `tcl = 0.2`).

```r
par(mar = c(4.1, 4.4, 4.1, 1.9), xaxs = "i", yaxs = "i")
plot(flowers$weight, flowers$shootarea,
     xlab = "weight (g)",
     ylab = expression(paste("shoot area (cm"^"2",")")),
     xlim = c(0, 30), ylim = c(0, 200), bty = "l",
     las = 1, cex.axis = 0.8, tcl  = -0.2)
```
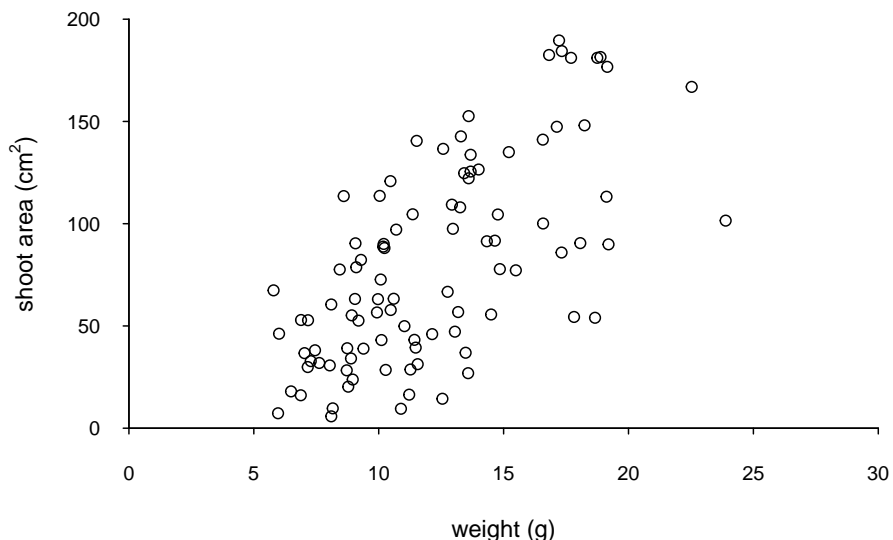
We can also change the type of plotting symbol, the colour of the symbol and the size of the symbol using the `pch =`, `col =` and `cex =` arguments respectively. The `pch =` argument takes an integer value between 0 and 25 to define the type of plotting symbol. Symbols 0 to 14 are open symbols, 15 to 20 are filled symbols and 21 to 25 are symbols where you can specify a different fill colour and outside line colour. Here's a summary table displaying the value and corresponding symbol type.

The `col =` argument changes the colour of the plotting symbols. This argument can either take an integer value to specify the colour or a character string giving the colour name. For example, `col = "red"` changes the plotting symbol to red. To see a list of all 657 preset colours available in base R use the `colours()` function (you can also use `colors()`) or perhaps even easier see this link. More colour options are available with other packages (see the excellent `RColorBrewer` package) or you can even 'mix' your own colours using the `colorRamp()` function (see `?colorRamp` for more details).

The `cex =` argument allow you to change the size of the plotting symbol. This argument works in the same way as the other `cex` arguments we've already seen (i.e. `cex.axis`) and requires a numeric value to indicate the proportional increase or decrease in size relative to the default value of 1.

Let's change the plotting symbol to a filled circle (16), the colour of the symbol to "dodgerblue1" and decrease the size of the symbol by 10%.

```r
par(mar = c(4.1, 4.4, 4.1, 1.9), xaxs = "i", yaxs = "i")
plot(flowers$weight, flowers$shootarea,
     xlab = "weight (g)",
     ylab = expression(paste("shoot area (cm"^"2",")")),
     xlim = c(0, 30), ylim = c(0, 200), bty = "l",
     las = 1, cex.axis = 0.8, tcl = -0.2,
     pch = 16, col = "dodgerblue1", cex = 0.9)
```
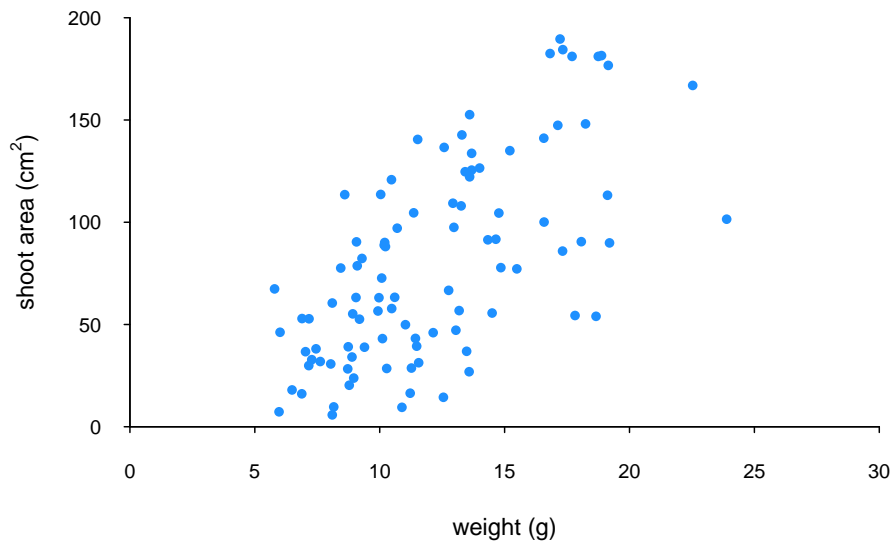

The last thing we'll do is add a text label to the plot so we can identify it. Perhaps this plot will be one of a series of plots we want to include in the same figure (see the section on plotting multiple graphs to see how to do this) so it would be nice to be able to refer to it in our figure title. To do this we'll use the `text()` function to add a capital 'A' to the top right of the plot. The `text()` function needs an `x =` and a `y =` coordinate to position the text, a `label =` for the text and we can use the `cex =` argument again to change the size of the text.

```r
par(mar = c(4.1, 4.4, 4.1, 1.9), xaxs = "i", yaxs = "i")
plot(flowers$weight, flowers$shootarea,
     xlab = "weight (g)",
     ylab = expression(paste("shoot area (cm"^"2",")")),
     xlim = c(0, 30), ylim = c(0, 200), bty = "l",
     las = 1, cex.axis = 0.8, tcl = -0.2,
     pch = 16, col = "dodgerblue1", cex = 0.9)
text(x = 28, y = 190, label = "A", cex = 2)
```
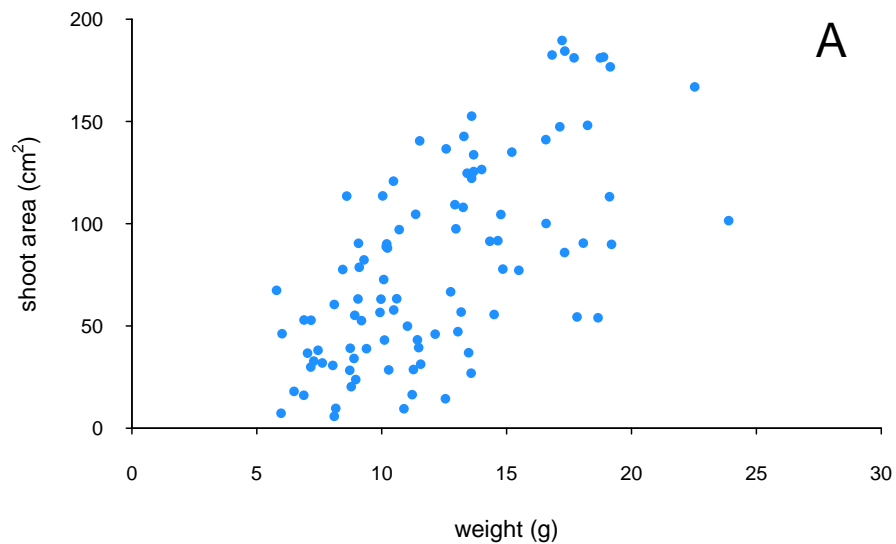
We think our plot now looks pretty good so we'll stop here! There are, however, a multitude of other arguments which you can play around with to change the look of your plots. The best place to quickly look for more information is the help page associated with the `par()` function (`?par`) or just do a quick Google search. Here's a table of the more commonly used arguments.

Argument	Description
adj	controls justification of the text (0 left justified, 0.5 centered, 1 right justified)
bg	specifies the background colour of the plot (i.e. : `bg = "red"`, `bg = "blue"`)
bty	controls the type of box drawn around the plot, values include: `"o"`, `"l"`, `"7"`, `"c"`, `"u"` , `"]"` (the box looks like the corresponding character); if `bty = "n"` the box is not drawn
cex	controls the size of text and symbols in the plotting area with respect to the default value of 1. Similar commands include: `cex.axis` controls the numbers on the axes, `cex.lab` numbers on the axis labels, `cex.main` the title and `cex.sub` the sub-title
col	controls the colour of symbols; additional argument include: `col.axis`, `col.lab`, `col.main`, `col.sub`
font	an integer controlling the style of text (1: normal, 2: bold, 3: italics, 4: bold italics); other argument include `font.axis`, `font.lab`, `font.main`, `font.sub`
las	an integer which controls the orientation of the axis labels (0: parallel to the axes, 1: horizontal, 2: perpendicular to the axes, 3: vertical)
lty	controls the line style, can be an integer (1: solid, 2: dashed, 3: dotted, 4: dotdash, 5: longdash, 6: twodash)

Argument	Description
`lwd`	a numeric which controls the width of lines. Works as per `cex`
`pch`	controls the type of symbol, either an integer between 0 and 25, or any single character within quotes `" "`
`ps`	an integer which controls the size in points of texts and symbols
`pty`	a character which specifies the type of the plotting region, "s": square, "m": maximal
`tck`	a value which specifies the length of tick marks on the axes as a fraction of the width or height of the plot; if `tck = 1` a grid is drawn
`tcl`	a value which specifies the length of tick marks on the axes as a fraction of the height of a line of text (by default `tcl = -0.5`)

4.3.2 Building plots

For even more control over how your plot looks we can build our plot up in layers, customising each step as we go along. For example, perhaps we want to create a plot of `shootarea` and `weight` as we did before but this time we want to change the symbol colours of our data points depending on what level of `nitrogen` the plants were exposed to. The general approach is to use the high level plotting function `plot()` to create the general plot (axes, axes labels etc) but without the data points by including the `type = "n"` argument. We then use the low level function `points()` to add the plotting symbols for each `nitrogen` level separately choosing a different colour for each set of points. Let's go through this approach a step at a time. First we'll make the plot but suppress plotting the data using the `type = "n"` argument in the `plot()` function.

```
par(mar = c(4.1, 4.4, 4.1, 1.9), xaxs = "i", yaxs = "i")
plot(flowers$weight, flowers$shootarea,
     type = "n",
     xlab = "weight (g)",
     ylab = expression(paste("shoot area (cm"^"2",")")),
     xlim = c(0, 30), ylim = c(0, 200), bty = "l",
     las = 1, cex.axis = 0.8, tcl = -0.2)
```
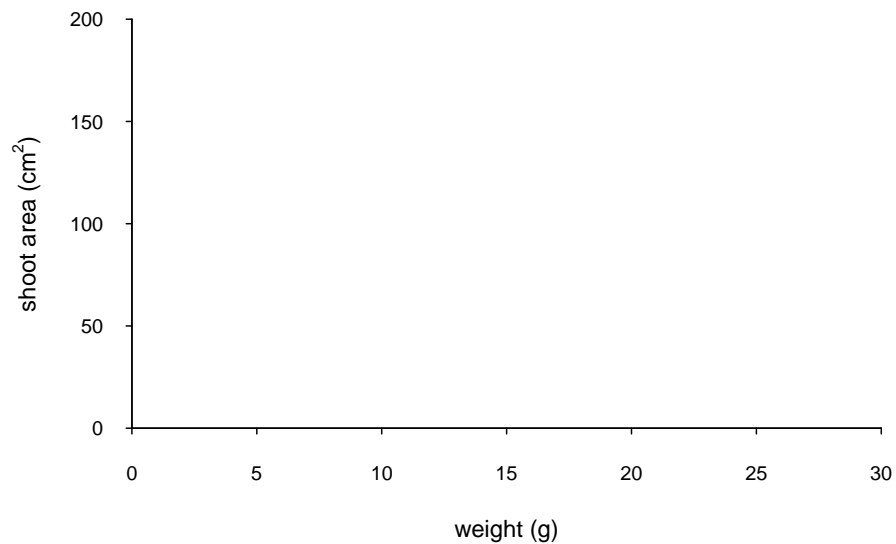
We can now use the `points()` function in combination with our square bracket `[ ]` skills to only select those data from the `low` level of `nitrogen`. Whilst using the `points()` function we can also set the symbol type and the symbol colour using the `pch =` and `col = ` arguments.

```r
par(mar = c(4.1, 4.4, 4.1, 1.9), xaxs = "i", yaxs = "i")
plot(flowers$weight, flowers$shootarea,
      type = "n",
      xlab = "weight (g)",
      ylab = expression(paste("shoot area (cm"^"2",")")),
      xlim = c(0, 30), ylim = c(0, 200), bty = "l",
      las = 1, cex.axis = 0.8, tcl = -0.2)
points(x = flowers$weight[flowers$nitrogen == "low"],
      y = flowers$shootarea[flowers$nitrogen == "low"],
      pch = 16, col = "deepskyblue")
```
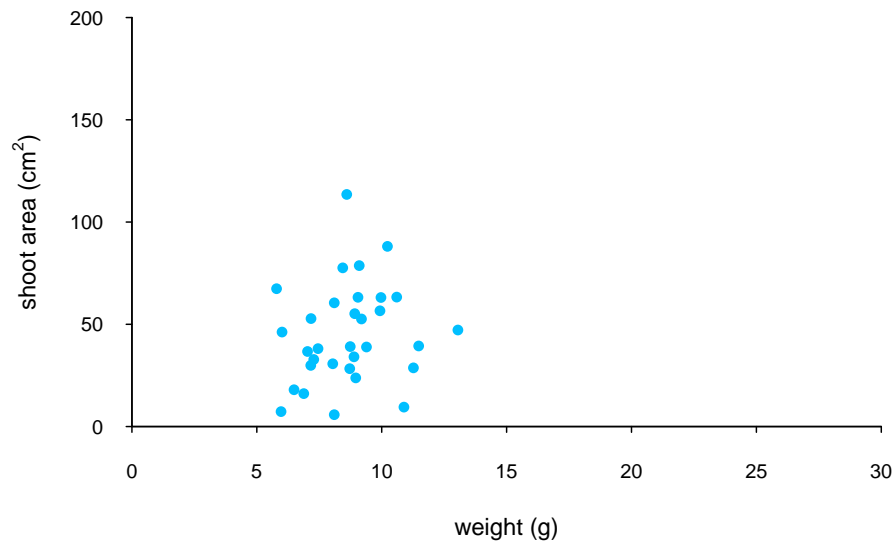
We can now use the `points()` function again to plot data for the `medium` level of nitrogen and change the symbol colour to something different. Notice that we do not reuse the `plot()` function here as we are just using the low level function `points()` to add data points to the existing plot.

```
points(x = flowers$weight[flowers$nitrogen == "medium"],
       y = flowers$shootarea[flowers$nitrogen == "medium"],
       pch = 16, col = "yellowgreen")
```
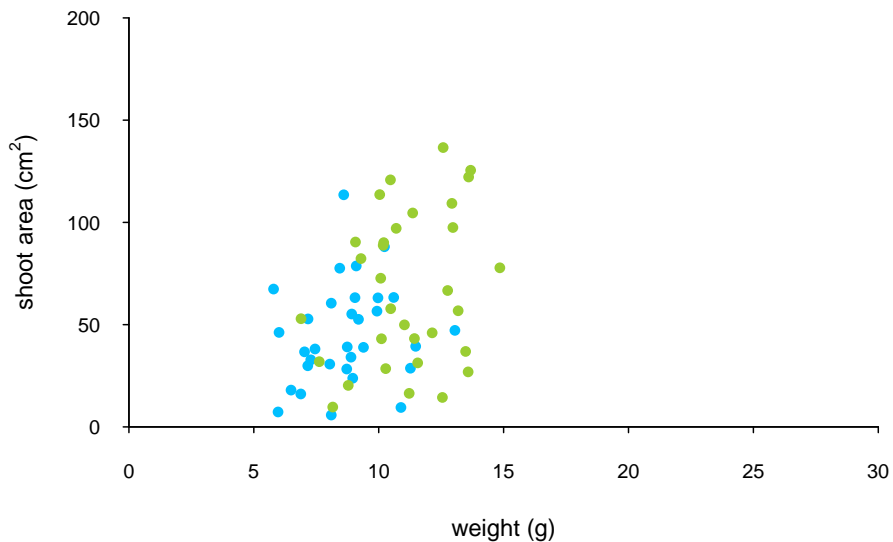

And finally to add the `high` level of `nitrogen` data points to the plot and add our text label ('A') to the plot as before.

```
points(x = flowers$weight[flowers$nitrogen == "high"],
       y = flowers$shootarea[flowers$nitrogen == "high"],
       pch = 16, col = "deeppink3")
text(x = 28, y = 190, label = "A", cex = 2)
```
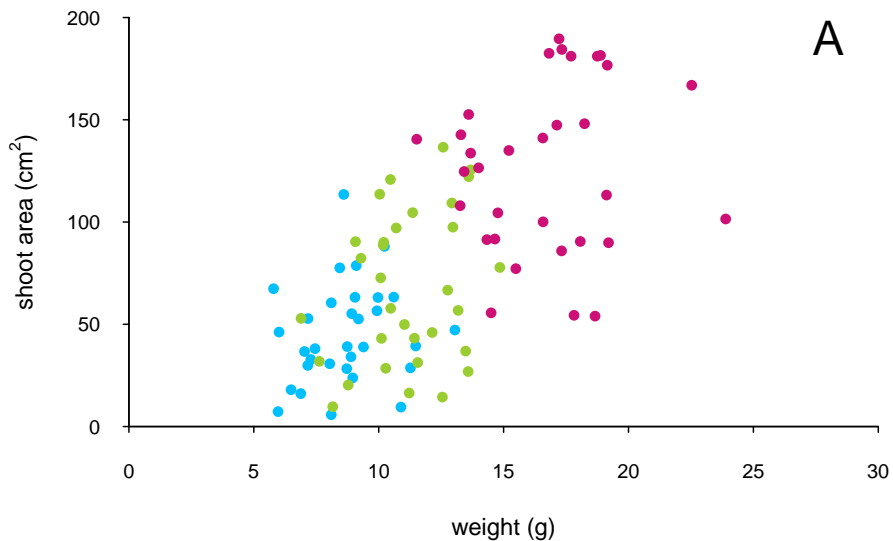
The only thing left to do is to add a legend to the plot to let your reader know what `nitrogen` level each colour corresponds to. We'll use another low level function,`legend()` to do this. The `legend()` function requires us to provide the x and y coordinates to specify the position of the top left of the legend in the plot, a vector of colours, symbol types and labels to use in the legend. The `bty = "n"` argument stops a border being drawn around the legend and the `title =` argument gives the legend a title.

```r
leg_cols <- c("deepskyblue", "yellowgreen", "deeppink3")
leg_sym <- c(16, 16, 16)
leg_lab <- c("low", "medium", "high")

legend(x = 1, y = 200, col = leg_cols, pch = leg_sym,
       legend = leg_lab, bty = "n",
       title = "Nitrogen level")
```
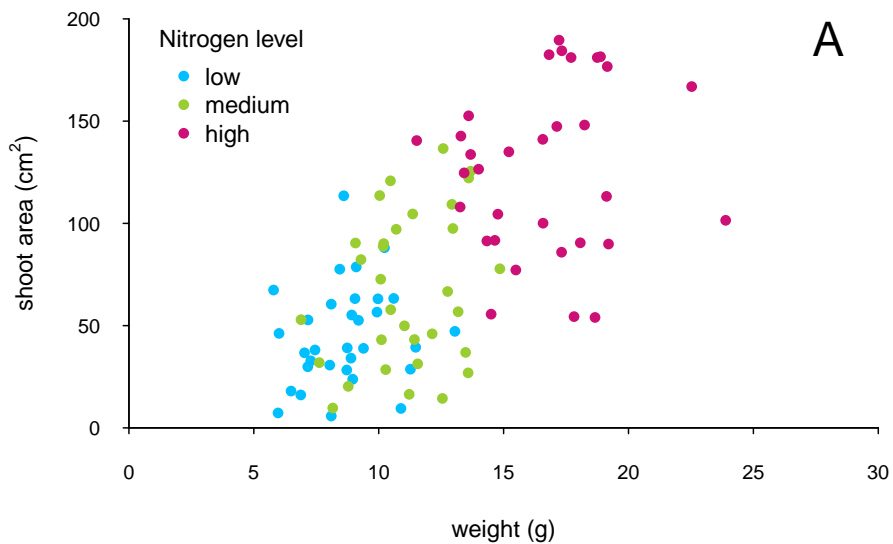

If you want to see all the code together.

```r
par(mar = c(4.1, 4.4, 4.1, 1.9), xaxs="i", yaxs="i")
plot(flowers$weight, flowers$shootarea,
     type = "n",
     xlab = "weight (g)",
     ylab = expression(paste("shoot area (cm"^"2",")")),
     xlim = c(0, 30), ylim = c(0, 200), bty = "l",
     las = 1, cex.axis = 0.8, tcl = -0.2)
points(x = flowers$weight[flowers$nitrogen == "low"],
       y = flowers$shootarea[flowers$nitrogen == "low"],
       pch = 16, col = "deepskyblue")
points(x = flowers$weight[flowers$nitrogen == "medium"],
       y = flowers$shootarea[flowers$nitrogen == "medium"],
```

```
        pch = 16, col = "yellowgreen")
points(x = flowers$weight[flowers$nitrogen == "high"],
       y = flowers$shootarea[flowers$nitrogen == "high"],
       pch = 16, col = "deeppink3")
text(x = 28, y = 190, label = "A", cex = 2)
leg_cols <- c("deepskyblue", "yellowgreen", "deeppink3")
leg_sym <- c(16, 16, 16)
leg_lab <- c("low", "medium", "high")

legend(x = 1, y = 200, col = leg_cols, pch = leg_sym,
       legend = leg_lab, bty = "n",
       title = "Nitrogen level")
```

The table below highlights some of the low level potting functions you might find useful.

Function	Description
`lines()`	add connected lines to a plot
`curve()`	draws a curve corresponding to a function
`arrows()`	draws arrows between 2 points
`text()`	adds text to a plot
`mtext()`	adds text to one of the 4 plot margins
`axis()`	adds an axis to the current plot
`rect()`	draws a rectangle
`legend()`	adds a legend to the plot
`points()`	adds points to the plot
`abline()`	adds a straight line to a plot
`grid()`	adds a rectangular grid to the current plot
`polygon()`	draws a polygon

4.4   Multiple graphs

There are a number of different methods for plotting multiple graphs within the same graphics device, some of which you've already met such as `pairs()`, `coplot()`, `xyplot()` etc. However these functions rely on plotting multiple graphs in different panels within the same plot. If you want to plot separate plots within the same graphics device you'll need a different approach. One of the most common methods is to use the main graphical function `par()` to split the plotting device up into a number of defined sections using the `mfrow =` argument. With this method, you first need to specify the number of rows and columns of plots you would like and then run the code for each plot. For

example, to plot two graphs side by side we would use `par(mfrow = c(1, 2))` to split the device into 1 row and two columns.

```r
par(mfrow = c(1, 2))
plot(flowers$weight, flowers$shootarea, xlab = "weight",
     ylab = "shoot area")
boxplot(shootarea ~ nitrogen, data = flowers, cex.axis = 0.6)
```
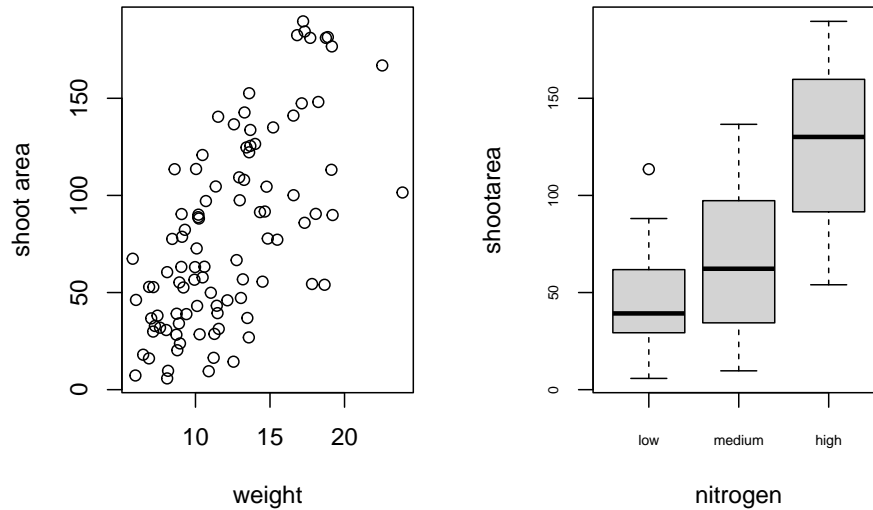

Or if we wanted to plot four plots we can split our plotting device into 2 rows and 2 columns.

```r
par(mfrow = c(2, 2))
plot(flowers$weight, flowers$shootarea, xlab = "weight",
     ylab = "shoot area")
boxplot(shootarea ~ nitrogen, cex.axis = 0.8, data = flowers)
hist(flowers$weight, main ="")
dotchart(flowers$weight)
```
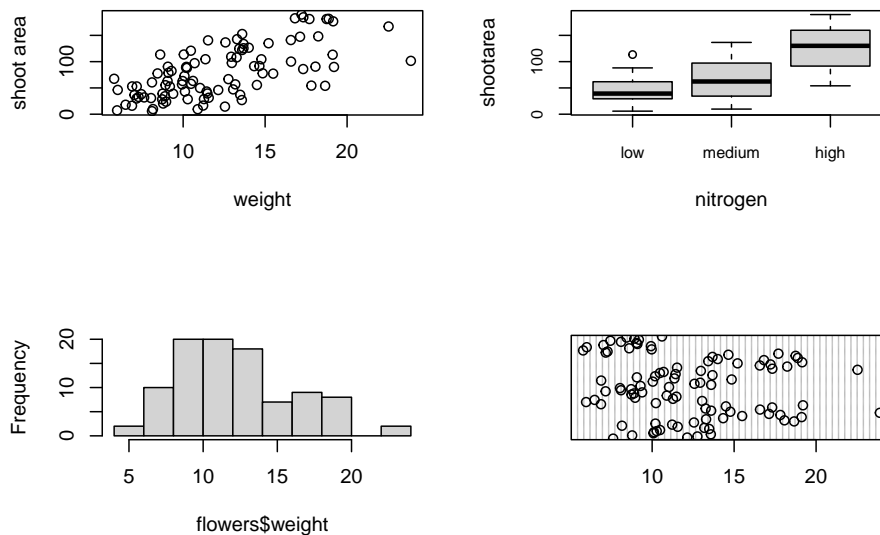
Once you've finished making your plots don't forget to reset your plotting device back to normal with `par(mfrow = c(1, 1))`.

A more flexible approach is to use the `layout()` function. The `layout()` function allows you to split your plotting device up into different sized regions and can be used to build complex figures. Before using the `layout()` function we first need to specify how we're going to split our plotting device by creating a matrix using the `matrix()` function (see Chapter 2 to remind yourself). Let's create a 2 x 2 matrix.

```
layout_mat <- matrix(c(2, 0, 1, 3), nrow = 2, ncol = 2,
                     byrow = TRUE)
layout_mat
```

```
##      [,1] [,2]
## [1,]    2    0
## [2,]    1    3
```

The matrix above represents splitting the plotting device into 2 rows and 2 columns. The first plot will occupy the lower left panel, the second plot the upper left panels and the third plot the lower right panel. The upper right panel will not contain a plot as we have placed a zero here.

We can now use the `layout()` function to define our layout. As we have two rows and two columns we need to specify the height of each row and the width of each column using the `heights =` and `widths =` arguments. The `respect = TRUE` argument ensures that the units used to define the widths are the same as those to define the heights. We can get a graphical representation of our layout by using the `layout.show()` function.

```
my_lay <- layout(mat = layout_mat,
                 heights = c(1, 3),
                 widths = c(3, 1), respect =TRUE)
layout.show(my_lay)
```
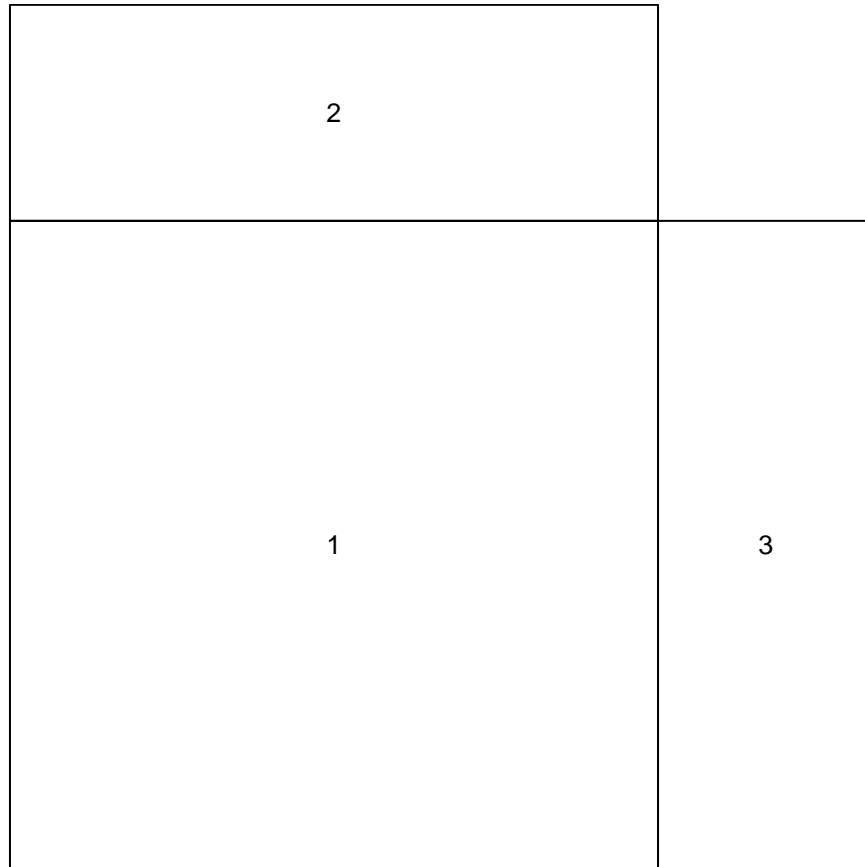
All we need to do now is create our three plots. However, before we do this we also need to change the figure margins for each of the figures using the `par(mar = )` command so all of the plots can fit together in the same plotting device. This will probably take a little bit of experimenting to get the plot looking exactly how you want. For our first figure (bottom left) we will reduce the size of the bottom and left margins a little and remove the margins completely from the top and right sides with `par(mar = c(4, 4, 0, 0))`. For our top plot we will remove the margins from the bottom, top and right sides and set the left side to have the same margin as our first figure (`par(mar = c(0, 4, 0, 0))`). For the third plot on the right we will set the bottom side to have the same margin as the first plot so they line up and remove the margins from the other sides with `par(mar = c(4, 0, 0, 0))`.

```r
par(mar = c(4, 4, 0, 0))
plot(flowers$weight, flowers$shootarea,
     xlab = "weight (g)", ylab = "shoot area (cm2)")
par(mar = c(0, 4, 0, 0))
boxplot(flowers$weight, horizontal = TRUE, frame = FALSE,
        axes =FALSE)
par(mar = c(4, 0, 0, 0))
boxplot(flowers$shootarea, frame = FALSE, axes = FALSE)
```
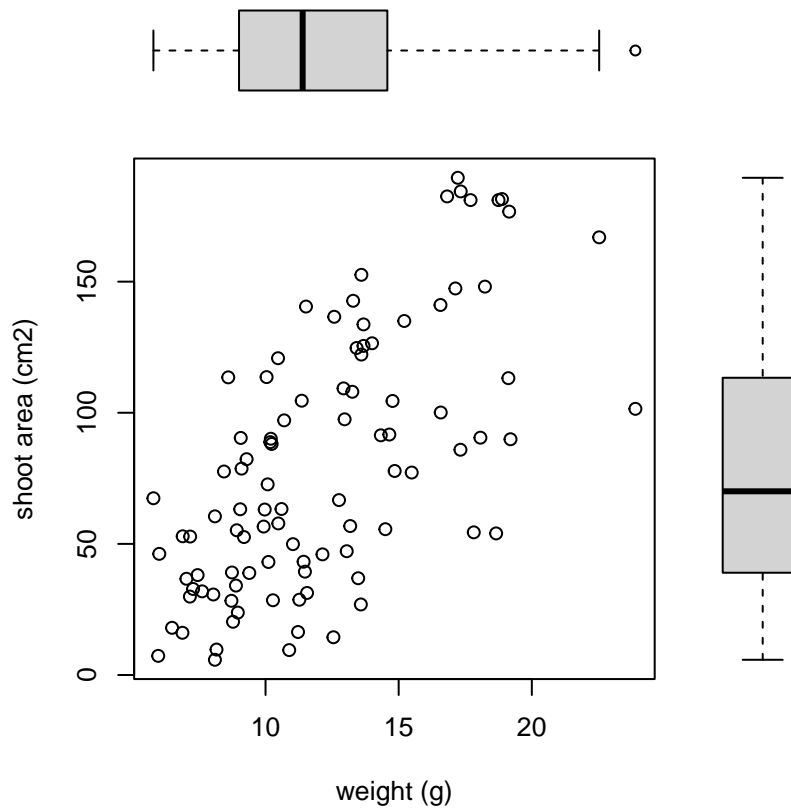
Notice we've specified that the boxplot at the top should be plotted horizontally with the `horizontal = TRUE` argument. The `frame = FALSE` argument prevents a border being plotted around each boxplot and the `axes = FALSE` argument suppresses the axes and axes labels from being plotted.

4.5   Exporting plots

Creating plots in R is all well and good but what if you want to use these plots in your thesis, report or publication? One option is to click on the 'Export' button in the 'Plots' tab in RStudio as we described previously. You can also export your plots from R to an external file by writing some code in your R script. The advantage of this approach is that you have a little more control over the output format and it also allows you to generate (or update) plots automatically whenever you run your script. You can export your plots in many different formats but the most common are, pdf, png, jpeg and tiff.

By default, R (and therefore RStudio) will direct any plot you create to the plot window. To save your plot to an external file you first need to redirect your plot to a different graphics device. You do this by using one of the many graphics device functions to start a new graphic device. For example, to save a plot in pdf format we will use the `pdf()` function. The first argument in the `pdf()` function is the filepath and filename of the file we want to save (don't forget to include the .pdf extension). Once we've used the `pdf()` function we can then write all of the code we used to create our plot including any graphical parameters such as setting the margins and splitting up the

plotting device. Once the code has run we need to close the pdf plotting device using the `dev.off()` function.

```
pdf(file = 'output/my_plot.pdf')
par(mar = c(4.1, 4.4, 4.1, 1.9), xaxs="i", yaxs="i")
plot(flowers$weight, flowers$shootarea,
     xlab = "weight (g)",
     ylab = expression(paste("shoot area (cm"^"2",")")),
     xlim = c(0, 30), ylim = c(0, 200), bty = "l",
     las = 1, cex.axis = 0.8, tcl = -0.2,
     pch = 16, col = "dodgerblue1", cex = 0.9)
text(x = 28, y = 190, label = "A", cex = 2)
dev.off()
```

If we want to save this plot in png format we simply use the `png()` function in more or less the same way we used the `pdf()` function.

```
png('output/my_plot.png')
par(mar = c(4.1, 4.4, 4.1, 1.9), xaxs="i", yaxs="i")
plot(flowers$weight, flowers$shootarea,
     xlab = "weight (g)",
     ylab = expression(paste("shoot area (cm"^"2",")")),
     xlim = c(0, 30), ylim = c(0, 200), bty = "l",
     las = 1, cex.axis = 0.8, tcl = -0.2,
     pch = 16, col = "dodgerblue1", cex = 0.9)
text(x = 28, y = 190, label = "A", cex = 2)
dev.off()
```

Other useful functions are; `jpeg()`, `tiff()` and `bmp()`. Additional arguments to these functions allow you to change the size, resolution and background colour of your saved images. See `?png` for more details.

4.6   Exercise 4

Congratulations, you've reached the end of Chapter 4! Perhaps now's a good time to practice some of what you've learned. You can find an exercise we've prepared for you (and our solutions) on the course website.

Chapter 5

Graphics with ggplot

For many people, using R to create informative and pretty figures is one of the more rewarding aspects of using R. These can either take the form of a rough and ready plot to get a feel for what's going on in your data, or a fancier, more complex figure to use in a publication or a report. This process is often as close as many scientists get to having a professional creative side (at least that's true for us), and it's a source of pride for some folk.

As mentioned in the Introduction, one of the many reasons for the rise in the popularity of R is its ability to produce publication quality figures. Not only can R users make figures well suited for publication, but the means in which the figures are produced also offers a wide-range of customisation. This in turn allows users to create their own particular styles and brands of figures which are well beyond the cookie-cutter styles in more traditional point and click software. Because of this inherent flexibility when producing figures, data visualisation in R and supporting packages has grown substantially over the years.

In this Chapter, we will focus on creating figures using a specialised package called `ggplot2`.

Before we get going with making some plots of the `gg` variety, how about a quick history of one of the most commonly used packages in R? `ggplot2` was based on a book called *Grammar of Graphics* by Leland Wilkinson (hence the gg in `ggplot2`), yours for only £100 or so. But before you spend all that money, see here for an interesting summary of Wilkinson's book.

The *Grammar of Graphics* approach moves away from the idea that to create, for example, a scatterplot, users should click the `scatterplot` button or use the `scatterplot()` function. Instead, by breaking figures down into their various components (e.g. the underlying statistics, the geometric arrangement, the theme, see Fig. 5.1), users will be able to manipulate each of these components (i.e. layers) and produce a tailor-made figure fit for their specific needs. Contrast this approach with the one used by, for example, Microsoft Excel. The user specifies the data and then clicks the scatterplot button. This inherently locks the user into many choices made by the software developer and not the user. Think of how easily you can spot an Excel scatterplot because other than a couple of pre-set options, there's really not much you can do to change the way the plot displays the data - you are at the mercy of the [insert corporation here] Gods.
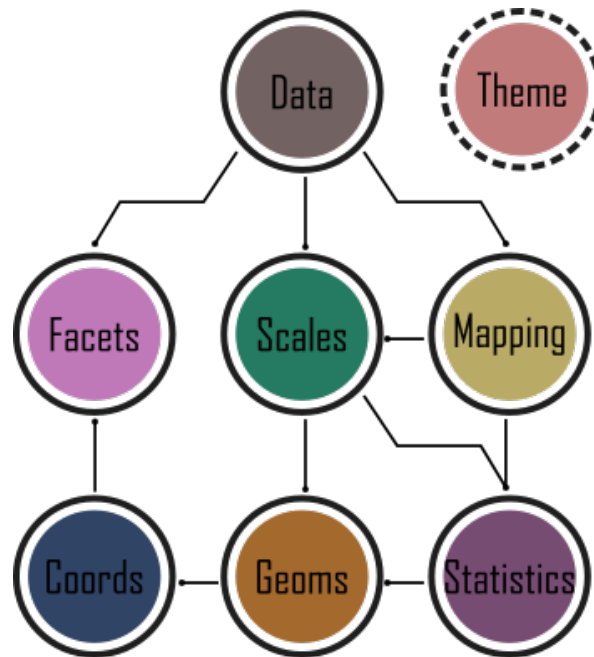
Figure 5.1: framework behind ggplot2

While Wilkinson would eventually go on to become vice-president of SPSS, his (and his oft forgotten co-author's) ideas would, never-the-less, make their way into R via `ggplot2` as well as other implementations (e.g. tableau).

In 2007 `ggplot2` was released by Hadley Wickham. By 2017 the package had reportedly been downloaded 10 million times and over the last few years `ggplot2` has become the foundation for numerous other packages which expand its functionality even more. `ggplot2` is now part of the tidyverse collection of R packages.

It's important to note that `ggplot2` is not **required** to make "fancy" and informative figures in R. If you prefer using base R graphics then feel free to continue as almost all `ggplot2` type figures can be created using base R (we often use either approach depending on what we're doing). The difference between`ggplot2` and base R is how you *get* to the end product rather than any substantial differences in the end product itself. This is, never-the-less, a common belief probably due to the fact that making a moderately attractive figure is (in our opinion at least), easier to do with `ggplot2` as many aesthetic decisions are made for the user, without you necessarily even knowing that a decision was ever made!

With that in mind, let's get started making some figures.

5.1   Beginning at the end

The approach we'll use in this Chapter will be to start off by showing you a figure which we suggest is at a standard that you could use in a poster or presentation. Using

that as the aim, we will then work towards it step-by-step. You should not view this final figure as any sort of holy grail. For instance, you would be very unlikely to use this in a publication (you'd be much more likely to use some results from your hard earned-analysis). Regardless, this "final figure" is, and will only ever be, a reflection of what our personal preferences are. As with anything subjective, you may well disagree, and to some extent we hope you do. Much better that we all have slightly (or grossly) different views on what a good figure is - otherwise we may as well go back to using cookie-cutter figures.

So what's the figure we're going to make together?

Note: Regression assumptions are unvalidated
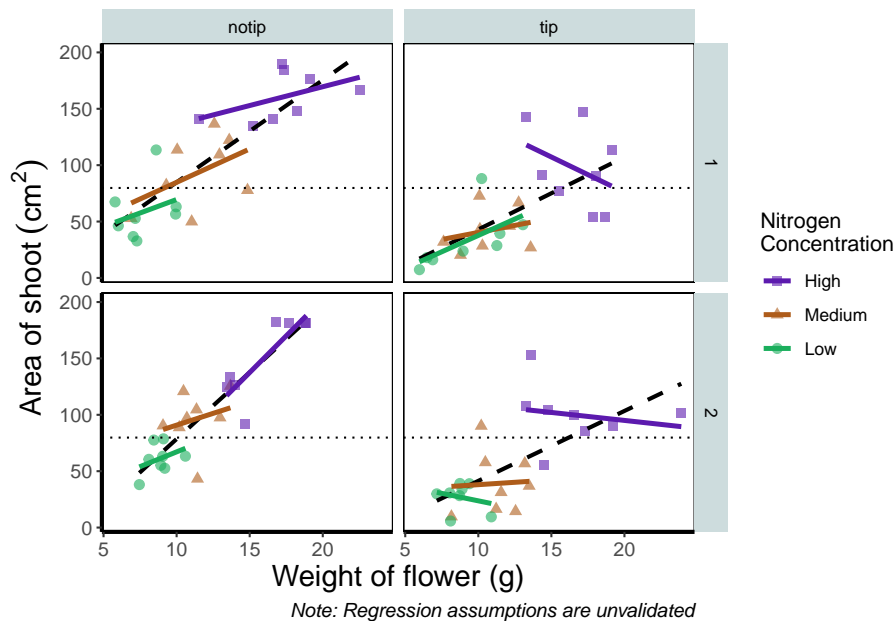
Before we go further, let's take a second and talk about what this figure is showing. On the y axes of the four plots we have the surface area of flower shoots, and on the x axes we have the weight of the flowers. Each column of plots shows whether the tip of the flower was removed (notip) or retained (tip). Each row of plots identifies which experimental block in the greenhouse the plants were grown in, either 'block 1' or 'block 2'.

The different coloured and shaped points within each plot represents plants grown at three nitrogen concentrations. These colours are used to distinguish which points correspond to a certain nitrogen concentration. For example, data points coloured green were plants grown in low nitrogen concentrations, brown in medium nitrogen and purple in high concentrations.

We have also added four trend lines to each plot (using a linear model, see Chapter 6). The three solid coloured lines show the relationship between shoot area and weight of the flower according to which nitrogen treatment and block the plants were grown. The dashed black line in each plot represents the relationship between shoot area and flower weight whilst ignoring any nitrogen affect.

Finally, the thin grey dotted line on each plot represents the overall mean shoot area regardless of the weight of the flowers, the nitrogen concentration or the block.

For the purposes of this Chapter, we won't worry about the biology here. Do not take this as standard practice, you should absolutely care deeply about the science in your own data. It's the science that should be the driving force behind the questions you ask, which in turn determines what figures you should make.

5.2   The start of the end

The first step in producing a plot with `ggplot()` is the easiest! We just need to install and then make the package avaialble. Use the skills you learnt in Chapter 1 to install and load the package. Note that although most people refer to the package as `ggplot`, it's proper name is `ggplot2`.

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

With that taken care of, let's make our first `ggplot`!

5.2.1   The purest of ggplots

When we run our 'in person' R courses that accompany this book, we often ask our students to name all of the functions they have either learnt during the course, have heard of previously, or have used before (we call it R bingo!). At this point in the course, the students have not yet learnt about `ggplot2`, but never-the-less one year a student suggested the function `ggplot()`. When asked what the `ggplot()` function does, they joked that it obviously makes a `ggplot`. This makes intuitive sense, so let's make a `ggplot` now:

```r
ggplot()
```
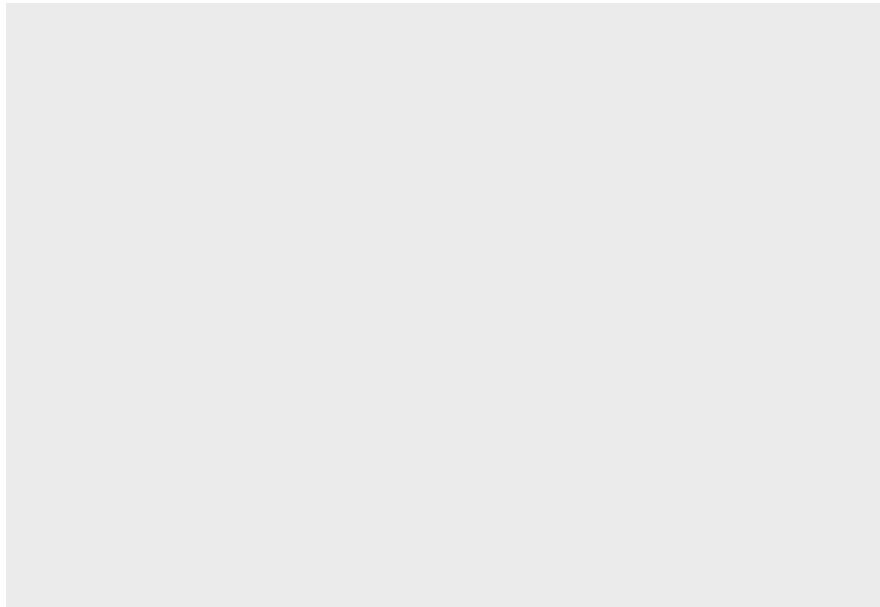
And here we have it. A fully formed, perfect `ggplot`. We may have a small issue though. Some puritan data visualisers/plotists/figurines make the claim that figures should include some form of information beyond a light grey background. As loathe as we are to agree with purists, we'll do so here. We really should include some information, but to do so, we need data.

We'll keep using the `flower` dataset that you've used in Chapter 3. Let's have a quick reminder of what the structure of the data looks like.

```
flower <- read.table("data/flower.csv", stringsAsFactors = TRUE,
                                     header = TRUE, sep = ",")
str(flower)
## 'data.frame':    96 obs. of  8 variables:
##  $ treat    : Factor w/ 2 levels "notip","tip": 2 2 2 2 2 2 2 2 2 2 ...
##  $ nitrogen : Factor w/ 3 levels "high","low","medium": 3 3 3 3 3 3 3 3 3 3 ...
##  $ block    : int  1 1 1 1 1 1 1 1 2 2 ...
##  $ height   : num  7.5 10.7 11.2 10.4 10.4 9.8 6.9 9.4 10.4 12.3 ...
##  $ weight   : num  7.62 12.14 12.76 8.78 13.58 ...
##  $ leafarea : num  11.7 14.1 7.1 11.9 14.5 12.2 13.2 14 10.5 16.1 ...
##  $ shootarea: num  31.9 46 66.7 20.3 26.9 72.7 43.1 28.5 57.8 36.9 ...
##  $ flowers  : int  1 10 10 1 4 9 7 6 5 8 ...
```

We know from the "final figure" that we want the variable `shootarea` on the y axis (response/dependent variable) and `weight` on the x axis (explanatory/independent variable). To do this in `ggplot2` we need to make use of the `aes()` function and also add a `data =` argument. `aes` is short for aesthetics, and it's the function we use to specify what we want displayed in the figure.

If we did not include the `aes()` function, then the `x =` and `y =` arguments would produce an error saying that the object was not found. A good rule to keep in mind when using

ggplot2 is that the variables which we want displayed on the figure must be included in aes() function via the mapping = argument (corresponding to the mapping layer from Figure. 4.1).

All features in the figure which alter the displayed information, not based on a variable in our dataset (e.g. increasing the size of points to an arbitrary value), is included outside of the aes() function. Don't worry if that doesn't make sense for now, we'll come back to this later.

Let's update our code to include the 'data' and 'mapping' layers (indicated by the grey Data and mustard Mapping *layer bubbles* which will precede relevant code chunks):


```
# Including aesthetics for x and y axes as well as specifying the dataset
ggplot(mapping = aes(x = weight, y = shootarea), data = flower)
```
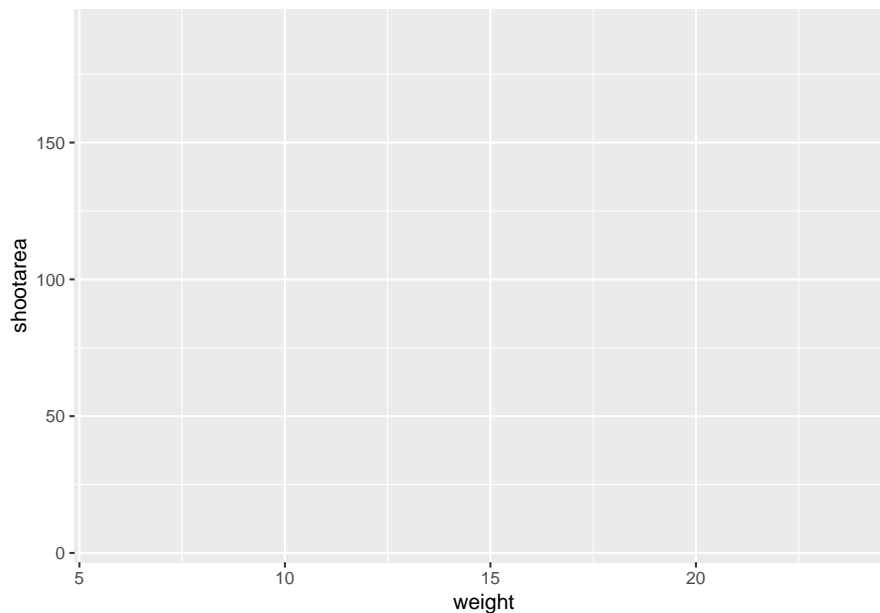

That's already much better. At least it's no longer a blank grey canvas. We've now told ggplot2 what we want as our x and y axes as well as where to find that data. But what's missing here is where we tell ggplot2 *how* to display that data. This is now the time to introduce you to 'geoms' or geometry layers.

Geometries are the way that ggplot2 displays information. For instance geom_point() tells ggplot2 that you want the information to be displayed as points (making scatterplots possible for example). Given that the "final figure" uses points, this is clearly the appropriate geom to use here.

Before we can do that, we need to talk about the coding structure used by `ggplot2`. The analogy that we and many others use is to say that making a figure in `ggplot2` is much like painting. What we've did in the above code was to make our "canvas". Now we are going to add sequential layers to that painting, increasing the complexity and detail over time. Each time we want to include a new layer we need to end a preceding layer with a + at the end to tell `ggplot2` that there are additional layers coming.

Let's add (+) a new geometry layer now:


```
ggplot(aes(x = weight, y = shootarea), data = flower) +
        geom_point()     # Adding a geom to display data as point data
```
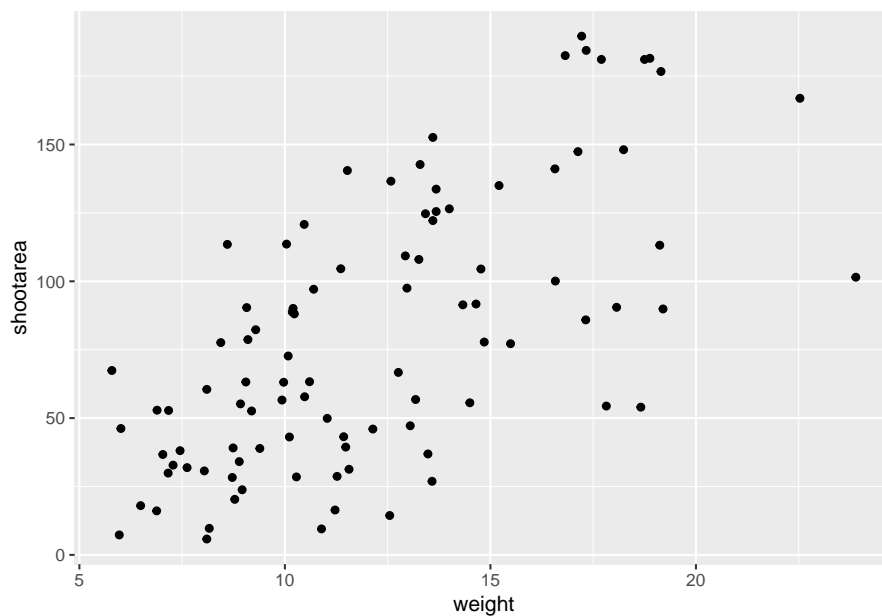

When you first start using `ggplot2` there are three crucial layers that require your input. You can safely ignore the other layers initially as they all receive sensible (if sometimes ugly) defaults. The three crucial layers are:

Figure 5.2: Data - the information we want to plot

Figure 5.3: Mapping - which variables we want displayed and where

Figure 5.4: Geometry - how we want that data displayed

Given that 'data' only requires us to specify the dataset we want to use, it is trivially easy to complete. 'Mapping' only requires you to specify what variables in the data to use, often just the x- and y-axes (specified using `aes()`). Lastly, 'geometry' is where we choose how we want the data to be visualised.

With just these three fundamentals, you will be able to produce a large variety of plots (see later in this Chapter for a bestiary of plots).

If what we wanted was a quick and dirty figure to get a grasp of the trend in the data we can stop here. From the scatterplot that we've produced, we can see that `shootarea` looks like it's increasing with `weight` in a linear fashion. So long as this answers the question we were asking from these data, we have a figure that is fit for purpose. However, for showing to other people we might want something a bit more developed. If we glance back to our "final figure" we can see that we have lines representing the different nitrogen concentrations. We can include lines using a geom. If you have a quick look through the available geoms here, you might think that `geom_line()` would be appropriate. Let's try it.


```
ggplot(aes(x = weight, y = shootarea), data = flower) +
  geom_point() +
    geom_line()     # Adding geom_line
```
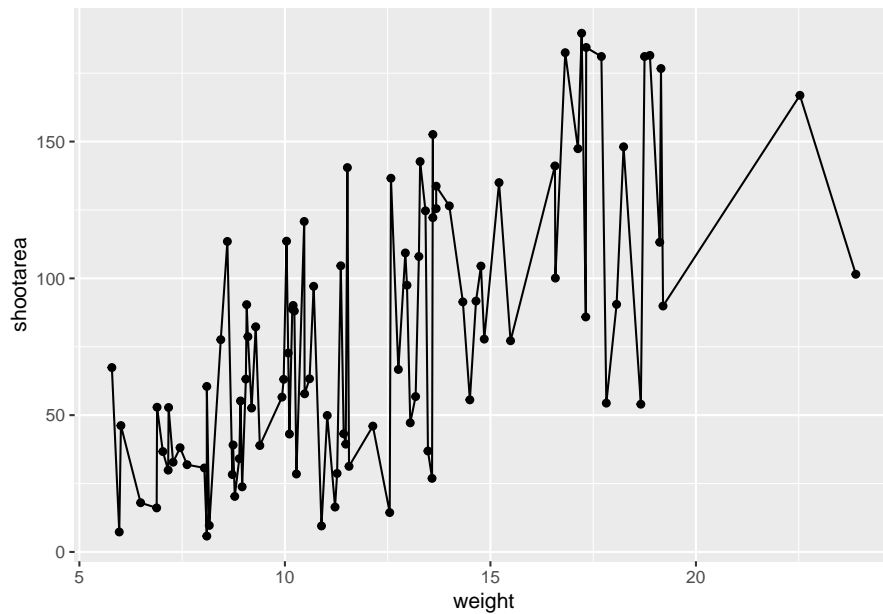
Not quite what we were going for. The problem that we have is that `geom_line()` is actually just playing join-the-dots in the order they appear in the data (an alternative to `geom_path()`). The geom we actually want to use is called `geom_smooth()`. We can fix that very easily just by changing "line" to "smooth".


```
ggplot(aes(x = weight, y = shootarea), data = flower) +
  geom_point() +
    geom_smooth()     # Changing to geom_smooth
```
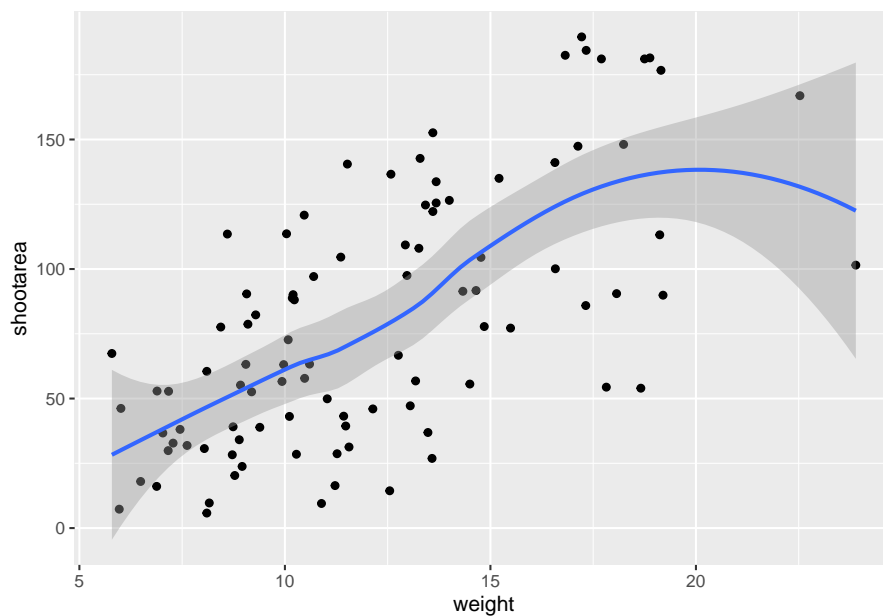
Better, but still not what we wanted. The challenge here is that drawing a line is actu-
ally somewhat complicated. The way our line above was drawn was by using a method
called "LOESS" (locally estimated scatterplot smoothing) which gives something very
close to a moving average; useful in some cases, less so in others. `ggplot2` will use LOESS
as default when you have < 1000 observations, so we'll need to manually specify the
method. Instead of a wiggly line, we want a nice simple 'line of best fit' to be drawn
using a method called "lm" (short for linear model - see Chapter 6 for more details).
Try looking at the help file, using `?geom_smooth`, to see what other options are available
for the `method =` argument.

While we're at it, let's get rid of the confidence interval ribbon around the line. We
prefer to do this as we think it's clearer to the audience that this isn't a properly
analysed line and to treat it as a visual aid only. We can do this at the same time as
changing the method by setting the `se =` argument (short for standard error) to `FALSE`.

Let's update the code to use a linear model without confidence intervals.

```
ggplot(aes(x = weight, y = shootarea), data = flower) +
  geom_point() +
    geom_smooth(method = "lm", se = FALSE)    # method and se
```
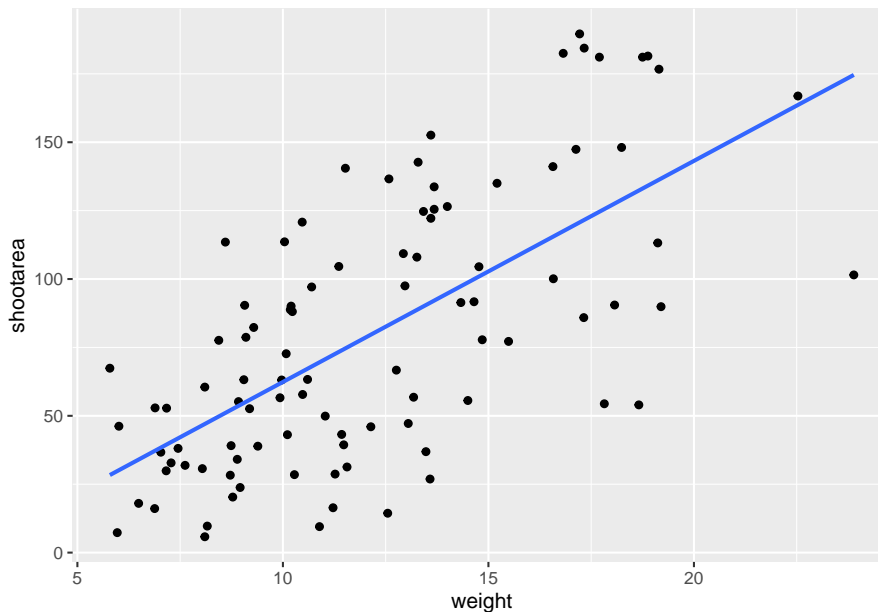

We get the straight line that we wanted, though it's still not matching the "final fig-
ure". We need to alter `geom_smooth()` so that it draws lines for each level of nitrogen

concentration. Getting `ggplot2` to do that is pretty straightforward. We can use the `colour =` argument within `aes()` (remember whatever we include in `aes()` will be something displayed in the figure) to tell `ggplot2` to draw a different coloured lines depending on the `nitrogen` variable. Keep in mind that we have no variable in our dataset called "nitrogen_colour", so `ggplot2` is taking care of that for us here and assigning a colour to each unique nitrogen level.

An aside: `ggplot2` was written with both UK English and American English in mind, so both `colour` and `color` spellings work in `ggplot2`.


```
ggplot(aes(x = weight, y = shootarea), data = flower) +
  geom_point() +
  # Including colour argument in aes()
  geom_smooth(aes(colour = nitrogen), method = "lm", se = FALSE)
```
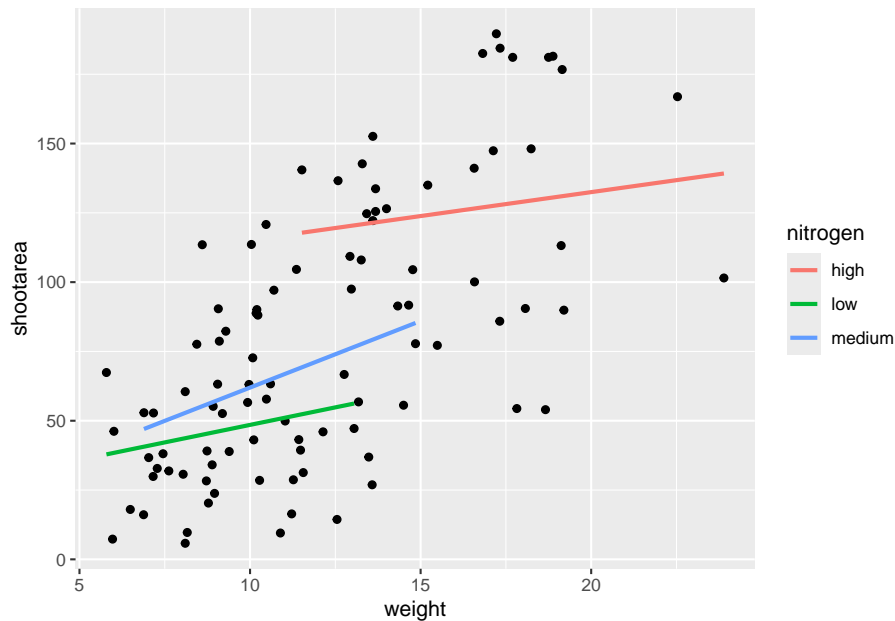

We're getting closer, especially since `ggplot2` has automatically created a legend for us. At this point it's a good time to talk about where to include information - whether to include it within a geom or in the main call to `ggplot()`. When we include information such as `data =` and `aes()` in `ggplot()` we are setting those as the default, universal values which all subsequent geoms use. Whereas if we were to include that information within a geom, only that geom would use that specific information. In this case, we can easily move the information around and get exactly the same figure.

```
ggplot() +
    # Moved aes() and data into geoms
  geom_point(aes(x = weight, y = shootarea), data = flower) +
  geom_smooth(aes(x = weight, y = shootarea, colour = nitrogen),
                    data = flower, method = "lm", se = FALSE)
```
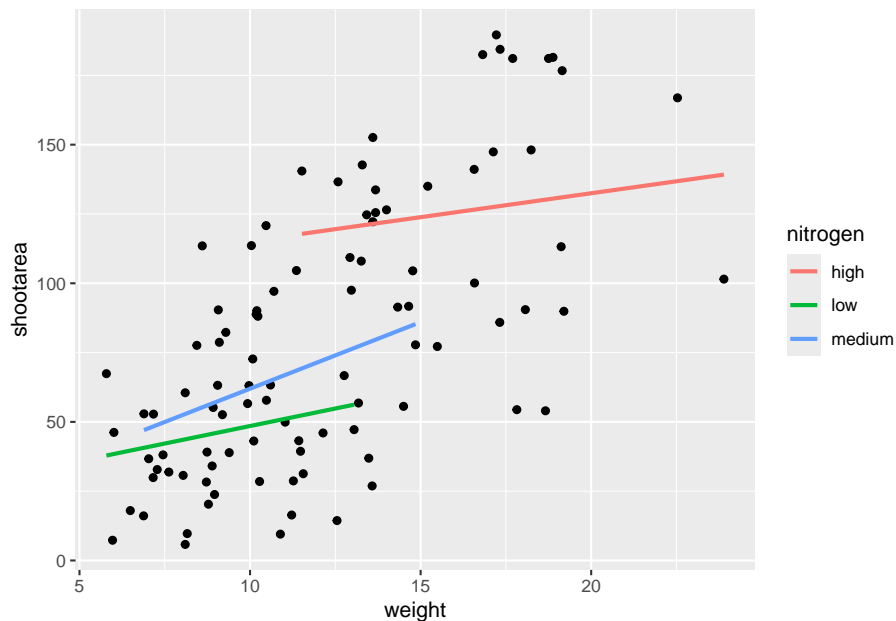

Doing so we get exactly the same figure. This ability to move information between the main ggplot() call or in specific geoms is surprisingly powerful (although sometime confusing!). It can allow different geoms to display different (albeit similar) information (see more on this later).

For this worked example, we'll move the same information back to the universal ggplot(), but we'll also move colour = nitrogen into ggplot() so that we can have the points coloured according to nitrogen concentration as well.


```
# Moved colour = nitrogen into the universal ggplot()
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
```
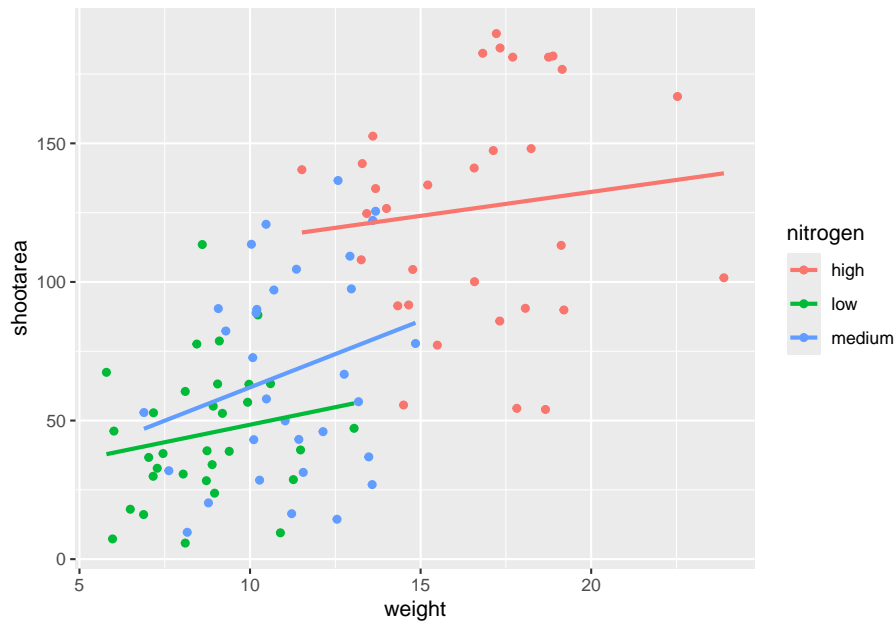
This figure is now what we would consider to be the typical `ggplot2` figure (once you know to look for it, you'll see it everywhere). We have specified some information, with only a few lines of code, yet we have something that looks quite attractive. While it's not yet the "final figure" it's perfectly suited for displaying the information we need from these data. You have now created your first "pure" `ggplot` using only the 'data', 'mapping' and 'geom' layers (as well as others indirectly).

Let's keep going as we're aiming for something a bit more "sophisticated".

5.2.2 Wrapping grids

Having made our "pure" `ggplot`, the next big obstacle we're going to tackle is the grid like layout of the "final figure" where our main figure has been split according to the `treat` and `block` variables, with new trends shown for each combination.

Each of these panels (technically "multiples") are a great way to help other people understand what's going on in the data. This is especially true with large datasets which can obscure subtle trends simply because so much data is overlaid on top of each other. When we split a single figure into multiples, the same axes are used for all multiples which serve to highlight shifts in the data (data in some multiples may have inherently higher or lower values for instance).

`ggplot2` includes options for specifying the layout of plots using the 'facets' layer. We'll start off by using `facet_wrap()` to show what this does. For `facet_wrap()` to work we need to specify a formula for how the facets will be defined (see `?facet_wrap` for more details and also how to define facets without using a formula). In our example we want to use the factor `treat` to determine the layout so our formula would look like ~ `treat`. You can read ~ `treat` as saying "according to treatment". Let's see how it works:

```r
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
    # Splitting the single figure into multiple depending on treatment
  facet_wrap(~ treat)
```
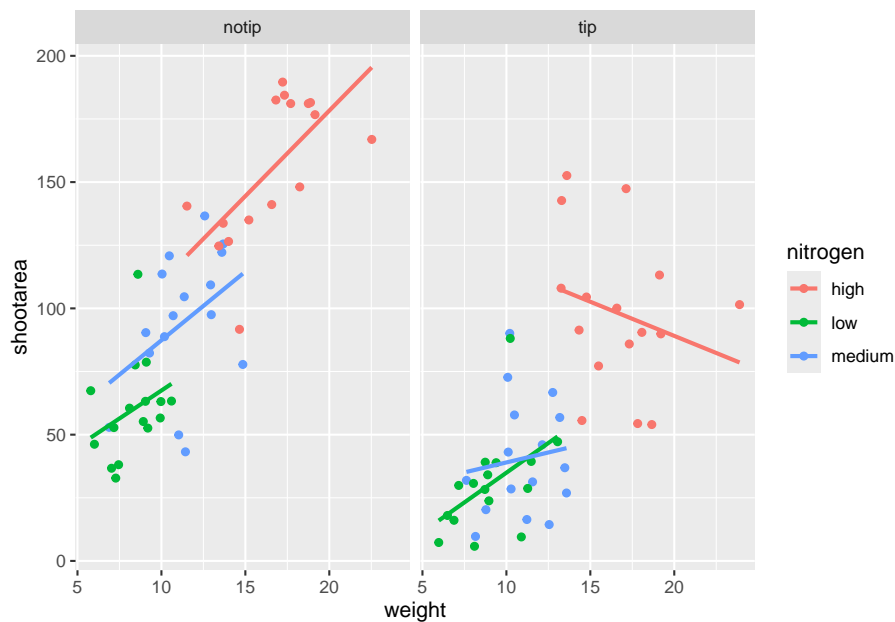

That's pretty good. Notice how we can see the impact that the `tip` treatment has on shoot area now (generally lowering shoot area), where in the previous figure this was much more difficult to see?

While this looks pretty good, we are still missing information showing any potential effect of the different blocks. Given that `facet_wrap()` can use a formula, maybe we could simply include `block` in the formula? Remember that the `block` variable refers to the region in the greenhouse where the plants were grown. Let's try it and see what happens.

```
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
    # Adding "block" to formula
  facet_wrap(~ treat + block)
```
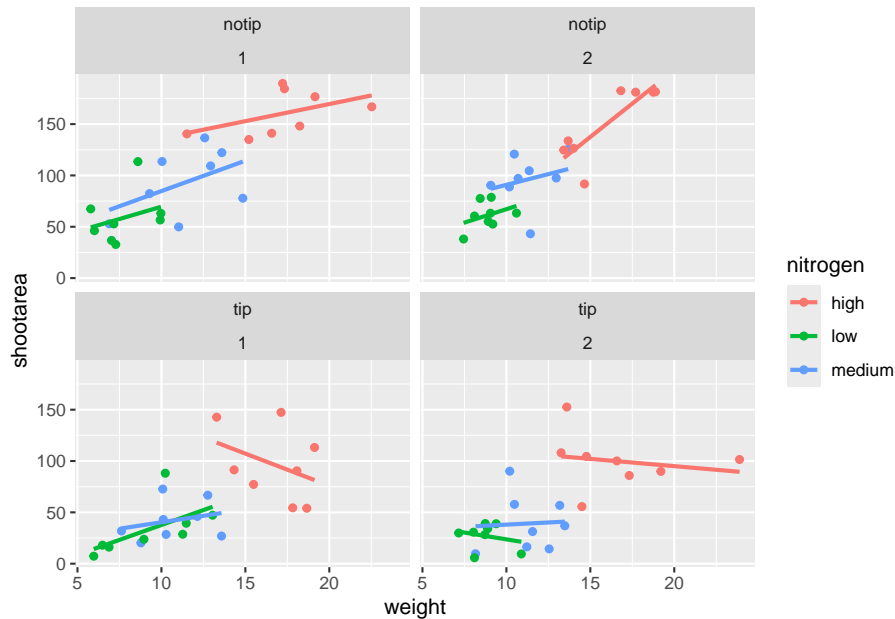

This facet layout is almost exactly what we want. Close but no cigar. In this case we actually want to be using `facet_grid()`, an alternative to `facet_wrap()`, which should put us back on track to make the "final figure".

Play around: Try changing the formula to see what happens. Something like ~ `treat + flowers` or even ~ `treat + block + flowers`. The important thing to remember here is that `facet_wrap()` will create a new figure for each value in a variable. So when you wrap using a continuous variable like flowers, it makes a plot for every unique number of flowers counted. Be aware of what it is that you are doing, but never be scared to experiment. Mistakes are easily fixed in R - it's not like a point and click programme where you'd have to go back through all those clicks to get the same figure produced. Made a mistake? Easy, change it back and rerun the code (see Chapter 9 for version control which takes this to the next level).

Let's try using `facet_grid` instead of `facet_wrap` to produce the following plot.

```r
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
    # Changing to facet_grid
  facet_grid(~ treat + block)
```
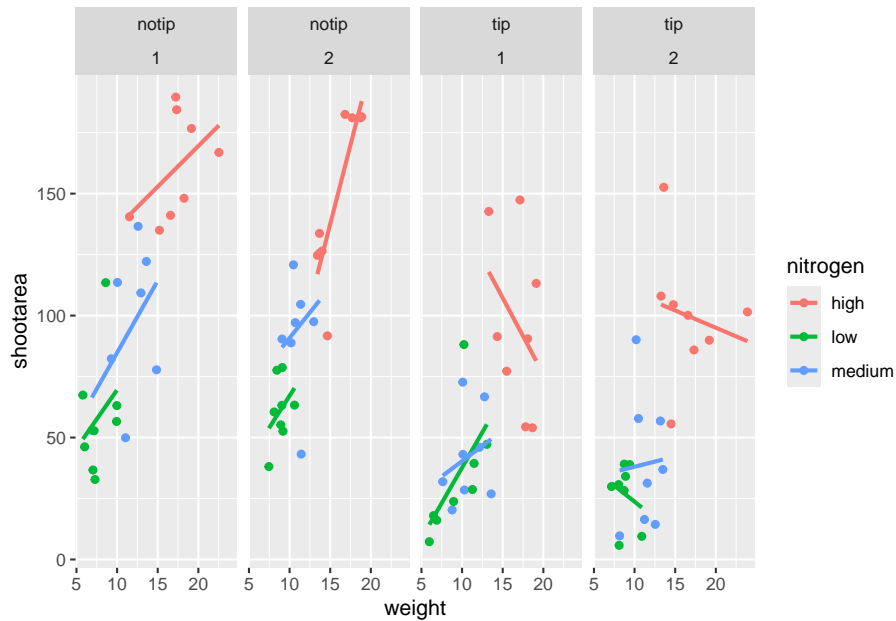

That's disappointing. It's pretty much the same as what we had before and is no closer to the "final figure". To fix this we need to do to rearrange our formula so that we say that it is block in relation to treatment (not in combination with).


```r
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
    # Rearranging formula, block in relation to treatment
  facet_grid(block ~ treat)
```
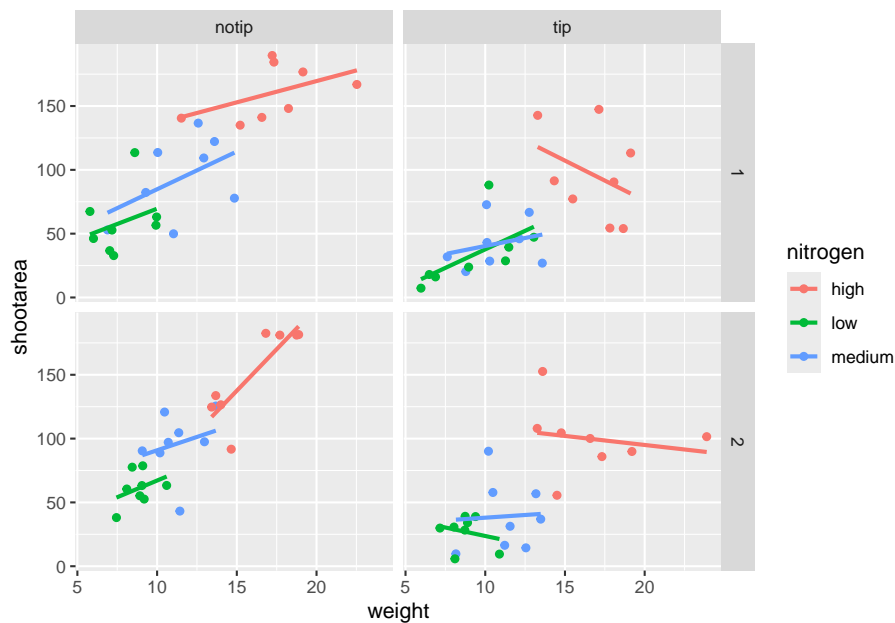
And we're there. Although the styling is not the same as the "final figure" this is showing the same core fundamental information.

5.2.3 Plotting multiple ggplots

While we've made multiples of the same figure, what if we wanted to take two completely different figures and plot them together in the same frame? As a demonstration, let's plot the last figure we made and the "final figure" shown at the start of this chapter one on top of the other to see how they compare. To do this we are going to use a package called `patchwork`. First you will need to install and make the `patchwork` package available.

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

An important note: For those who have used base R to produce their figures and are familiar with using `par(mfrow = c(2,2))` (which allows plotting of four figures in two rows and two columns) be aware that this does not work for `ggplot2` objects. Instead you will need to use either the `patchwork` package or alternative packages such as `gridArrange` or `cowplot` or covert the `ggplot2` objects to grobs.

To plot both of the plots together we need to go back to our previous code and do something clever. We need to assign each figure to a separate object and then use these objects when we use `patchwork`. For instance, we can assigned our "final figure" plot to an object called `final_figure` (we're not very imaginative!), you haven't see the code yet so you'll just have to take our word for it! You may see this method used a lot in other textbooks or online, especially when adding addition layers. Something like this:

```r
p <- ggplot(df, mapping = aes(x = x, y = y))
```

And later to add additional layers:

```r
p + geom_point()
```

We prefer not to use this approach here, as we like to always have the code visible to you while you're reading this book. Anyway, let's remind ourselves of the final figure.

Note: Regression assumptions are unvalidated
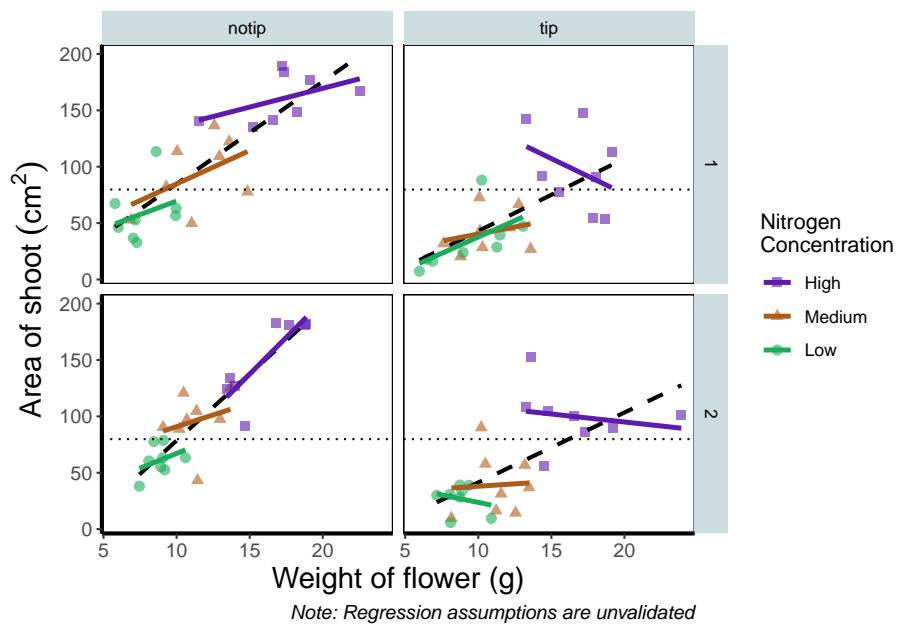
We'll now assign the code we wrote when creating our previous plot to an object called `rbook_figure`:

```r
# Naming our figure object
rbook_figure <- ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  facet_grid(block ~ treat)
```

Now when the code is run, the figure won't be shown immediately. To show the figure we need to type the name of the object. We'll do this at the same time as showing you how `patchwork` works.

An old headache when using `ggplot2` was that it could be difficult to create a nested figure (different plots, or "multiples", all part of the same dataset). `patchwork` resolves this problem very elegantly and simply. We have two immediate and simple options with patchwork; arrange figures on top of each other (specified with a `/`) or arrange figures side-by-side (specified with either a `+` or a `|`). Let's try to plot both figures, one on top of the other.

```
rbook_figure / final_figure
```


Note: Regression assumptions are unvalidated
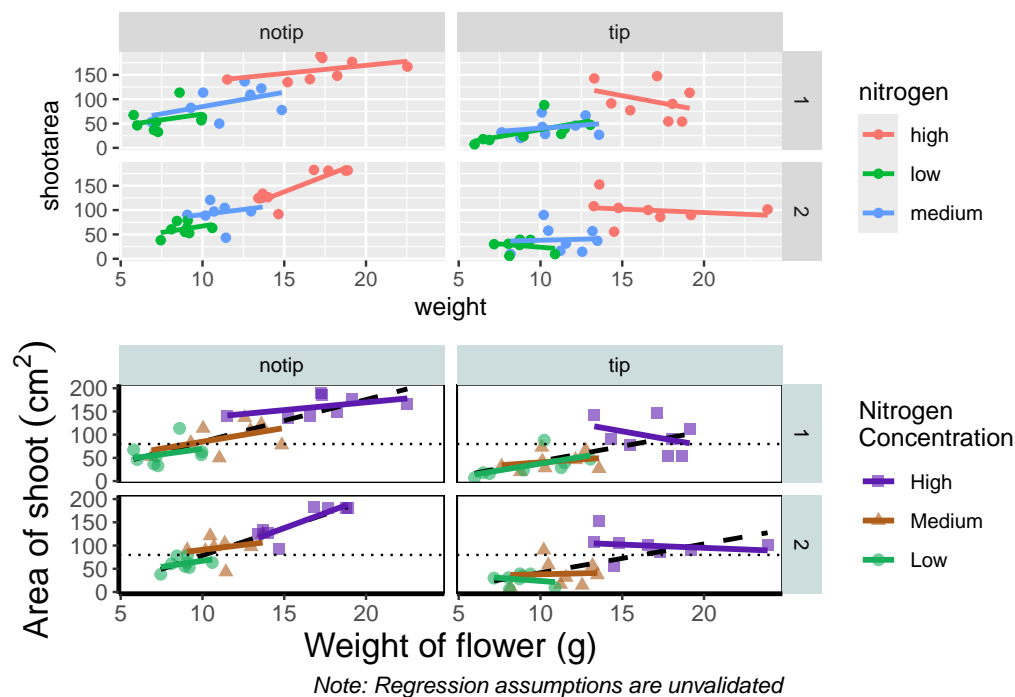
Play around: Try to create a side-by-side version of the above figure (hint: try the other operators [`+`, `/`, `|`]).

We can take this one step further and assign nested `patchwork` figures to an object and use this in turn to create labels for individual figures.

```
nested_compare <- rbook_figure / final_figure

nested_compare +
  plot_annotation(tag_levels = "A", tag_suffix = ")")
```

Note: Regression assumptions are unvalidated
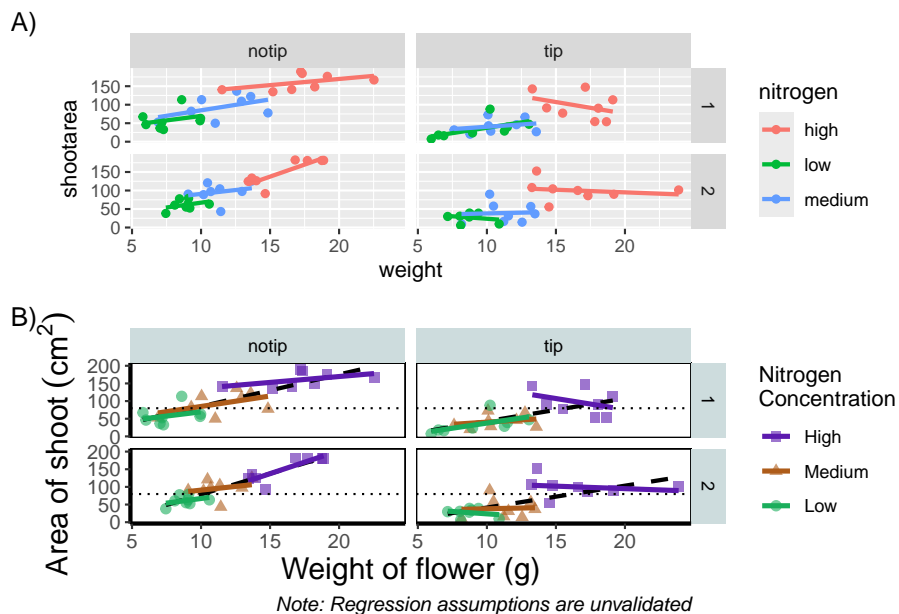
This is only the basics of what the `patchwork` package can do but there are many other uses. We won't go into them in any great detail here, but later in the tips and tricks section we'll show you some more cool features.

5.2.4   Make it your own

While we already have a great figure showing the main aspects of our data, it uses many default layer options. Whilst the default options are fine we may want to change them to get our plot looking exactly how we want it. Maybe we're going to use this figure in a presentation and we want to make sure someone in the very back of the room can easily read the figure. Maybe we want to use our own colour scheme. Maybe we want to change the grey background to a nice bright neon pink. In essence, maybe we want to decide things for ourselves. This next section will go through how to customise the appearance of our figure.

Let's start with the easier stuff, namely changing the size of the plotting symbols using the `size =` argument. Before we do, have a think about where we'd include this argument? Should it be in main call to `ggplot()` or in the `geom_point()` geom? Does size depend on a variable in our dataset and is therefore something we want displayed on the figure (meaning we should include it within `aes()`)? Or is it merely changing the appearance of information?

Let's include it in the `geom_point` geom

```
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    # Including size argument to change the size of the points
```

```
    geom_point(size = 2) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat)
```
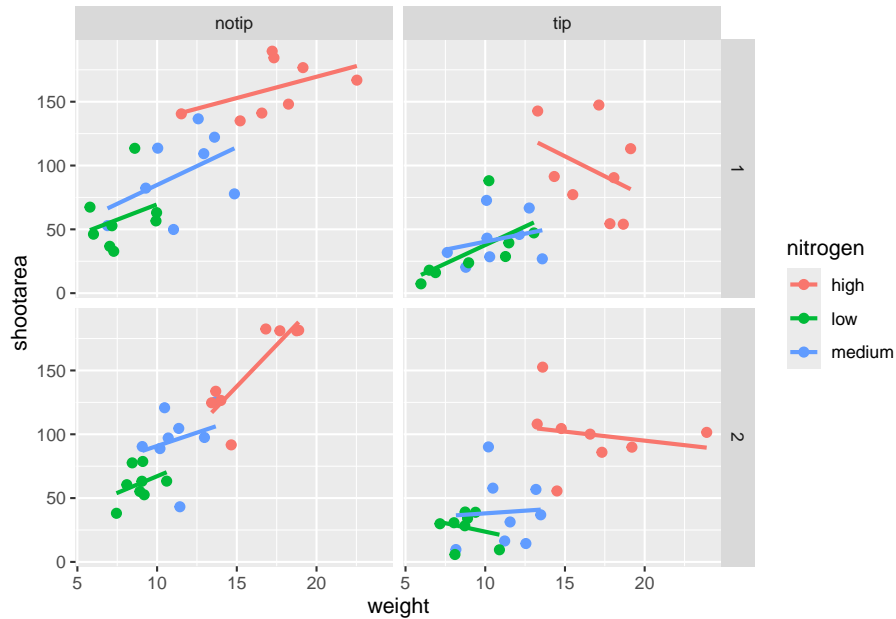

Pretty straight forward, we changed the size from the default of `size = 1` to a value that we decide for ourselves. What happens if you included size in `ggplot()` or within the `aes()` of `geom_point()`?

If we wanted to change the shape of the plotting symbols to reflect the different nitrogen concentrations (low, high, medium), how do you think we'd do that? We'd use the `shape = argument`, but this time we need to include an `aes()` within `geom_point()` because we want to include specific information to be displayed on the figure.


```
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    # Including shape argument to change the shape of the points
    geom_point(aes(shape = nitrogen), size = 2) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat)
```
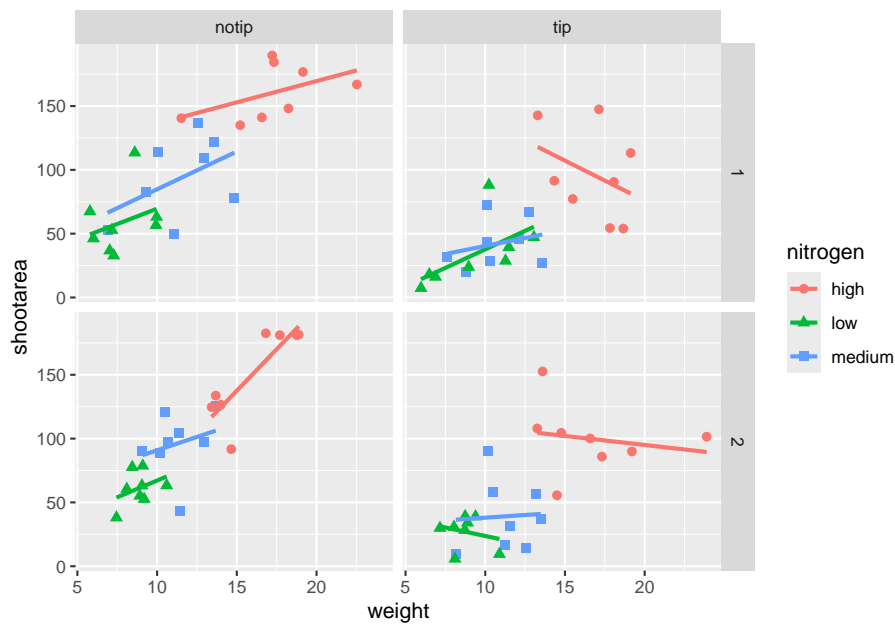
Try including `shape = nitrogen` without also including `aes()` and see what happens.

We're edging our way closer to our "final figure". Another thing we may want to be able to do is change the transparency of the points. While it's not actually that useful here, changing the transparency of points is really valuable when you have lots of data resulting in clusters of points obscuring information. Doing this is easily accomplished using the `alpha =` argument. Again, ask yourself where you think the `alpha =` argument should be included (hint: you should put it in the `geom_point` geom!).

```
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    # Including alpha argument to change the transparency of the points
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat)
```
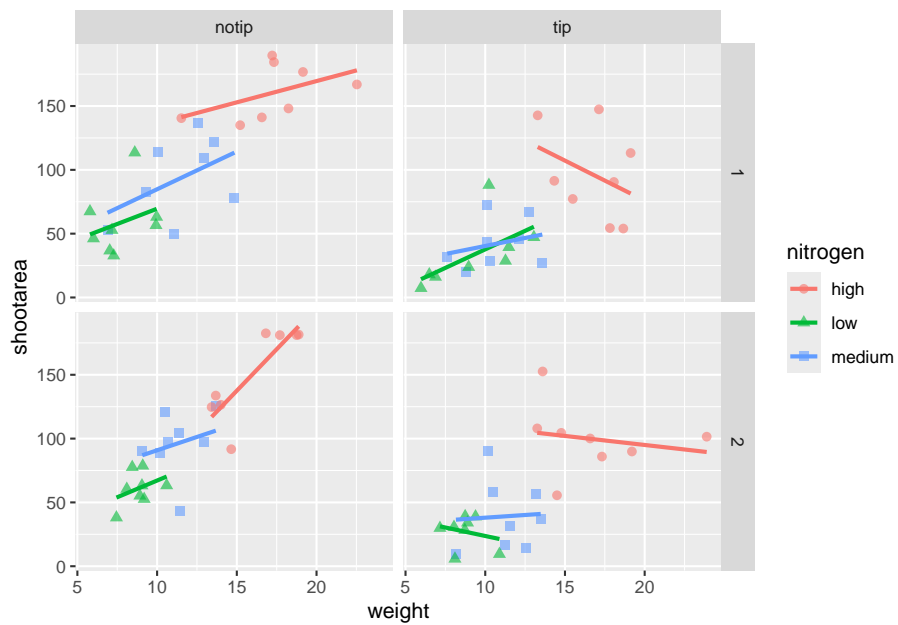
We can also include user defined labels for the x and y axis. There are a couple of ways to do this, but a more familiar way may be to use the same syntax as used in base R figures; using `xlab()` and `ylab()`. We'll specify that these belong to the `ggplot` by using the `+` symbol.

```
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat) +
    # Adding layers for x and y labels
    xlab("Weight of flower (g)") +
    ylab("Area of shoot (cm^2)")
```
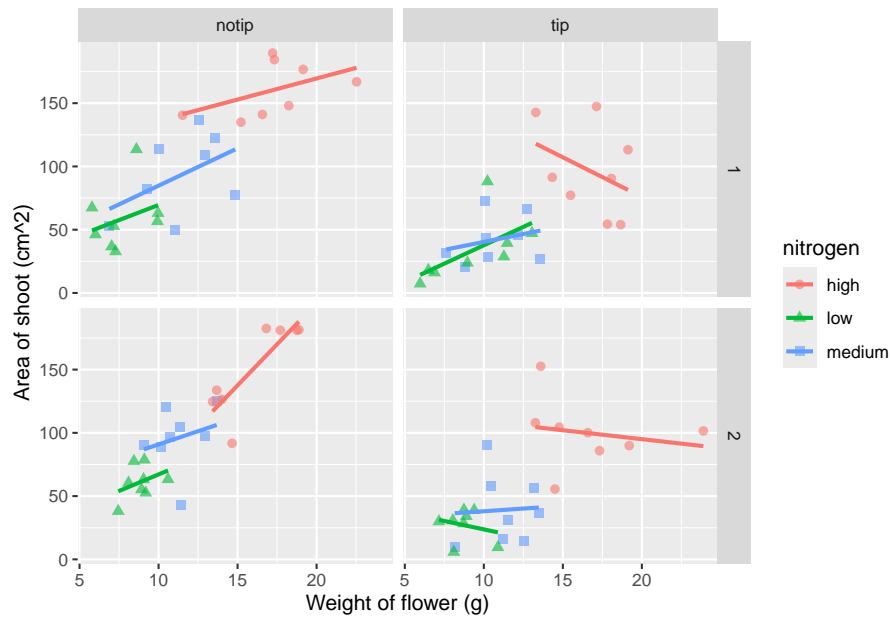
Great. Just as we wanted, though getting the "(cm^2)" to show the square as a super-script would be ideal. Here, we're going to accomplish that using a function which is part of base R called `bquote()` which allows for special characters to be shown.

```r
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat) +
    xlab("Weight of flower (g)") +
    # Using bquote to get mathematically correct formatting
    ylab(bquote("Area of shoot"~(cm^2)))
```
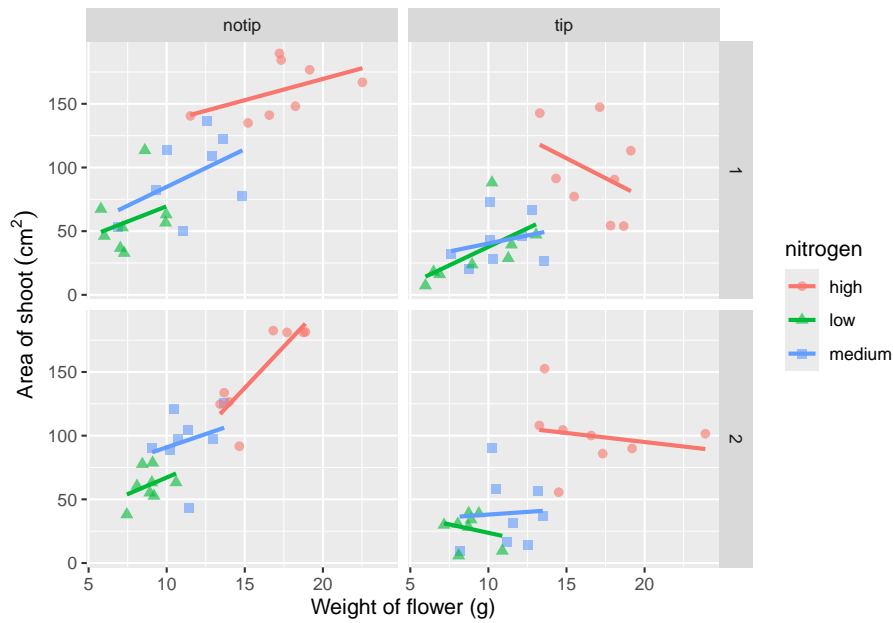
Let's now work on the legend title while also including a caption to warn people looking at the figure to treat the trend lines with caution. We'll use a new layer called `labs()`, short for labels, which we could have also used for specifying the x and y axes labels (we didn't only for demonstration purposes, but give it a shot). `labs()` is a fairly straight-forward function. Have a look at the help file (using `?labs`) to see which arguments are available. We'll be using `caption =` argument for our caption, but notice that there isn't a single simple argument for `legend =`? That's because the legend actually contains multiple pieces of information; such as the colour and shape of the symbols. So instead of `legend =` we'll use `colour =` and `shape =`. Here's how we do it:

```
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    # Adding labels for shape, colour and a caption
    labs(shape = "Nitrogen Concentration", colour = "Nitrogen Concentration",
            caption = "Regression assumptions are unvalidated")
```
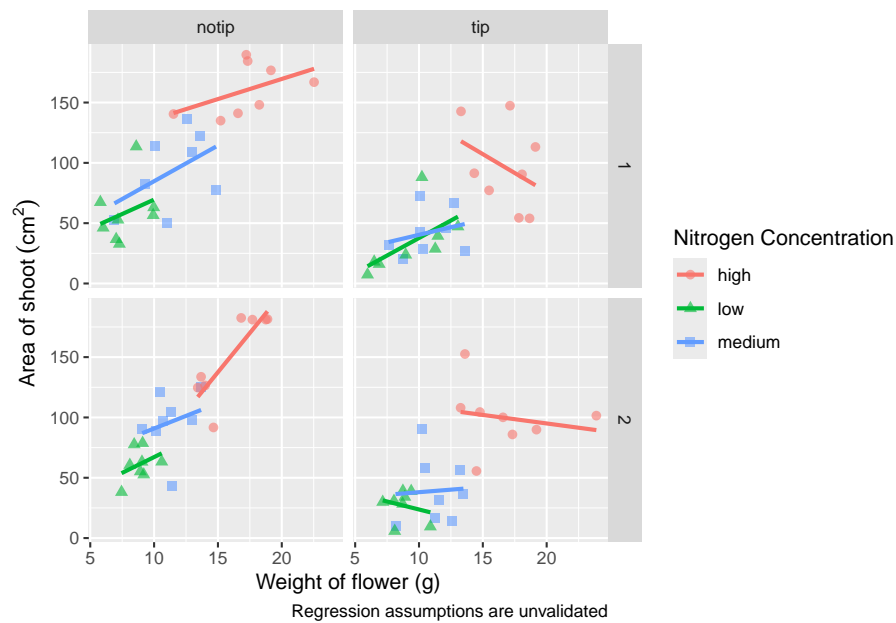
Play around: Try removing `colour =` or `shape =` from `labs()` to see what happens. The resulting legends are why we need to specify both colour and shape (and call it the same thing).

Now's a good time to introduce the `\n`. This is a base R feature that tells R that a string should be continued on a new line. We can use that with "Nitrogen Concentration" so that the legend title becomes more compact.

```r
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    # Including \n to split legend title over two lines
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
            caption = "Regression assumptions are unvalidated")
```
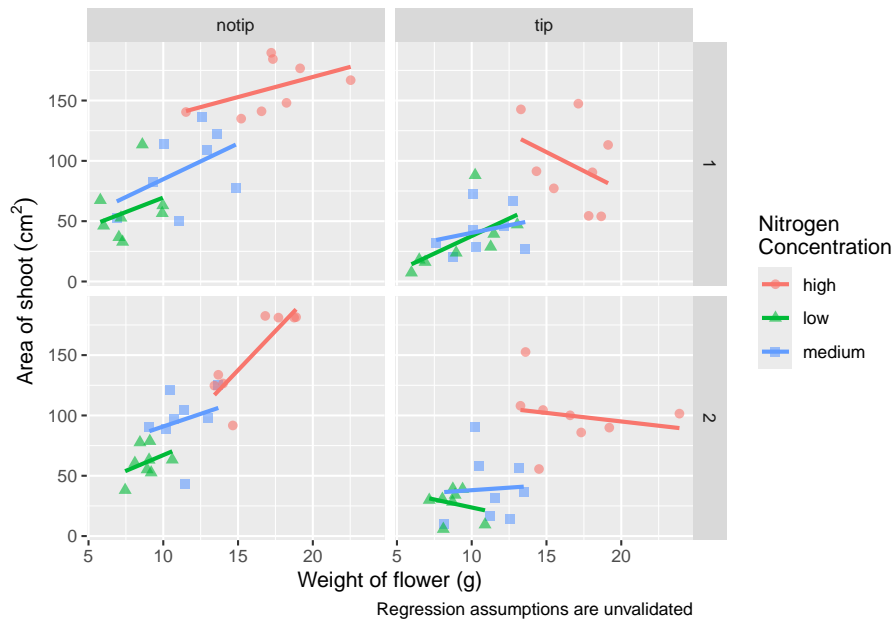
We can now move onto some more wholesale-stylistic choices using the themes layer.

5.2.5   Setting the theme

Themes control the general style of a `ggplot` (things like the background colour, size of text etc.) and comes with a whole bunch of predefined themes. Let's play around with themes using some skills we've already learnt; assigning plots to an object and plotting multiple `ggplots` in a single figure using `patchwork`. We assign themes by creating a new layer with the general notation - `theme_NameOfTheme()`. For example, to use the `theme_classic`, `theme_bw`, `theme_minimal` and `theme_light` themes


```r
classic <- ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
            caption = "Regression assumptions are unvalidated") +
    # Classic theme
    theme_classic()
```

```r
bw <- ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
            caption = "Regression assumptions are unvalidated") +
    # Black and white theme
    theme_bw()

minimal <- ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
            caption = "Regression assumptions are unvalidated") +
    # Minimal theme
    theme_minimal()

light <- ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
      caption = "Regression assumptions are unvalidated") +
    # Light theme
    theme_light()

(classic | bw) /
    (minimal | light)
```
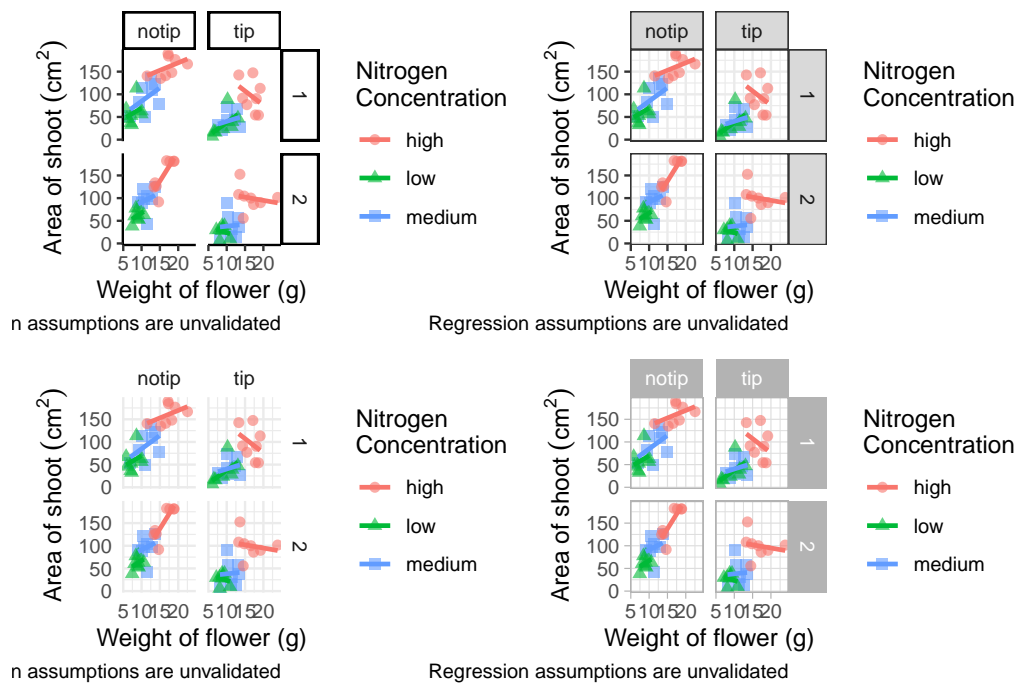
In terms of finding a theme that most closely matches our "final figure", it's probably going to be `theme_classic()`. There are additional themes available to you, and even more available online. `ggthemes` is a package which contains many more themes for you to use. The BBC even have their own `ggplot2` theme called "BBplot" which they use when making their own figures (while good, we don't like it too much for scientific figures). Indeed, you can even make your own theme which is what we'll work on next. To begin with, we'll have a look to see how `theme_classic()` was coded. We can do that easily enough by just writing the function name without the parentheses (see Chapter 7 for a bit more on this).

```
theme_classic
function (base_size = 11, base_family = "", base_line_size = base_size/22,
    base_rect_size = base_size/22)
{
    theme_bw(base_size = base_size, base_family = base_family,
        base_line_size = base_line_size, base_rect_size = base_rect_size) %+replace%
        theme(panel.border = element_blank(), panel.grid.major = element_blank(),
            panel.grid.minor = element_blank(), axis.line = element_line(colour = "black",
                linewidth = rel(1)), strip.background = element_rect(fill = "white",
                colour = "black", linewidth = rel(2)), complete = TRUE)
}
<bytecode: 0x7f8d36b0fbd8>
<environment: namespace:ggplot2>
```

Let's use this code as the basis for our own theme and modify it according to our needs. We'll call the theme, `theme_rbook`. Not all of the options will immediately make sense,

but don't worry about this too much for now. Just know that the settings we're putting
in place are:

- Font size for axis titles = 13
- Font size for x axis text = 10
- Font size for y axis text = 10
- Font for caption = 10 and italics
- Background colour = white
- Background border = black
- Axis lines = black
- Strip colour (for facets) = light blue
- Strip text colour (for facets) = black
- Legend box colours = No colour

This is by no means an exhaustive list of features you can specify in your own theme,
but this will get you started. Of course, there's no need to use a personalised theme as
the pre-built options are perfectly suitable.


```r
theme_rbook <- function(base_size = 13, base_family = "", base_line_size = base_size/22,
                        base_rect_size = base_size/22) {
  theme(
    axis.title = element_text(size = 13),
    axis.text.x = element_text(size = 10),
    axis.text.y = element_text(size = 10),
    plot.caption = element_text(size = 10, face = "italic"),
    panel.background = element_rect(fill="white"),
    axis.line = element_line(size = 1, colour = "black"),
    strip.background =element_rect(fill = "#cddcdd"),
    panel.border = element_rect(colour = "black", fill=NA, size=0.5),
        strip.text = element_text(colour = "black"),
        legend.key=element_blank()
  )
}
```

`theme_rbook()` is now available for us to use just like any other theme. Let's try remaking
our figure using our new theme.

```r
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
            caption = "Regression assumptions are unvalidated") +
    # Updated theme to our theme_rbook
    theme_rbook()    # use our new theme
```
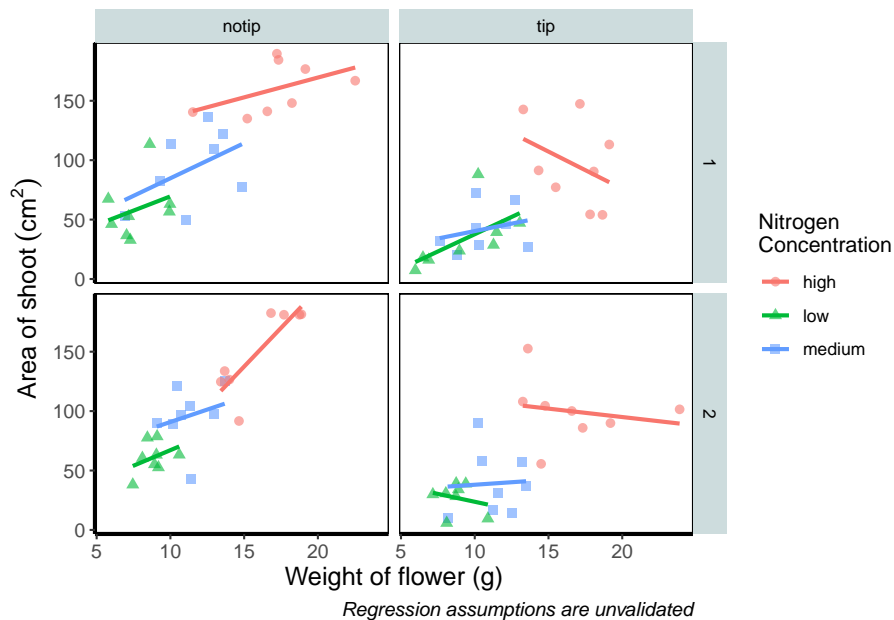

5.2.6   Prettification

We've pretty much replicated our "final figure". We just have a few final adjustments to make, and we'll do so in order of difficulty.

Let's remind ourselves of what that "final figure" looked like. Remember, since we've previously stored the figure as an object called `final_figure` we can just type that into the console and pull up the figure.

```
final_figure +
    labs(title = "Reminder of the final figure")
```


Reminder of the final figure
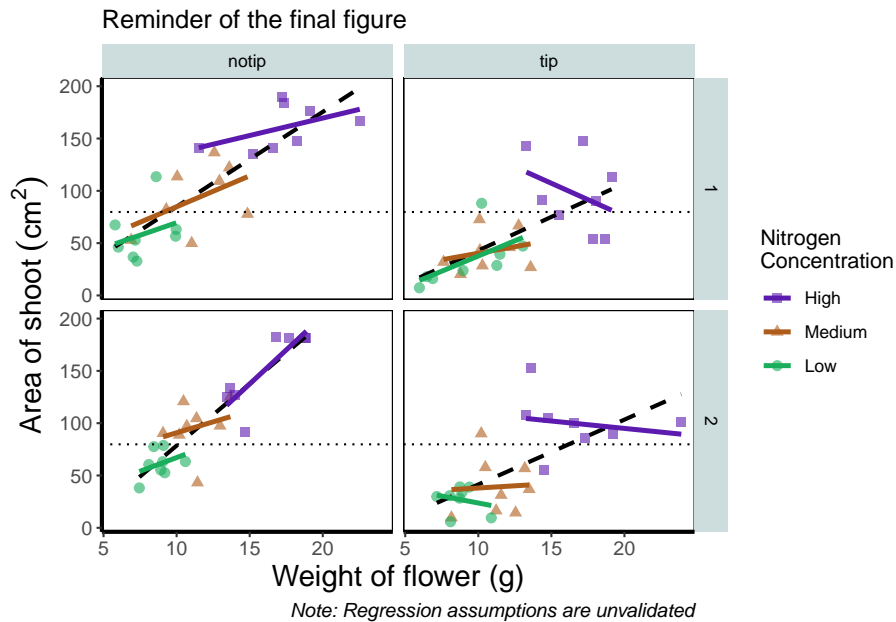
Let's begin the final push by including that dashed horizontal line at the average shoot area, at about 80, on our y axis. This represents the overall mean area of a shoot, regardless of nitrogen concentration, treatment, or block. To draw a horizontal line we use a geom called `geom_hline()`, and the most important thing we need to specify is the y intercept value (in this case the mean area of a shoot). We can also change the type of line using the argument `linetype =` and also the colour (as we did before). Let's see how it works.


```
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
            caption = "Regression assumptions are unvalidated") +
    # Added a horizontal line using geom_hline
    geom_hline(aes(yintercept = mean(shootarea)), size = 0.5, colour = "black", linetype = 3) +
```

```
theme_rbook()
```


Regression assumptions are unvalidated
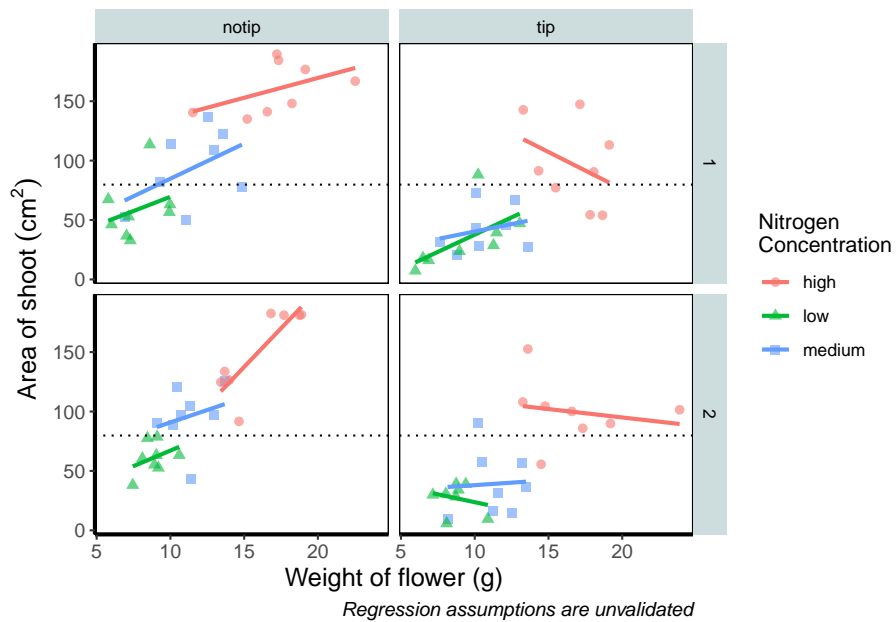
Notice how we included the function `mean(shootarea)` within the `geom_hline()` function? We could also do that externally to the `ggplot2` code and get the same result.

```
mean(flower$shootarea)
## [1] 79.78333
```

```
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
         caption = "Regression assumptions are unvalidated") +
    # Manually entering mean value
    geom_hline(aes(yintercept = 79.8), size = 0.5, colour = "black", linetype = 3) +

    theme_rbook()
```

Regression assumptions are unvalidated
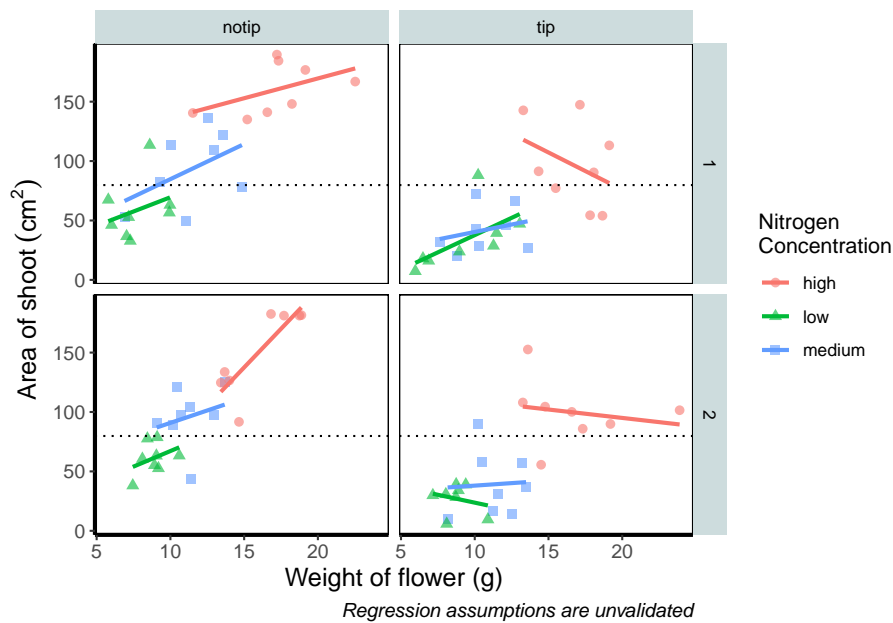
Exactly the same figure but produced in a slightly different way (the point being that there are always multiple ways to get what you want).

Now let's tackle that "overall" nitrogen effect. This overall line is effectively the figure we produced much earlier when we learnt how to include a line of best fit from a linear model. However, we are already using `geom_smooth()`, surely we can't use it again? This may shock and/or surprise you so please ensure you are seated. You *can* use `geom_smooth()` again. In fact you can use it as many times as you want. You can use any layer as many times as you want! Isn't the world full of wonderful miracles? ... Anyway, here's the code...


```r
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    # Adding a SECOND geom_smooth :O
    geom_smooth(method = "lm", se = FALSE, linetype = 2, alpha = 0.6, colour = "black") +
    facet_grid(block ~ treat) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
         caption = "Regression assumptions are unvalidated") +
    geom_hline(aes(yintercept = 79.8), size = 0.5, colour = "black", linetype = 3) +
    theme_rbook()
```

Regression assumptions are unvalidated
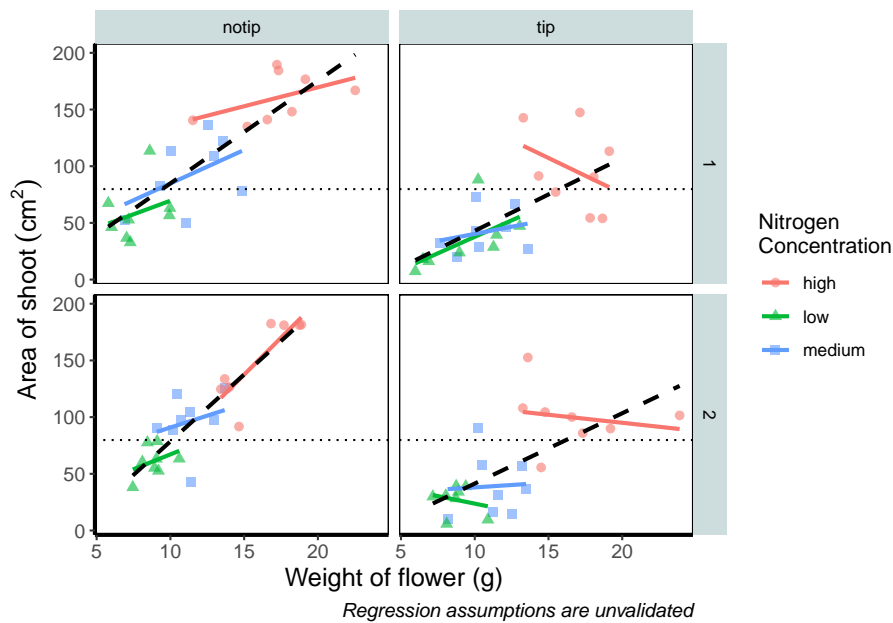
That's great! But you should be asking yourself why that worked. Why when we specified the first `geom_smooth()` did it draw 3 lines, whereas the second time we used `geom_smooth()` it just drew a single line? The secret lies in a "conflict" (it isn't actually a conflict but that's what we'll call it) between the colour specified in the main call to `ggplot()` and the colour specified in the second `geom_smooth()`. Notice how in the second we've specifically told `ggplot2` that the colour will be black, while prior to this it drew lines based on the number of groups (or colours) in nitrogen? In "overriding" the universal `ggplot()` with a geom specific argument we're able to get `ggplot2` to plot what we want.

The only things left to do are to change the colour and the shape of the points to something of our choosing and include information on the "overall" trend line in the legend. We'll begin with the former; changing colour and shape to something we specifically want. When we first started using `ggplot2` this was the thing which caused us the most difficulty. We think the reason is, that to manually change the colours actually requires an additional layer, where we assumed this would be done in either the main call to `ggplot()` or in a geom.

Instead of doing this within the specific geom, we'll use `scale_colour_manual()` as well as `scale_shape_manual()`. Doing it this way will allow us to do two things at once; change the shape and colour to our choosing, and assign labels to these (much like what we did with `xlab()` and `ylab()`). Doing so is not too complex but will require nesting a function (`c()`) within our `scale_colour_manual` and `scale_shape_manual` functions (see Chapter 2 for a reminder on the concatenate function (`c()`) if you've forgotten).

Choosing colours can be fiddly. We've found using a colour wheel helps with this step. You can always use Google to find an interactive colour wheel, or use Google's Colour picker. Any decent website should give you a HEX code, something like: #5C1AAE which is a "code" representation of a colour. Alternatively, there are colour names

which R and `ggplot2` will also understand (e.g. "firebrick4"). Having chosen our colours using whichever means, let's see how we can do it:



```r
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    geom_smooth(method = "lm", se = FALSE, linetype = 2, alpha = 0.6, colour = "black") +
    facet_grid(block ~ treat) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
            caption = "Regression assumptions are unvalidated") +
    geom_hline(aes(yintercept = 79.8), size = 0.5, colour = "black", linetype = 3) +
    # Setting colour and associated labels
    scale_colour_manual(values = c("#5C1AAE", "#AE5C1A", "#1AAE5C"),
        labels = c("High", "Medium", "Low")) +
    # Setting shape and associated labels
    scale_shape_manual(values = c(15,17,19),
        labels = c("High", "Medium", "Low")) +

    theme_rbook()
```
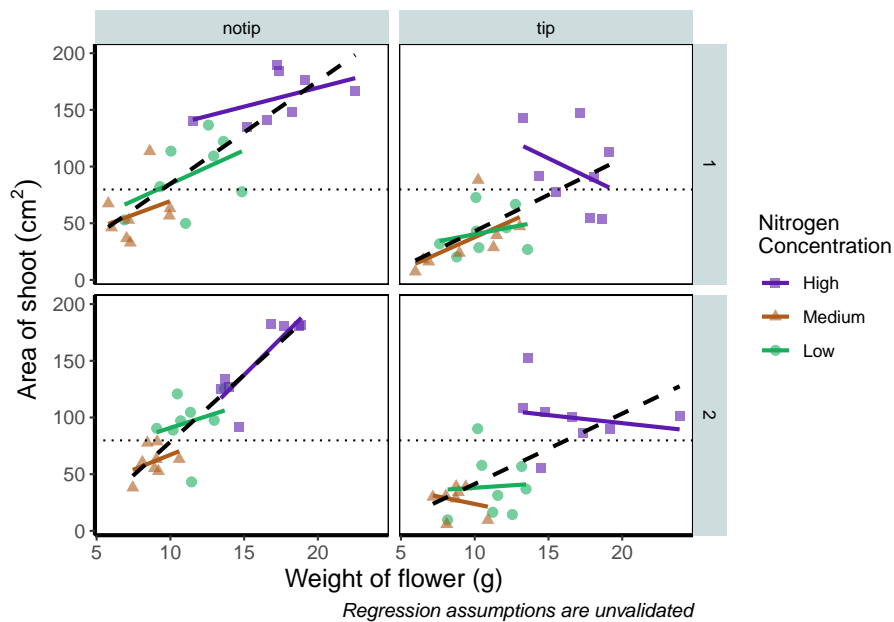
To make sense of that code (or any code for that matter) try running it piece by piece. For instance in the above code, if we run `c("#5C1AAE", "#AE5C1A", "#1AAE5C")` we'll get a list of those strings. That list is then passed on to `scale_colour_manual()` as the colours we wish to use. Since we only have three nitrogen concentrations, it will use these three colours. Try including an additional colour in the list and see what happens (if you place the new colour at the end of the list, nothing will happen since it will use the first three colours of the list - try adding it to the start of the list). The same is true for `scale_shape_manual()`.

But if you're paying close attention you'll notice that there's a mistake with the figure now. What should be labelled "Low" is actually labelled "Medium" (the green points and line are our low nitrogen concentration, but `ggplot2` is saying that it's purple). `ggplot2` hasn't made a mistake here, we have. Remember that code is purely logical. It will do explicitly what it is told to do, and in this case we've told it to call the labels High, Medium and Low. We could have just as easily told `ggplot2` to call them Pretoria, Tokyo, and Copenhagen. The lesson here is to always be critical of what your outputs are. Double check everything you do.

So how do we fix this? We need to do a little data manipulation to rearrange our factors so that the order goes High, Medium, Low. Let's do that:

```
flower$nitrogen <- factor(flower$nitrogen, levels = c("high", "medium", "low"))
```

With that done, we can re-run our above code to get a correct figure and assign it the name "rbook_figure".

```
rbook_figure <- ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    geom_smooth(method = "lm", se = FALSE, linetype = 2, alpha = 0.6, colour = "black") +
    facet_grid(block ~ treat) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
            caption = "Regression assumptions are unvalidated") +
    geom_hline(aes(yintercept = 79.8), size = 0.5, colour = "black", linetype = 3) +
    scale_colour_manual(values = c("#5C1AAE", "#AE5C1A", "#1AAE5C"),
        labels = c("High", "Medium", "Low")) +
    scale_shape_manual(values = c(15,17,19),
        labels = c("High", "Medium", "Low")) +
    theme_rbook()

rbook_figure
```

Regression assumptions are unvalidated
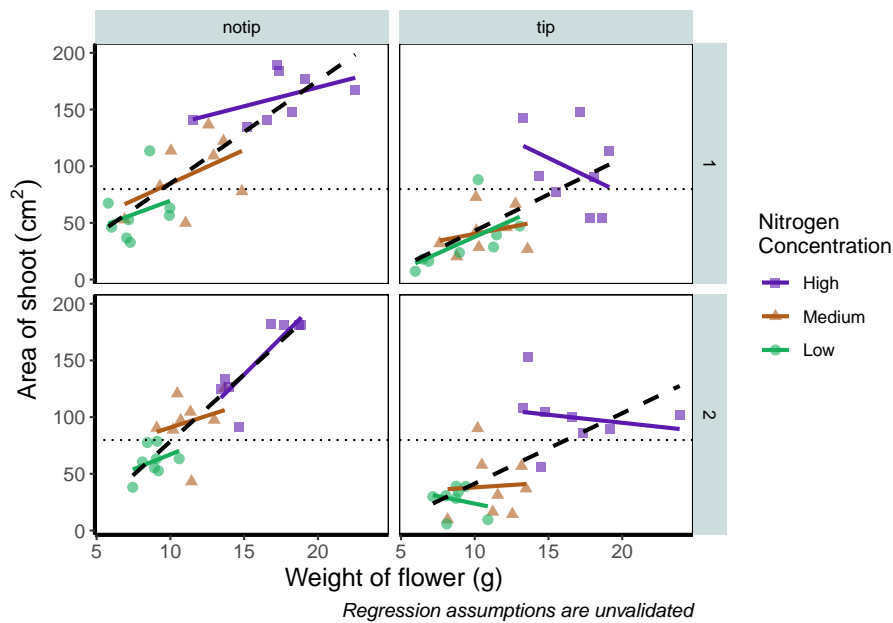
And we've done it! Our final figure matches the "final figure" exactly. We can then save the final figure to our computer so that we can include it in a poster etc. The code for this is straightforward, but does require an understanding of file paths. Be sure to check Chapter 1 for an explanation if you're unsure. To save `ggplot` figures to an external file we use the function `ggsave`.

This is the point when having assigned the code to the object named `rbook_figure` comes in handy. Doing so allows us to specify which figure to save within `ggsave()`. If we hadn't specified which plot to save, `ggsave()` would instead save the last figure produced.

Other important arguments to take note of are: `device =` which tells `ggplot2` what format we want to save the figure (in this case a pdf) though `ggplot2` is often smart enough to guess this based on the extension we give our file name, so it is often redundant; `units =` which specifies the units used in `width =` and `height =`; `width =` and `height =` specify the width and height of figure (in this case in mm as specified using `units =`); `dpi =` which controls the resolution of the saved figure; and `limitsize =` which prevents accidentally saving a massive figure of 1 km x 1 km!

```
ggsave(filename = "areashoot_weight_facet.pdf", plot = rbook_figure, device = "pdf",
        path = "output", width = 250, height = 150, units = "mm",
        dpi = 500, limitsize = TRUE)
```

This concludes the worked example to reproduce our final figure. While absolutely not an exhaustive list of what you can do with `ggplot2`, this will hopefully help when you're making your own from scratch or, perhaps likely when starting, copying `ggplots` made by other people (in which case hopefully this will help you understand what they've done).

Two sections follow this one. The first is simply a collection of tips and tricks that couldn't quite make their way into this worked example, and the second section is a collection of different geometries to give you a feel for what's possible. We would recommend only glancing through each section and coming back to them as and when you need to know how to do *that particular thing.*

Hopefully you've enjoyed reading through this Chapter, and more importantly, that it's increased your confidence in making figures. Here are two final parting *words of wisdom!.* First, as we've said before, don't fall into the trap of thinking your figures are superior to those who don't use `ggplot2`. As we've mentioned previously, equivalent figures are possible in both base R and `ggplot2`. The only difference is how you get to those figures.

Secondly, don't overcomplicate your figures. With regards to the final figure we produced here, we've gone back and forth as to whether we are guilty of this. The figure does contain a lot of information, drawing on information from five different variables, with one of those being presented in three different ways. We would be reluctant, for example, to include this in a presentation as it would likely be too much for an audience to fully appreciate in the 30 seconds to 1 minute that they'll see it. In the end we decided it was worth the risk as it served as a nice demonstration. Fortunately, or unfortunately depending on your view, there are no hard and fast rules when it comes to making figures. Much of it will be at your discretion, so please ensure you give it some thought.

5.3 Tips and tricks

For this section we'll use various versions of the final figure without including the `facet_grid()`. We do so only to allow the changes to be more apparent. With this plot, we'll run through some tips and tricks that we wish we'd learnt when we started using `ggplot2`.

5.3.1 Statistics layer

The statistics layer is often ignored in favour of working solely with the geometry layer, as we've done above. The two are largely interchangeable, though the relative rareness of online help and discussions on the statistics layer seems to have relegated it to almost a state of anonymity. There is real value in at least understanding what the statistics layer is doing, even if you don't ever need to make direct use of it. It is perhaps most clear what the statistical layer is doing in the early `geom_smooth()` figure we made.

```
ggplot(aes(x = weight, y = shootarea), data = flower) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
```

```
## `geom_smooth()` using formula = 'y ~ x'
```
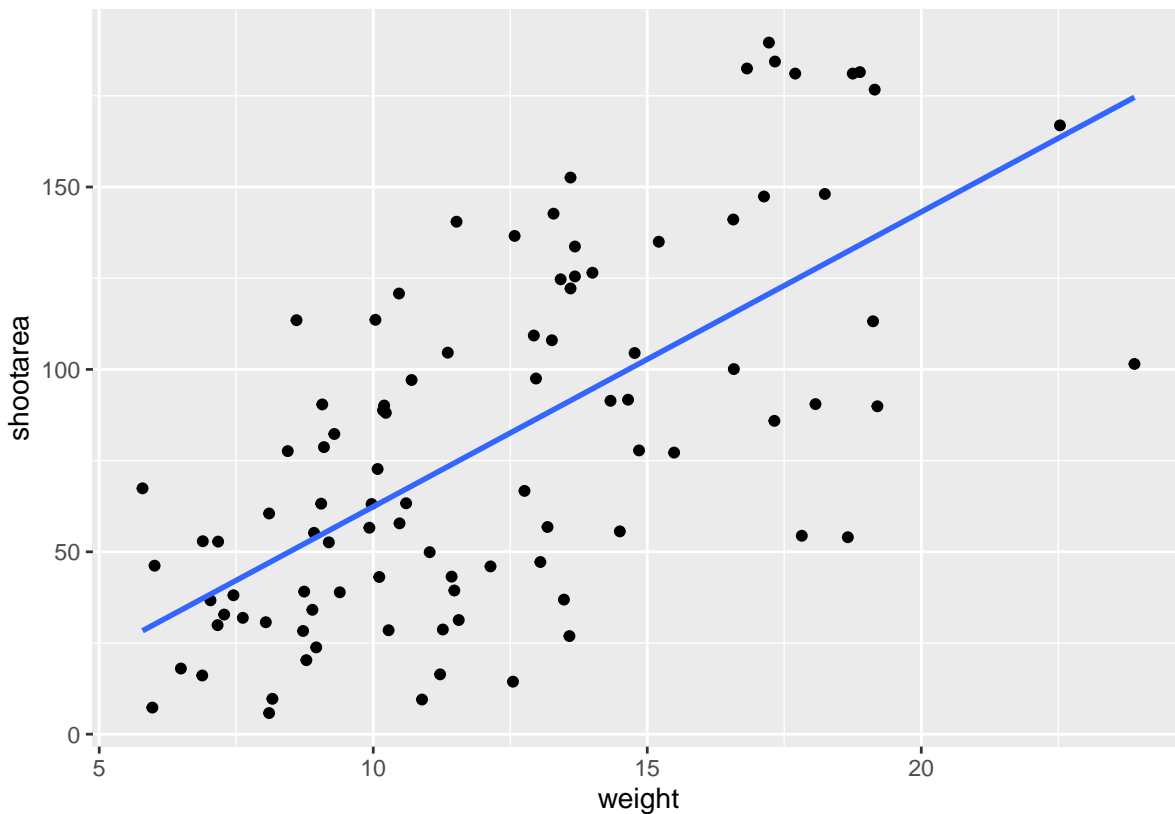

Nowhere in our dataset are there columns for either the y-intercept or the gradient needed to draw the straight line, yet we've managed to draw one. The statistics layer calculates these based on our data, without us necessarily knowing what we've done. It's also the engine behind converting your data into counts for producing a bar chart, or densities for violin plots, or summary statistics for boxplots and so on.

It's entirely possible that you'll be able to use `ggplot2` for the vast majority of you plotting without ever consulting the statistics layer in any more detail than we have here (simply by "calling" - unknowingly - to it via the geometry layer), but be aware that it exists.

If we wanted to recreate the above figure using the statistics layer we would do it like this:


```
ggplot(aes(x = weight, y = shootarea), data = flower) +
  geom_point() +
    # using stat_smooth instead of geom_smooth
  stat_smooth(geom = "smooth", method = "lm", se = FALSE)
```
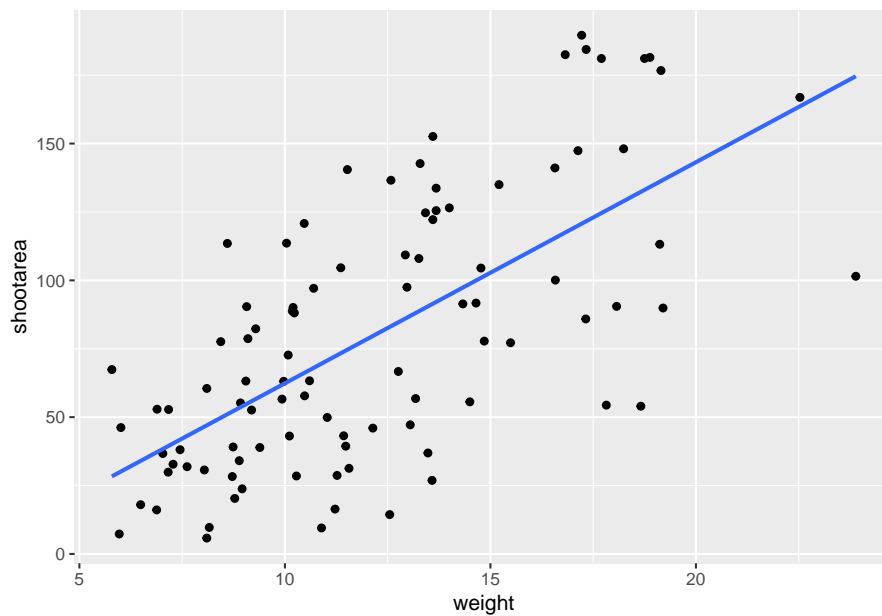
While in this example it doesn't make a difference which we use, in other cases we may want to use the calculated statistics in alternative ways. We won't get into it, but see `?after_stat` if you are interested.

5.3.2   Axis limits and zooms

Fairly often, you may want to limit the range of your axes. Maybe you want to focus a particular part of the data to really tease apart any patterns occurring there. Whatever the reason, it's a useful skill, and with most things code related, there's a couple of ways to do this. We'll show two here; `xlim()` and `ylim()`, and `coord_cartesian()`. Using both of these we'll set the x axis to only show data between 10 and 15 g and the y axis to only show the area of the shoot between 50 and 150 mm^2. We'll start with limiting the axes:



```
ggplot(aes(x = weight, y = shootarea), data = flower) +
    geom_point(aes(colour = nitrogen, shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(colour = "black", method = "lm", se = FALSE, linetype = 2, alpha = 0.6) +
    geom_smooth(aes(colour = nitrogen), method = "lm", se = FALSE, size = 1.2) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    geom_hline(aes(yintercept = 79.7833), size = 0.5, colour = "black", linetype = 3) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
```

```
           caption = "Regression assumptions are unvalidated") +
   scale_colour_manual(values = c("#5C1AAE", "#AE5C1A", "#1AAE5C"),
       labels = c("High", "Medium", "Low")) +
   scale_shape_manual(values = c(15,17,19,21),
       labels = c("High", "Medium", "Low")) +
   theme_rbook() +
   # New x and y limits
   xlim(c(10, 15)) +
   ylim(c(50, 150))
```


Regression assumptions are unvalidated
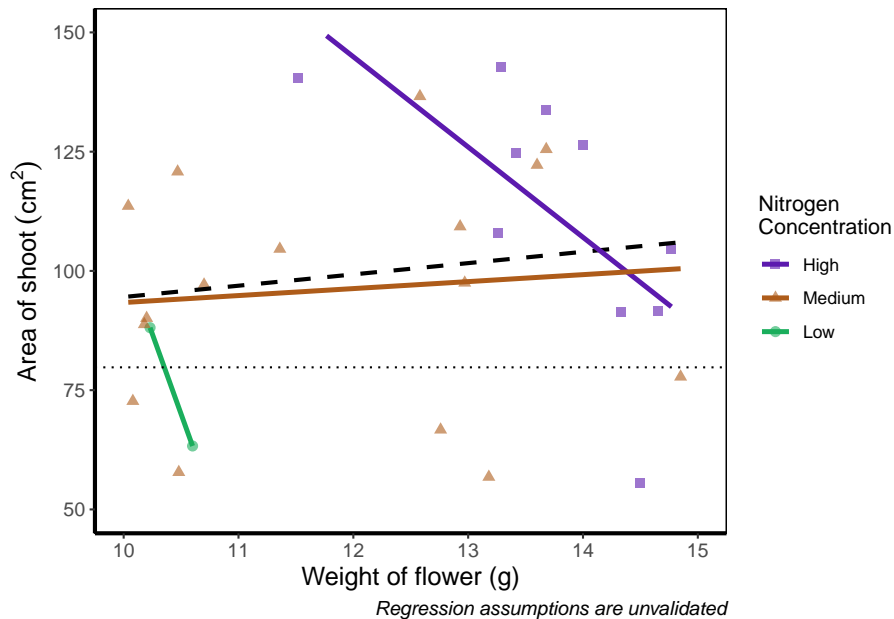
If you run this yourself you'll see warning messages telling us that n rows contain either missing or non-finite values. That's because we've essentially chopped out a huge part of our data using this method (everything outside of the ranges that we specified is "removed" from the data). As a result of doing this our lines have now completely changed direction. Notice that for low nitrogen concentration, the line is being drawn using only two points? This may, or may not be a problem depending on the aim we have, but we can use an alternative method; `coord_cartesian()`.

`coord_cartesian()` works in much the same way, but instead of chopping out data, it instead zooms in. Doing so means that the entire dataset is maintained (and any trends are maintained).

```
ggplot(aes(x = weight, y = shootarea), data = flower) +
    geom_point(aes(colour = nitrogen, shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(colour = "black", method = "lm", se = FALSE, linetype = 2, alpha = 0.6) +
    geom_smooth(aes(colour = nitrogen), method = "lm", se = FALSE, size = 1.2) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    geom_hline(aes(yintercept = 79.7833), size = 0.5, colour = "black", linetype = 3) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
            caption = "Regression assumptions are unvalidated") +
    scale_colour_manual(values = c("#5C1AAE", "#AE5C1A", "#1AAE5C"),
        labels = c("High", "Medium", "Low")) +
    scale_shape_manual(values = c(15,17,19,21),
        labels = c("High", "Medium", "Low")) +
    theme_rbook() +
    # Zooming in rather than chopping out
    coord_cartesian(xlim = c(10, 15),   ylim = c(50, 150))
```


Regression assumptions are unvalidated
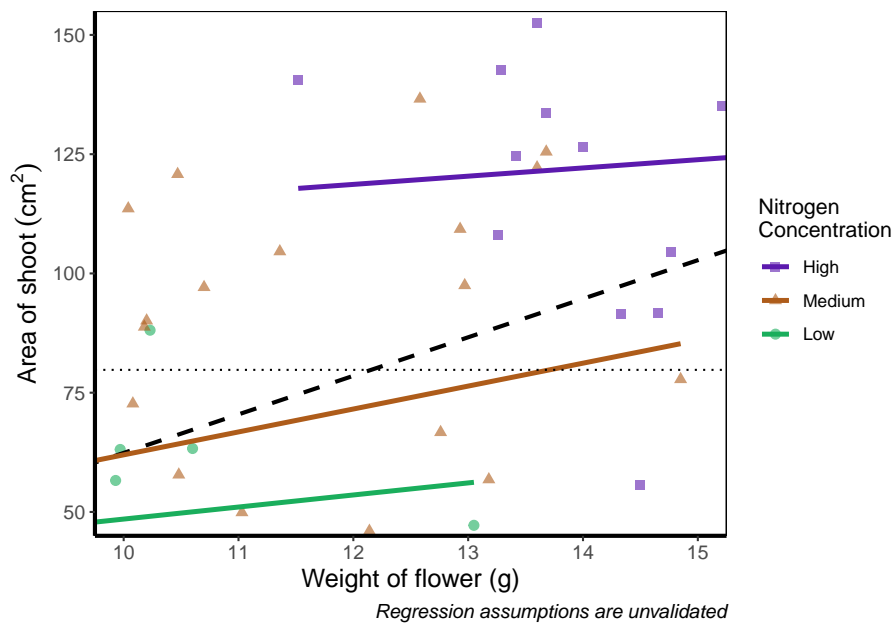
Notice now that the trends are maintained (as the lines are being informed by data which are off-screen). We would generally advise using `coord_cartesian()` as it protects you against possible misinterpretations.

5.3.3 Layering layers

Remember that `ggplot2` works like painting. Let's imagine we were one of the great renaissance painters. We've just finished the focal point of our painting, the half naked Duke of Toulouse looking moody or something. Now that we've finished painting the

honourable Duke, we proceed to paint the background, a beautiful landscape showing the Pyrenees mountains in the distance. Unfortunately, in doing so we've painted over the Duke because the order of our layers was wrong. We get our heads chopped off, but learn a valuable lesson in the process: the order of layers matter.

The exact same is true in ggplot2 (minus the chopping off of heads, though your situation may vary). The layers are read and "painted" in order of their appearance in the code. If geom_point() comes before geom_col(), then your points may well end up being hidden. To fix this is easy, we simply need to move the layers up or down in the code. A useful tip for those using RStudio, is that you can move lines of code especially easily. Simply click on a line of code, hold down Alt and then press either the up or down arrows, and the entire line will move up or down as well. For multiple lines of code, simply highlight all those lines you want to move and press Alt + Up/Down arrow.

We'll move both the points and the overall black dashed line down in the code so that they are superimposed over the nitrogen lines:

```r
ggplot(aes(x = weight, y = shootarea), data = flower) +
    geom_smooth(aes(colour = nitrogen), method = "lm", se = FALSE, size = 1.2) +
    # Move one line down
    geom_point(aes(colour = nitrogen, shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(colour = "black", method = "lm", se = FALSE, linetype = 2, alpha = 0.6) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    geom_hline(aes(yintercept = 79.7833), size = 0.5, colour = "black", linetype = 3) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
           caption = "Regression assumptions are unvalidated") +
    scale_colour_manual(values = c("#5C1AAE", "#AE5C1A", "#1AAE5C"),
       labels = c("High", "Medium", "Low")) +
    scale_shape_manual(values = c(15,17,19,21),
       labels = c("High", "Medium", "Low")) +
    theme_rbook()
```

Regression assumptions are unvalidated
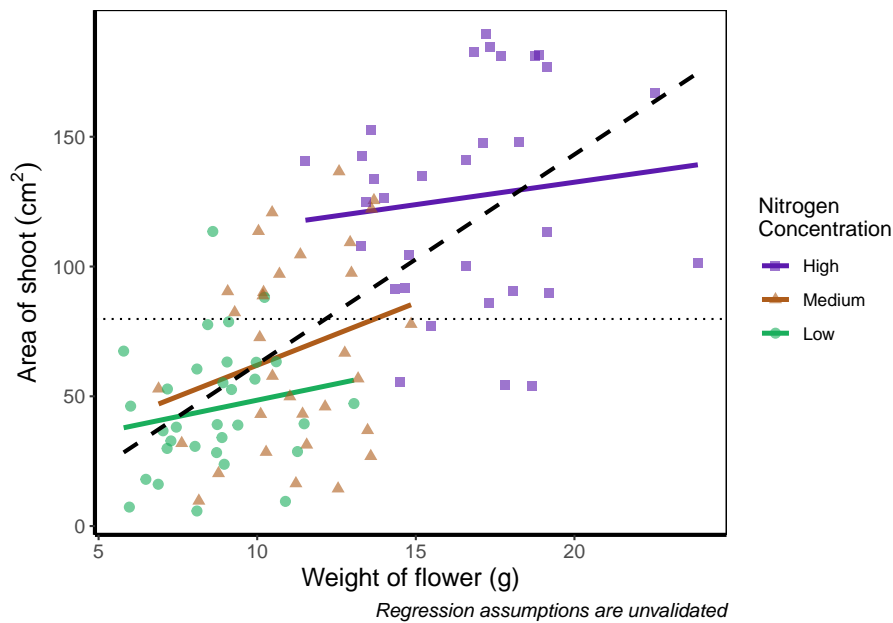
It's not the clearest example, but give it a shot with your own data.

5.3.4   Continuous colours

Instead of categorical colours, such as we've used for nitrogen concentration, what if instead we wanted a gradient? To illustrate this, we'll remove the trend lines to highlight the changes we make. We'll also be using the `flowers` variable (i.e. number of flowers) to specify the colour that points should be coloured. We have three options that we'll use here; the default colour scheme, the `scale_colour_gradient()` scheme, and an alternative scheme`scale_colour_gradient2()`. Remember that we'll also need to change our label for the legend.

We'll start with the default option. Here we only need to change nitrogen (which is a factor) to flowers (which is continuous) in the `colour =` argument within `aes()`:


```
ggplot(aes(x = weight, y = shootarea), data = flower) +
   # Deleted geom_smooths for illustrative purposes only
  # (and also removed alpha argument from geom_point)
   geom_point(aes(colour = flowers, shape = nitrogen), size = 2) +
   xlab("Weight of flower (g)") +
   ylab(bquote("Area of shoot"~(cm^2))) +
   geom_hline(aes(yintercept = 79.7833), size = 0.5, colour = "black", linetype = 3) +
```

```
# Changed colour argument label
labs(shape = "Nitrogen\nConcentration", colour = "Number of\nFlowers",
        caption = "Regression assumptions are unvalidated") +
scale_shape_manual(values = c(15,17,19,21),
    labels = c("High", "Medium", "Low")) +
theme_rbook()
```
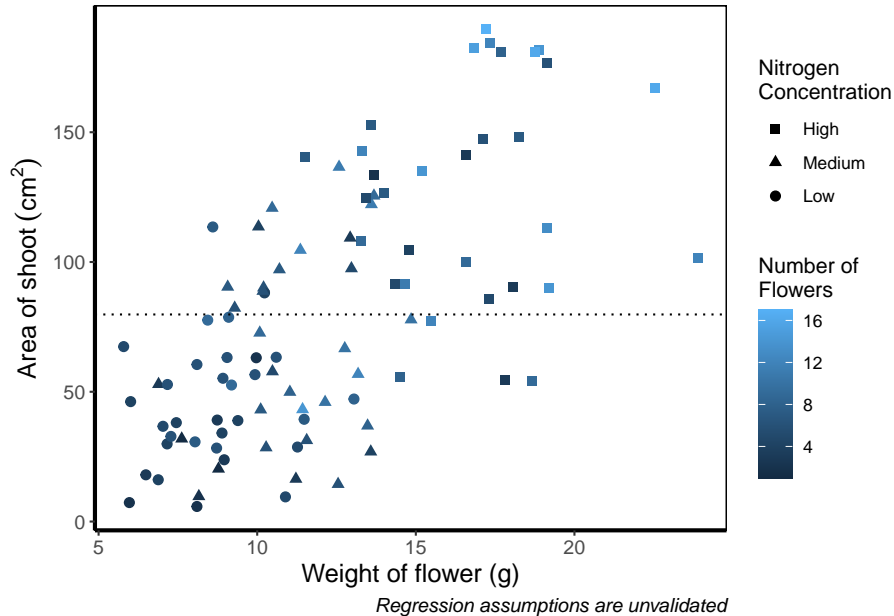

And as easily as that we have a colour gradient to show number of flowers. Dark blue shows low flower numbers and light blue shows higher flower numbers. While the code has worked, we have a hard time distinguishing between the different shades of blue. It would help ourselves (and our audience), if we changed the colours to something more noticeably different, using `scale_colour_gradient()`. This works much as `scale_colour_manual()` except that this time we specify the `low =` and `high =` values using arguments.


```
ggplot(aes(x = weight, y = shootarea), data = flower) +
    geom_point(aes(colour = flowers, shape = nitrogen), size = 2) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    geom_hline(aes(yintercept = 79.7833), size = 0.5, colour = "black", linetype = 3) +
    # Updated legend name for colour
```

```r
    labs(shape = "Nitrogen\nConcentration", colour = "Number of\nFlowers",
         caption = "Regression assumptions are unvalidated") +
    scale_shape_manual(values = c(15,17,19,21),
        labels = c("High", "Medium", "Low")) +
    # Adding scale_colour_gradient
    scale_colour_gradient(low = "#9F00FF", high = "#FF9F00") +
    theme_rbook()
```


Regression assumptions are unvalidated
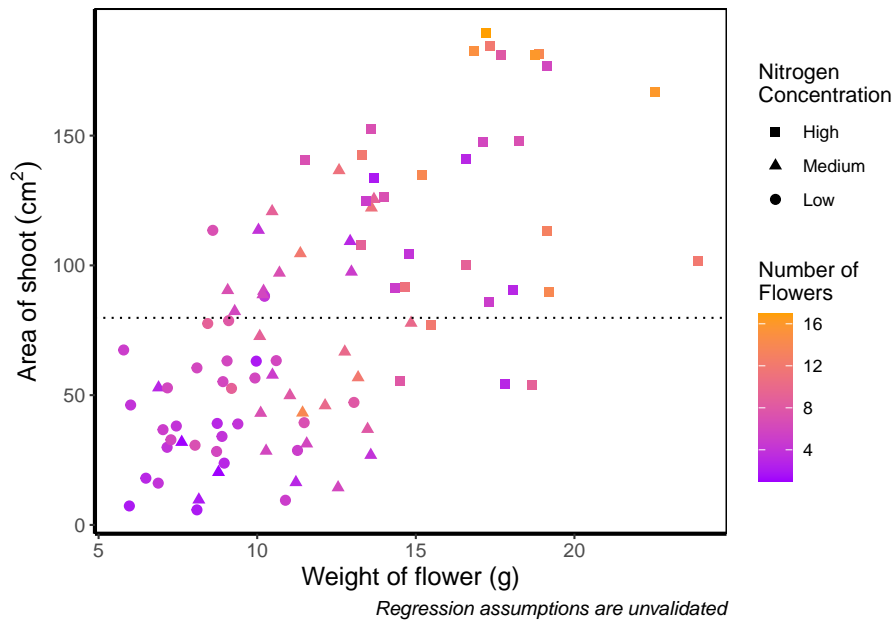
Although arguably better, we still struggle to spot the difference when there are between 5 and 12 flowers. Maybe having an additional colour would help those mid values stand out a bit more. The way we can do that here is to set a midpoint where the colours shift from green to blue to pink. Doing so might help us see the variation even more clearly. This is exactly what `scale_colour_gradient2()` allows. `scale_colour_gradient2()` works in much the same way as `scale_colour_gradient()` except that we have two additional arguments to worry about; `midpoint =` where we specify a value for the midpoint, and `mid =` where we state the colour the midpoint should take. We'll set the midpoint using `mean()`.


```r
ggplot(aes(x = weight, y = shootarea), data = flower) +
    geom_point(aes(colour = flowers, shape = nitrogen), size = 2) +
    xlab("Weight of flower (g)") +
```

```
    ylab(bquote("Area of shoot"~(cm^2))) +
    geom_hline(aes(yintercept = 79.7833), size = 0.5, colour = "black", linetype = 3) +
    labs(shape = "Nitrogen\nConcentration", colour = "Number of\nFlowers",
            caption = "Regression assumptions are unvalidated") +
    scale_shape_manual(values = c(15,17,19,21),
        labels = c("High", "Medium", "Low")) +
    # Adding scale_colour_gradient2
    scale_colour_gradient2(midpoint = mean(flower$flowers),
        low = "#9F00FF", mid = "#00FF9F", high = "#FF9F00") +
    theme_rbook()
```


Regression assumptions are unvalidated
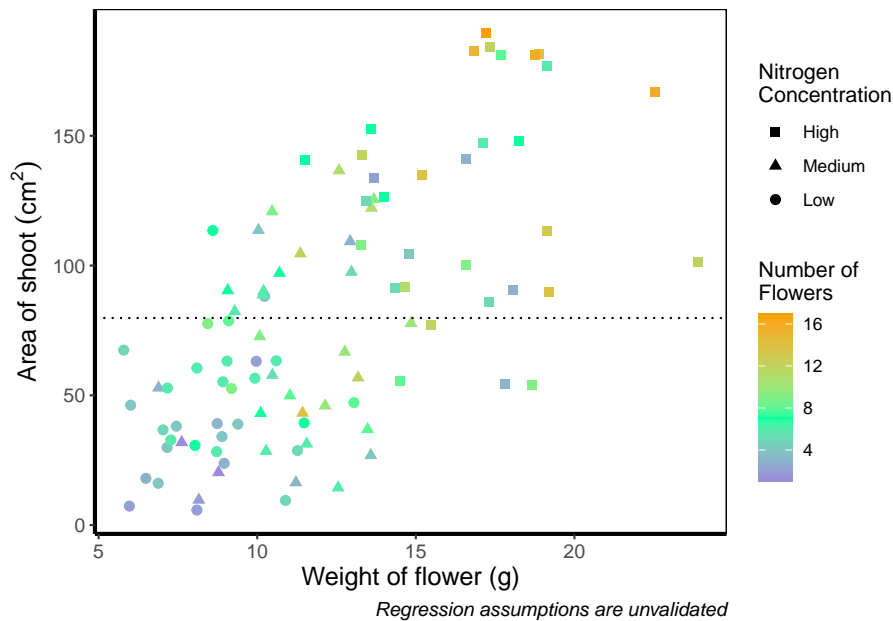
Definitely not our favourite figure. Perhaps if we add **more** colours, that will help things a bit (probably not but let's do it anyway). We now move onto using `scale_colour_gradientn()`, which diverges slightly. Instead of specifying colours for low, mid, and/or high, here we'll be specifying them using proportions within the `values =` argument. A common mistake with `values =`, within `scale_colour_gradient()`, is to assume (justifiably in our opinion) that we'd specify the actual numbers of flowers as our values. This is wrong. Try doing so and you'll likely see a grey colour bar and grey points. Instead `values =` represent the proportional ranges where we want the colour to occupy. In the code below, we use 0, 0.25, 0.5, 0.75 and 1 as our proportions, corresponding to 4 colours (note that we have one fewer colour than proportions given as the colours occupy a range and not a value).

```
ggplot(aes(x = weight, y = shootarea), data = flower) +
    geom_point(aes(colour = flowers, shape = nitrogen), size = 2) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    geom_hline(aes(yintercept = 79.7833), size = 0.5, colour = "black", linetype = 3) +
    labs(shape = "Nitrogen\nConcentration", colour = "Number of\nFlowers",
            caption = "Regression assumptions are unvalidated") +
    scale_shape_manual(values = c(15,17,19,21),
        labels = c("High", "Medium", "Low")) +
    # Adding scale_colour_gradientn
    scale_colour_gradientn(colours = c("#1252ED", "#12ED3F", "#EDAD12", "#ED12C0"),
        values = c(0, 0.25, 0.5, 0.75, 1)) +
    theme_rbook()
```


Regression assumptions are unvalidated
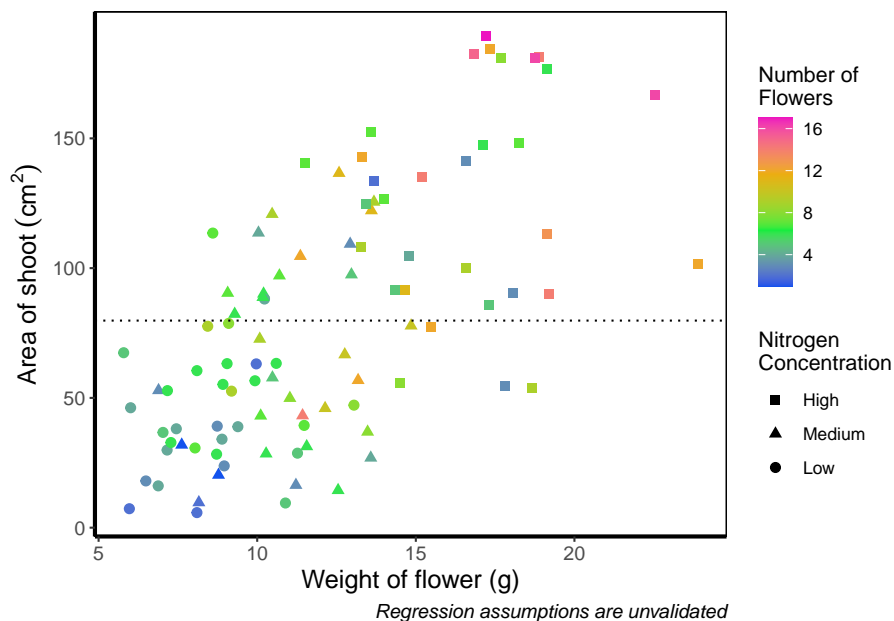
Slightly nauseating but it's doing what we wanted it to do, so we shouldn't really complain.

5.3.5 Size of points

Previously, we altered to size of points to be a constant number (e.g. `size = 2`). What if instead we wanted size to change according to a variable in our dataset? We can do this very easily by including a continuous variable with the `size =` argument.

```r
ggplot(aes(x = weight, y = shootarea), data = flower) +
    # Moving size into aes and changing to a continuous variable
    geom_point(aes(colour = nitrogen, shape = nitrogen, size = height), alpha = 0.6) +
    geom_smooth(colour = "black", method = "lm", se = FALSE, linetype = 2, alpha = 0.6) +
    geom_smooth(aes(colour = nitrogen), method = "lm", se = FALSE, size = 1.2) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    geom_hline(aes(yintercept = 79.7833), size = 0.5, colour = "black", linetype = 3) +
    # Including size label
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
            caption = "Regression assumptions are unvalidated",
            size = "Plant\nHeight (cm)") +
    scale_colour_manual(values = c("#5C1AAE", "#AE5C1A", "#1AAE5C"),
        labels = c("High", "Medium", "Low")) +
    scale_shape_manual(values = c(15,17,19,21),
        labels = c("High", "Medium", "Low")) +
    theme_rbook()
```
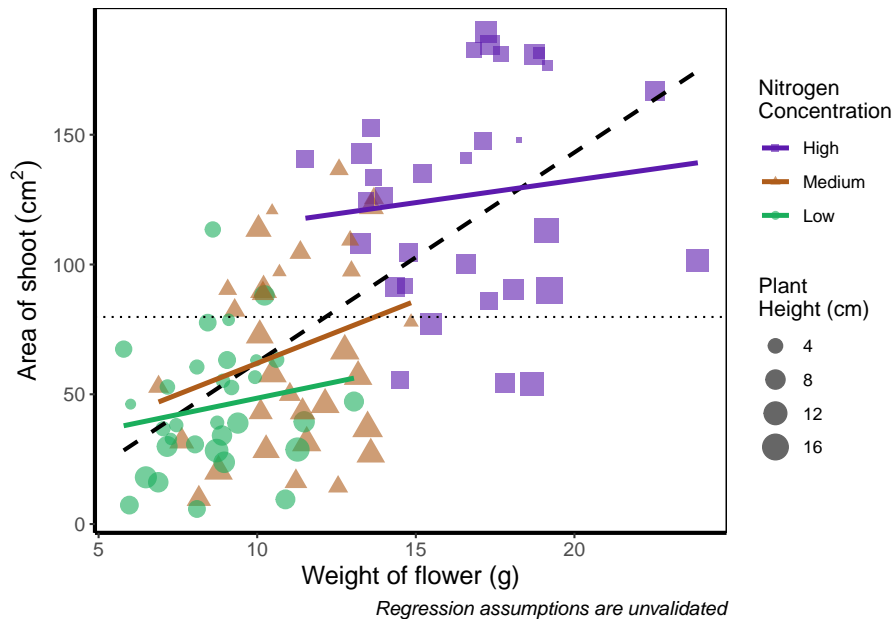

Now the sizes reflect the height of the plants, with bigger points representing taller plants and vice-versa.

5.3.6   Moving the legend

To move the position of the legend requires tweaking the theme, just as we did before with `theme_rbook()`. But for legends we might not want this to be set in stone whenever we use the theme (i.e. coding this into `theme_rbook()`). Instead we can change it *on the*

fly depending on the individual figure. To do so, we can use a `theme()` layer and the argument `legend.position =`, followed swiftly by another layer specifying that we still want to use `theme_rbook()`.



```r
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    geom_smooth(method = "lm", se = FALSE, linetype = 2, alpha = 0.6, colour = "black") +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
            caption = "Regression assumptions are unvalidated") +
    geom_hline(aes(yintercept = 79.8), size = 0.5, colour = "black", linetype = 3) +
    scale_colour_manual(values = c("#5C1AAE", "#AE5C1A", "#1AAE5C"),
        labels = c("High", "Medium", "Low")) +
    scale_shape_manual(values = c(15,17,19),
        labels = c("High", "Medium", "Low")) +
    # Moving the legend
    theme(legend.position = "bottom") +
    theme_rbook()
```
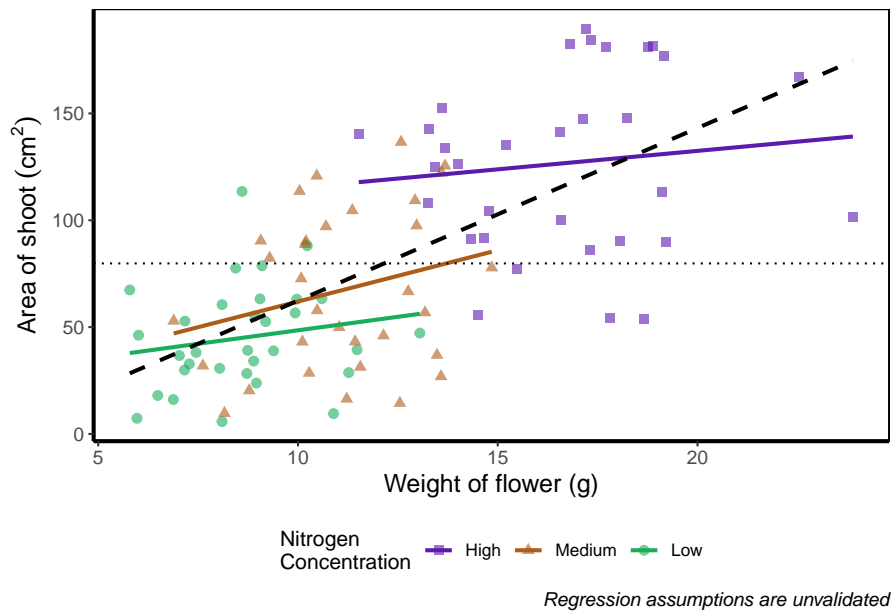

Play around: Try the code above but don't include `theme_rbook()` and see what happens? What about if you try `theme_rbook(legend.position = "bottom")`?

5.3.7 Hiding the legend

Suppose we don't want a legend at all. How would we go about hiding it?


```
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    geom_smooth(method = "lm", se = FALSE, linetype = 2, alpha = 0.6, colour = "black") +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
            caption = "Regression assumptions are unvalidated") +
    geom_hline(aes(yintercept = 79.8), size = 0.5, colour = "black", linetype = 3) +
    scale_colour_manual(values = c("#5C1AAE", "#AE5C1A", "#1AAE5C"),
        labels = c("High", "Medium", "Low")) +
    scale_shape_manual(values = c(15,17,19),
        labels = c("High", "Medium", "Low")) +
    # Hiding the legend
    theme(legend.position = "none") +
    theme_rbook()
```


Regression assumptions are unvalidated
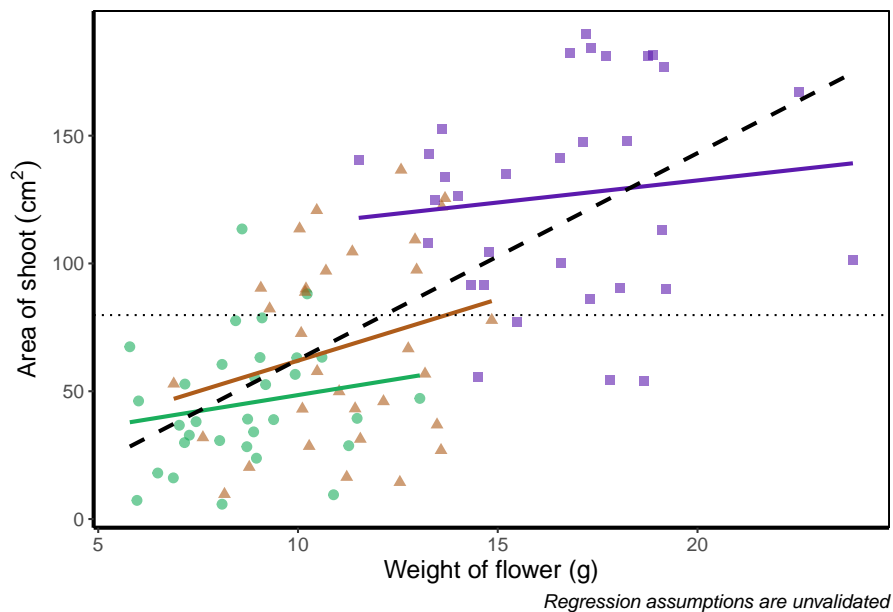
5.3.8 Hiding part of the legend

What if we really don't want points included in the legend? Instead of stating this using `theme()`, we'll include it within `geom_point()` using `show.legend = FALSE`.


```r
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    # Including show.legend = FALSE to prevent inclusion in legend
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6, show.legend = FALSE) +
    geom_smooth(method = "lm", se = FALSE) +
    geom_smooth(method = "lm", se = FALSE, linetype = 2, alpha = 0.6, colour = "black") +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
         caption = "Regression assumptions are unvalidated") +
    geom_hline(aes(yintercept = 79.8), size = 0.5, colour = "black", linetype = 3) +
    scale_colour_manual(values = c("#5C1AAE", "#AE5C1A", "#1AAE5C"),
        labels = c("High", "Medium", "Low")) +
    scale_shape_manual(values = c(15,17,19),
        labels = c("High", "Medium", "Low")) +
    theme_rbook()
```


Regression assumptions are unvalidated
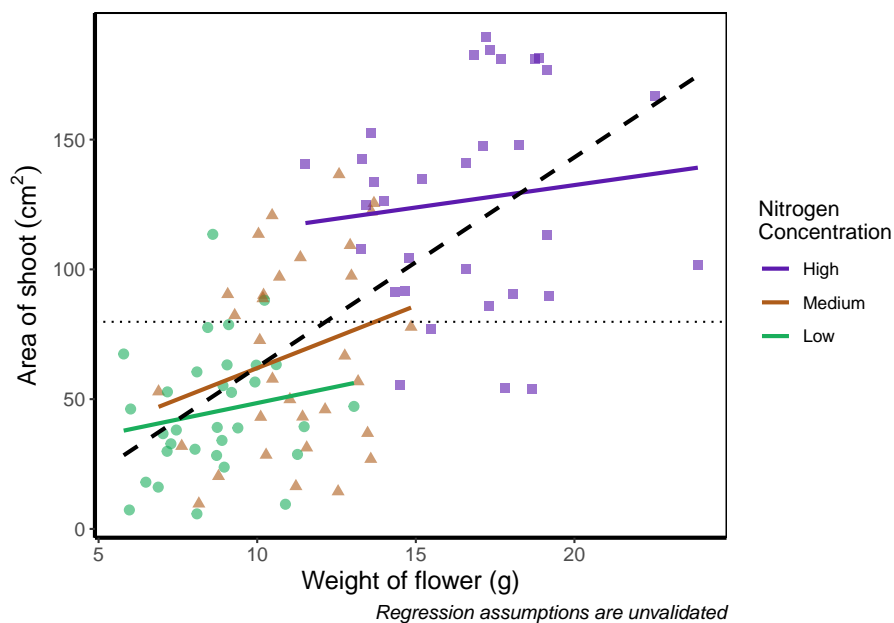
5.3.9 Writing on a figure

What if we were so utterly proud of our figure that we wanted to sign it (just like a painter signs their works of art)? We can do this using `geom_text()`. If we want to change the font, we need to check which ones are available for us to use. We can check this by running `windowsFonts()`. We'll use Times New Roman in this example, which is referred to as `serif`.

Not happy with your font options? Check out the `extrafont` package which expands your 'fontage'.


```
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    geom_smooth(method = "lm", se = FALSE, linetype = 2, alpha = 0.6, colour = "black") +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
            caption = "Regression assumptions are unvalidated") +
    geom_hline(aes(yintercept = 79.8), size = 0.5, colour = "black", linetype = 3) +
    scale_colour_manual(values = c("#5C1AAE", "#AE5C1A", "#1AAE5C"),
        labels = c("High", "Medium", "Low")) +
    scale_shape_manual(values = c(15,17,19),
        labels = c("High", "Medium", "Low")) +
    # Including layer to display text
    geom_text(x = 22, y = 5, label = "A. Artist", colour = "grey", family = "serif") +
    theme_rbook()
```

Regression assumptions are unvalidated
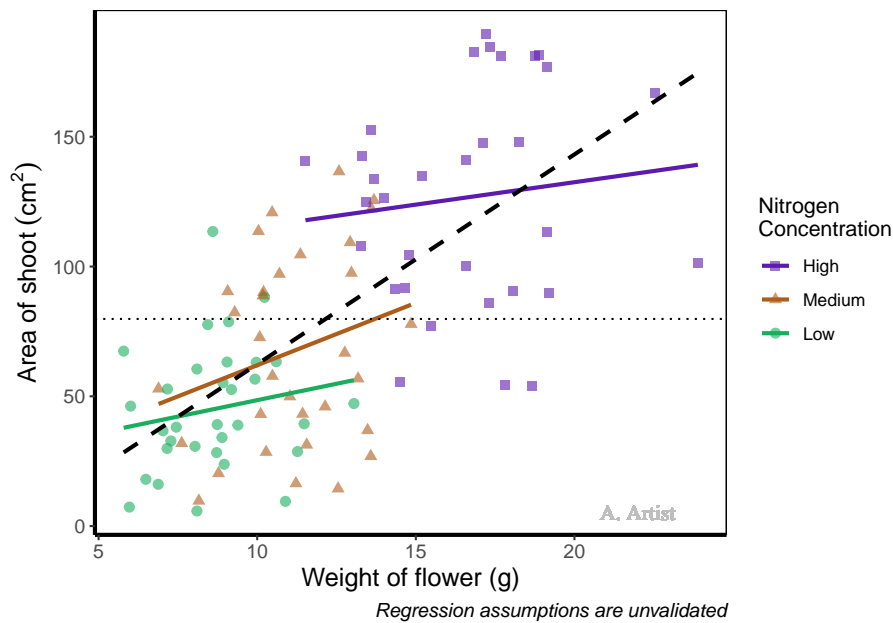
In reality there are more appropriate uses for `geom_text()` but whatever the reason, the mechanics remain the same. We specify what the x and y position are (on the scale of the axes), what we want written (using `label =`), the font (using `family =`). If you want to do more complex tasks with text and labels, check out `ggrepel` which extends the options available to you.

Similarly, if we want to include a tag for the figure, for instance we may want to refer to this figure as A we can do this using an additional argument in `labs()`. Let's see how that works:

```r
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    geom_smooth(method = "lm", se = FALSE, linetype = 2, alpha = 0.6, colour = "black") +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    # Including tag argument
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
            caption = "Regression assumptions are unvalidated", tag = "A") +
    geom_hline(aes(yintercept = 79.8), size = 0.5, colour = "black", linetype = 3) +
    scale_colour_manual(values = c("#5C1AAE", "#AE5C1A", "#1AAE5C"),
        labels = c("High", "Medium", "Low")) +
    scale_shape_manual(values = c(15,17,19),
        labels = c("High", "Medium", "Low")) +
    theme_rbook()
```
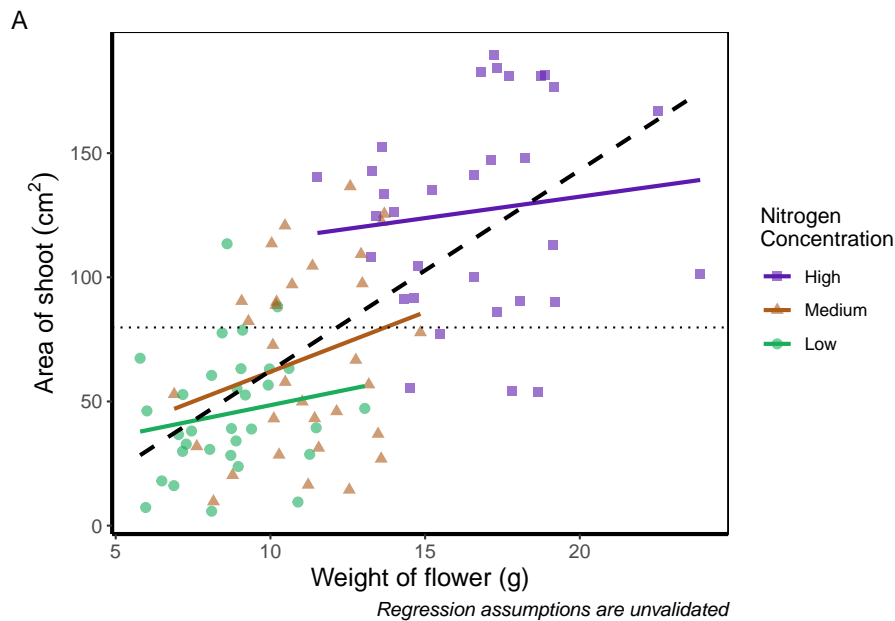
Doing so we get an "A" at the top left of the figure. If we weren't happy with the position of the "A", we can always use `geom_text()` instead and position it ourselves.

5.3.10   Axes tick marks and tick labels

What are we to do if we want more or fewer ticks on an axis? We can do this using the appropriate layers; `scale_y_discrete()` and `scale_x_discrete()` for discrete data (e.g. factors); and `scale_y_continuous()` and `scale_x_continuous()` for continuous data. Within these layers, the argument we want to use is called `breaks =`, though we need to use this in combination with `seq()` (see Chapter 2 for a reminder on how the `seq()` function works). We'll alter the x axis ticks in this example, with a tick label every 2.5 units.


```
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    geom_smooth(method = "lm", se = FALSE, linetype = 2, alpha = 0.6, colour = "black") +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
            caption = "Regression assumptions are unvalidated") +
    geom_hline(aes(yintercept = 79.8), size = 0.5, colour = "black", linetype = 3) +
```

```
    scale_colour_manual(values = c("#5C1AAE", "#AE5C1A", "#1AAE5C"),
        labels = c("High", "Medium", "Low")) +
    scale_shape_manual(values = c(15,17,19),
        labels = c("High", "Medium", "Low")) +
    # Adjusting breaks on x axis
    scale_x_continuous(breaks = seq(from = 5, to = 25, by = 2.5)) +
    theme_rbook()
```


Regression assumptions are unvalidated
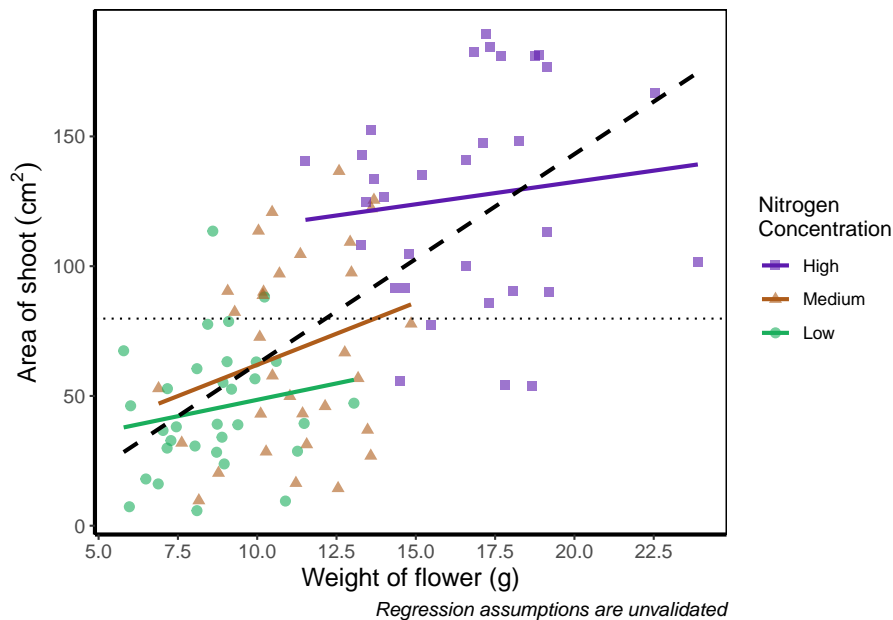
Axis tick labels sometimes need to be rotated. If you've ever worked with data from multiple species (with those lovely long latin names) for example, you'll know that it can be a nightmare making figures. The names can end up overlapping to such an extent that your axis tick labels merge into a giant black blob of unreadable abstractionism. In such cases it's best to rotate the text to make it readable. Doing so isn't too much of a pain and we'll be using `theme()` again to set the text angle to 45 degrees in addition to a little vertical adjustment so that the text doesn't get too close or run too far away from the axis.


```
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    geom_smooth(method = "lm", se = FALSE, linetype = 2, alpha = 0.6, colour = "black") +
```

```r
xlab("Weight of flower (g)") +
ylab(bquote("Area of shoot"~(cm^2))) +
labs(shape = "Nitrogen\nConcentration", colour = "Nitrogen\nConcentration",
        caption = "Regression assumptions are unvalidated") +
geom_hline(aes(yintercept = 79.8), size = 0.5, colour = "black", linetype = 3) +
scale_colour_manual(values = c("#5C1AAE", "#AE5C1A", "#1AAE5C"),
    labels = c("High", "Medium", "Low")) +
scale_shape_manual(values = c(15,17,19),
    labels = c("High", "Medium", "Low")) +
# Changing the angle of the axis text
theme(axis.text.x=element_text(angle = 45, vjust = 0.5)) +
theme_rbook()
```


Regression assumptions are unvalidated
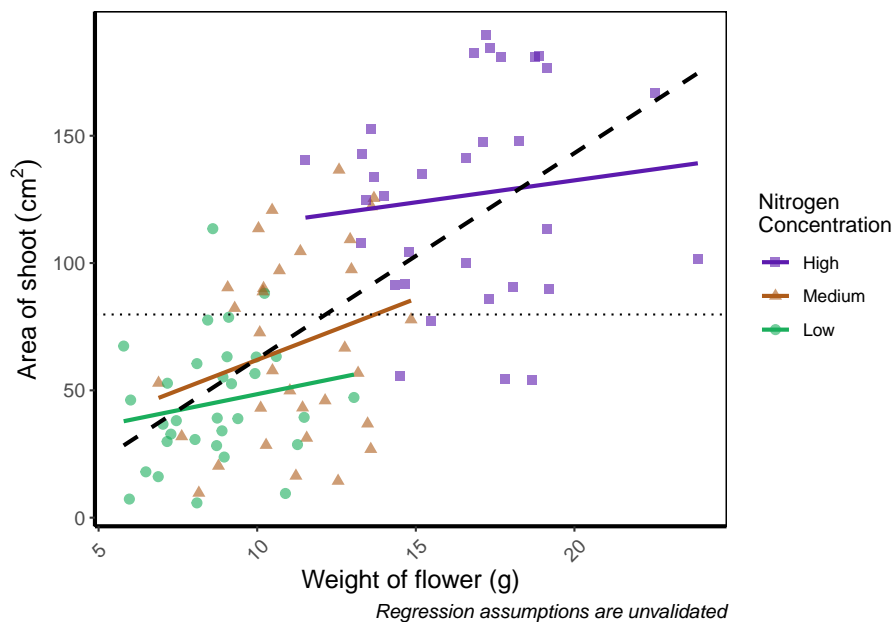
Phew. Now we can really read those numbers. Granted, rotating the axis text here isn't needed, but keep this trick in mind if you have lots of levels in a factor which leads to illegible blobs.

5.3.11   More advanced patchwork

When we used `patchwork` previously, we used the basics of the package to create nested figures. There are more advanced features which we'll quickly go through here.

As a precursor, we'll generate four simple scatterplots each with one of the four different themes used earlier and assign the nested figure the name `nested_1`. We'll use the `patchwork` operator + to link them all together and see how that looks.

```
a <- ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    theme_classic()


b <- ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    theme_bw()


c <- ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    theme_minimal()


d <- ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    theme_light()


nested_1 <- a + b + c +d


nested_1
```
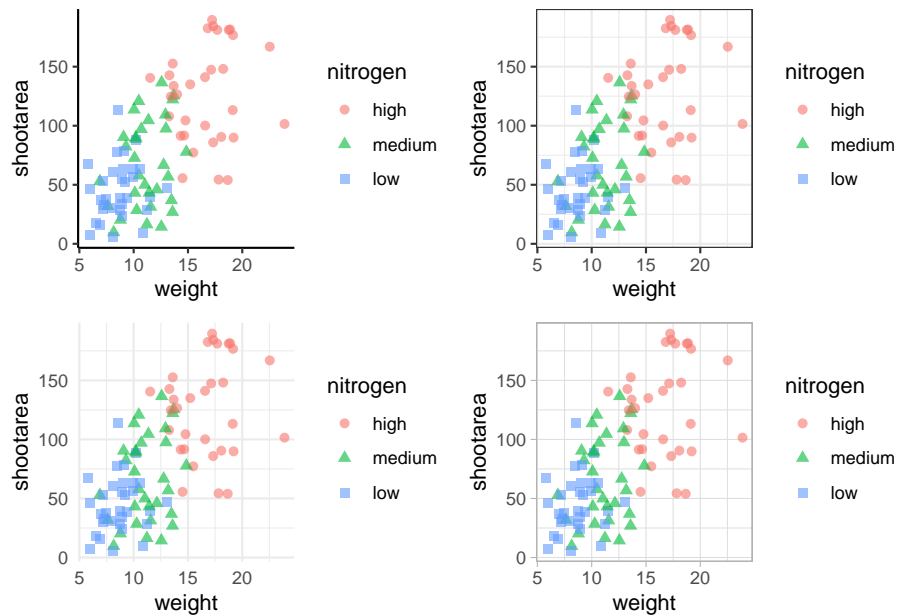

The problem with this figure is that we're using a lot of space to get legends squeezed into the middle. We can solve that by using the `plot_layout()` function and collecting the guides together:

```
nested_1 +
    plot_layout(guides = "collect")
```
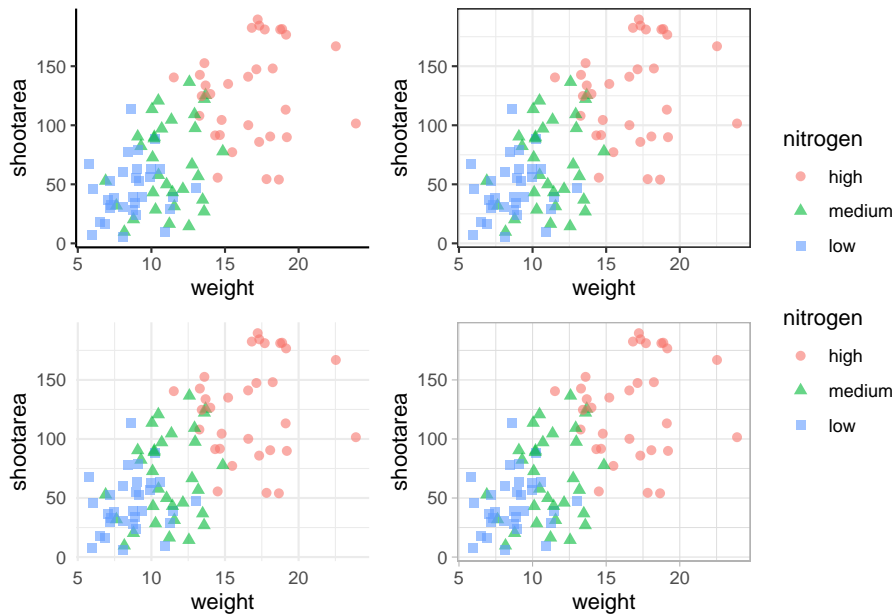

Looks a bit tidier, though in this case say we want to remove the legends entirely. Note that to do this we use a new operator, the ampersand (`&`). When we use the `&` operator in `patchwork`, we add `ggplot2` elements to all plots within patchwork. To show how you can use `patchwork` and `ggplot2` together, we'll also add in an additional `geom_path()` to all plots, while simultaneously removing the legends.

```
# note the use of '&' instead of '+'
nested_1 &
    theme(legend.position = 'none') &
    geom_path()
```
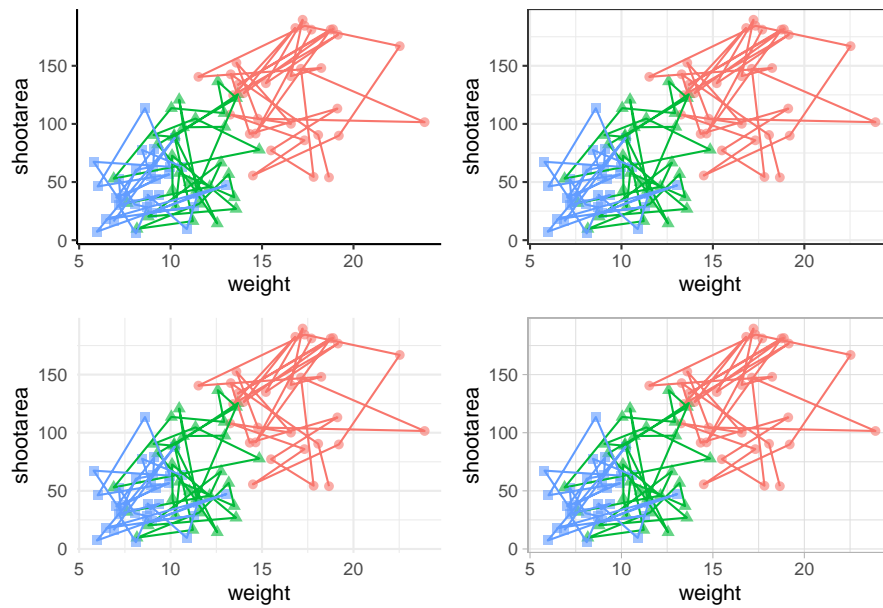
To change the layout of the figure we can use the `plot_layout()` function to specify a design. We'll start by creating a layout to use, which we call `grid_layout` using A, B, C, D to represent plots `a`, `b`, `c` and `d` (we could have called the plots anything we wanted to). We'll arrange it so that the four graphs surround an empty white space in the centre of the plotting space. Each plot will either be five units wide or five unites tall.

```
grid_layout <- "
AAAAA
B###C
B###C
B###C
B###C
B###C
DDDDD
"

nested_1 +
    plot_layout(design = grid_layout) &
    theme(legend.position = 'none')
```
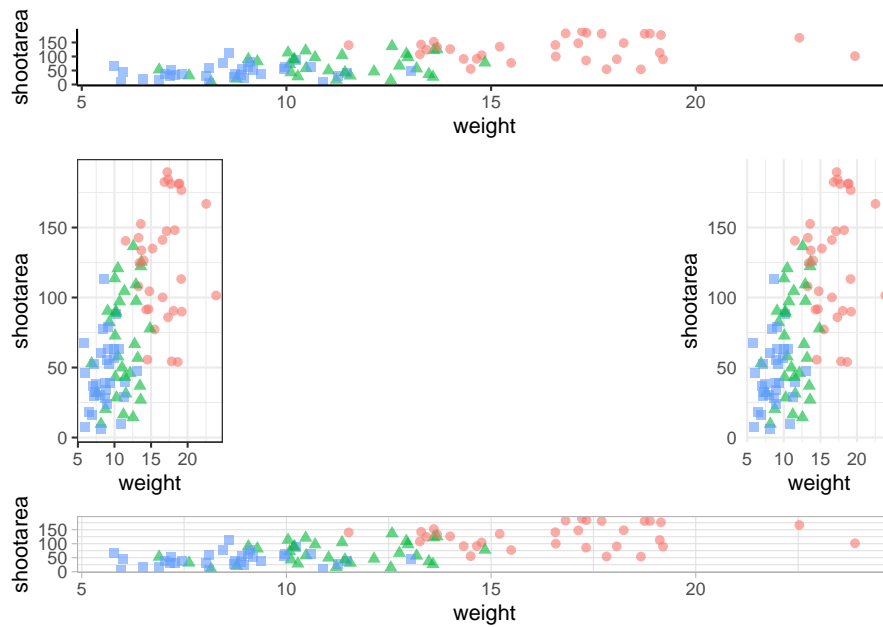
To round off this Chapter, we'll take you on a whistle-stop tour of some of the common types of plots you can make in the aptly named "ggplot bestiary section".

5.4   A ggplot bestiary

What follows is a quick run through of example `ggplots`. These will predominantly be created by changing the geoms used, but there will be additional tweaks which we'll highlight.

5.4.1   Density plot

Below is a density plot which is much like a histogram. The x axis shows observations of given numbers of flowers, while the y axis is the density of observations (roughly equivalent to number of rows with that many flowers, calculated in the background by the statistics layer). Each density is coloured according to nitrogen concentration, though note that we're using `fill =` instead of `colour =`. Try using colour instead to see what happens.

Notice that we haven't used `data = flower` here and instead just used `flower`? When an object is not assigned with an argument, `ggplot2` will assume that it is the dataset. We're using that here, but we actually prefer to explicitly state the argument name in our own work.

```
ggplot(flower) +
    geom_density(aes(x = flowers, fill = nitrogen), alpha = 0.5) +
    labs(y = "Density", x = "Number of Flowers", fill = "Nitrogen\nConcentration") +
    scale_fill_manual(labels = c("High", "Medium", "Low"),
        values = c("#DB24BC", "#BCDB24", "#24BCDB")) +
    theme_rbook()
```
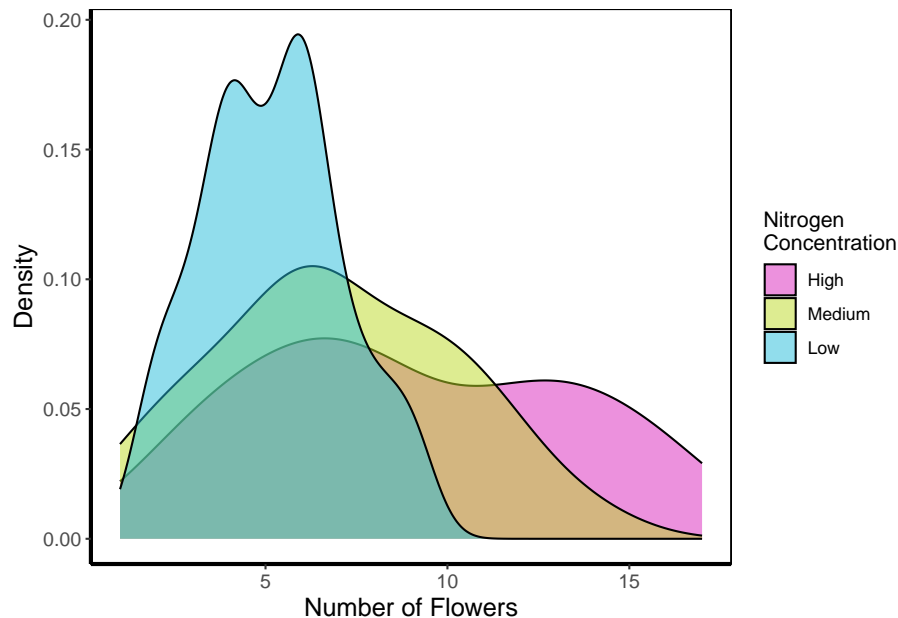

5.4.2 Histogram

Next is a histogram (a much more traditional version of a density plot). There are a couple of things to take note of here. The first is that `flower$block` is numeric and not a factor. We can correct that here fairly easily using the `factor()` function to convert it from numeric to factor (though ideally we'd have done this before - see Chapter 3). The other thing to take note of is that we've specified `bins = 20`. The number of bins control how many times the y-axis is broken up to show the data. Try increasing and decreasing to see the effect. The last is using the `position =` argument and stating that we do not want the bars to be stacked (the default position), instead we want them side-by-side, a.k.a. dodged.

```
ggplot(flower) +
    geom_histogram(aes(x = flowers, fill = factor(block)), colour = "black", bins = 20,
                position = "dodge") +
    labs(y = "Count", x = "Number of Flowers", fill = "Greenhouse\nBlock") +
    scale_fill_manual(labels = c("1", "2"),
```

```
        values = c("#8CD926", "#7326D9")) +
    theme_rbook()
```
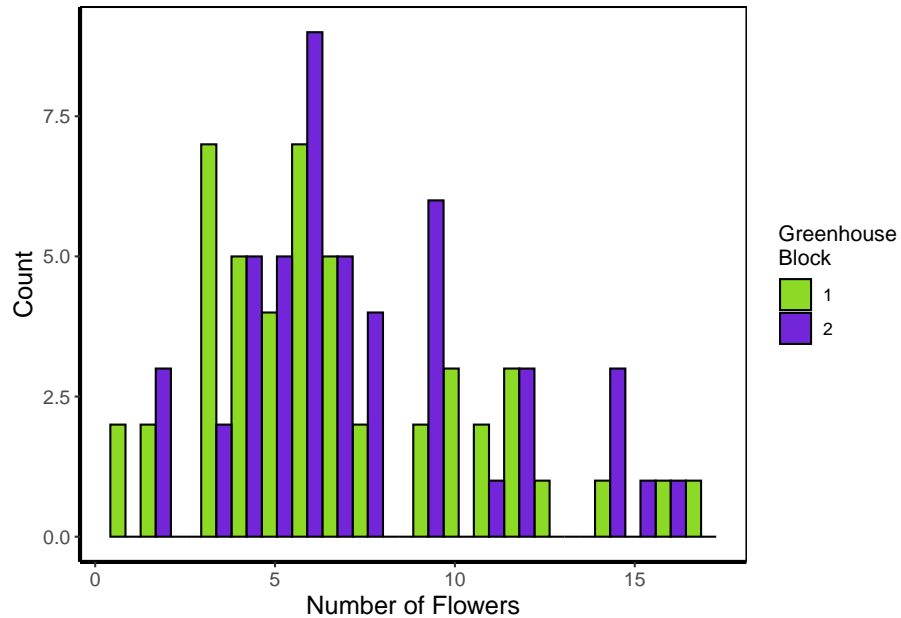

5.4.3 Frequency polygons

A frequency polygon is yet another visualisation of the above two. The only difference
here is that we are drawing a line to each value, instead of a density curve or bars.

```
ggplot(flower) +
    geom_freqpoly(aes(x = flowers, colour = factor(block)), size = 1, bins = 20) +
    labs(y = "Count", x = "Number of Flowers", colour = "Greenhouse\nBlock") +
    scale_colour_manual(labels = c("1", "2"),
        values = c("#8CD926", "#7326D9")) +
    theme_rbook()
```
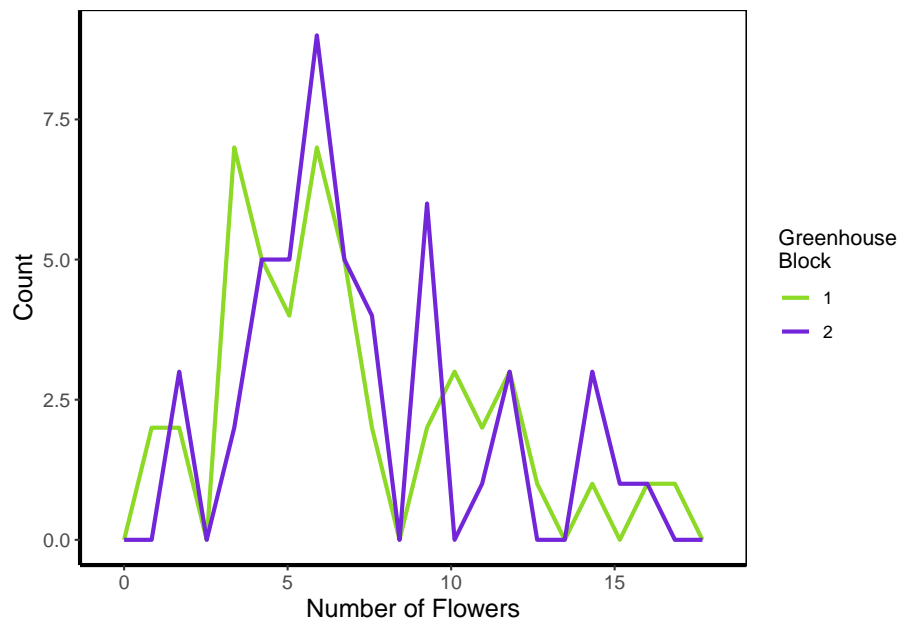
5.4.4 Boxplot

Boxplots are a classic way to show to spread of data, and they're easy to make in `ggplot2`. The dark line in the middle of the box shows the median, the boxes show the 25th and 75th percentiles (which is different from the base R `boxplot()`), and the whiskers show 1.5 times the inter-quartile range (i.e. the distance between between the lower and upper quartiles).

```
ggplot(flower) +
    geom_boxplot(aes(y = flowers, x = nitrogen)) +
    labs(y = "Number of Flowers", x = "Nitrogen Concentration") +
    theme_rbook()
```
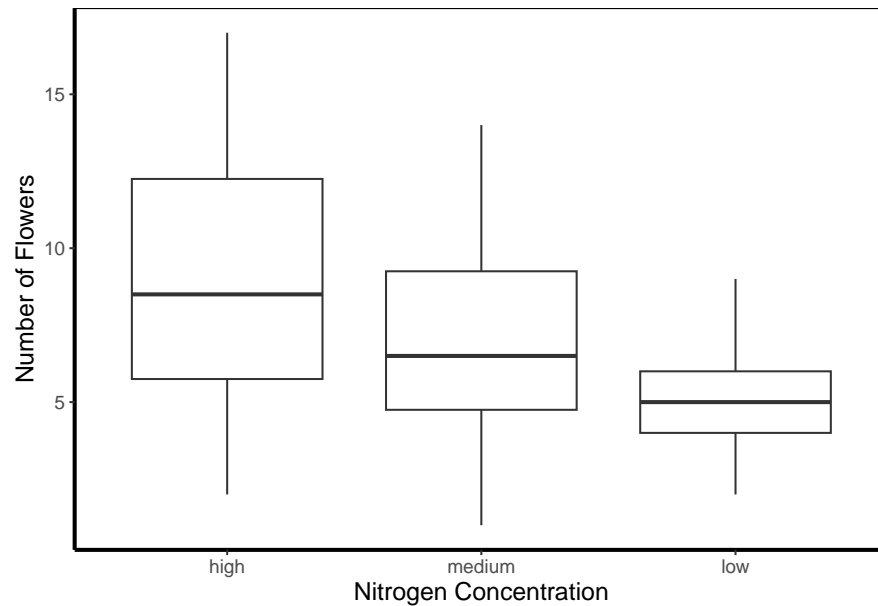
5.4.5   Violin plots

Violin plots are an increasingly popular alternative to boxplots. They display much of the same information, as well as showing a version of the density plot above (imagine each violin plot, cut in half vertically and place on it's side, thus showing the overall distribution of the data). In the plot below the figure is slightly more complex than those above and so deserves some explanation.

Within `geom_violin()` we've included `draw_quantiles =` where we've specified we want quantile lines drawn at the 25, 50 and 75 quantiles (using the `c()` function). In combination with `geom_violin()` we've also included `geom_jitter()`. `geom_jitter()` is similar to `geom_point()` but induces a slight random spread of the points, often helpful when those points would otherwise be clustered. Within `geom_jitter()` we've also set `height = 0`, and `width = 0.1` which specifies how much to jitter the points in a given dimension (here essentially telling `ggplot2` not to jitter by height, and only to jitter width by a small amount).

Finally, we're also using this plot to show `scale_y_log10`. Hopefully this is largely self-explanatory (it converts the y-axis to the $\log_{10}$ scale). There are additional scaling options for axis (for instance `scale_y_sqrt()`). Please note that using a log scaled axis in this case is actually doing harm in terms of understanding the data, we'd actually be much better off not doing so in this particular case.

```
ggplot(flower) +
  geom_violin(aes(y = flowers, x = nitrogen, fill = nitrogen),
              draw_quantiles = c(0.25, 0.5, 0.75)) +
  geom_jitter(aes(y = flowers, x = nitrogen), colour = "black", height = 0,
              width = 0.1, alpha = 0.5) +
```

```
    scale_fill_manual(labels = c("High", "Medium", "Low"),
        values = c("#5f7f5c", "#749770", "#9eb69b")) +
    labs(y = "Number of Flowers", x = "Nitrogen Concentration") +
    scale_y_log10() +
    theme_rbook()
```
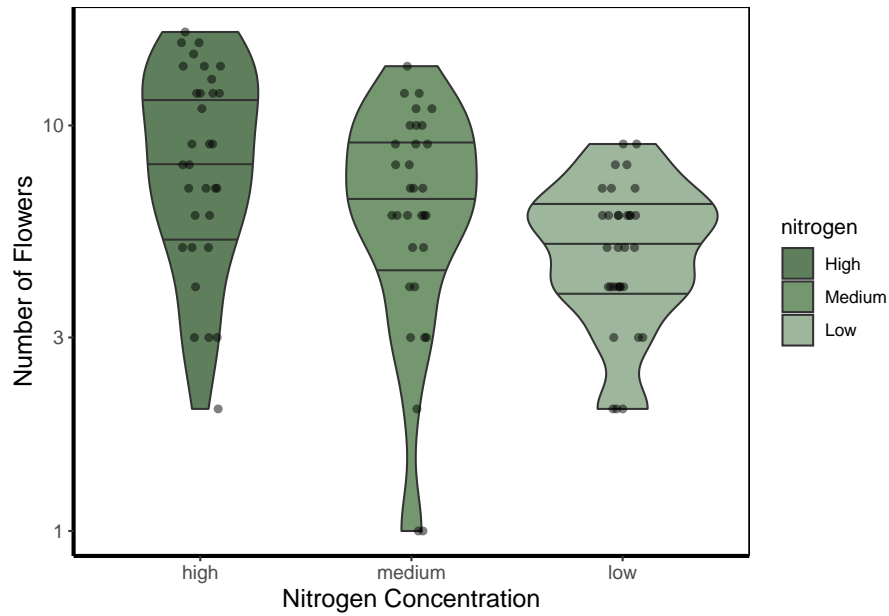

5.4.6 Barchart

Below is an example of barcharts. It is included here for completeness, but be aware that they are viewed with contention (with good reason). Briefly, barcharts can hide information, or imply there is data where there is none; ultimately misleading the reader. There are almost always better alternatives to use that better demonstrate the data.

```
ggplot(flower) +
  geom_bar(aes(x = factor(block), fill = nitrogen)) +
    scale_fill_manual(labels = c("High", "Medium", "Low"),
        values = c("#2613EC", "#9313EC", "#EC13D9")) +
    labs(y = "Count", x = "Block") +
    theme_rbook()
```
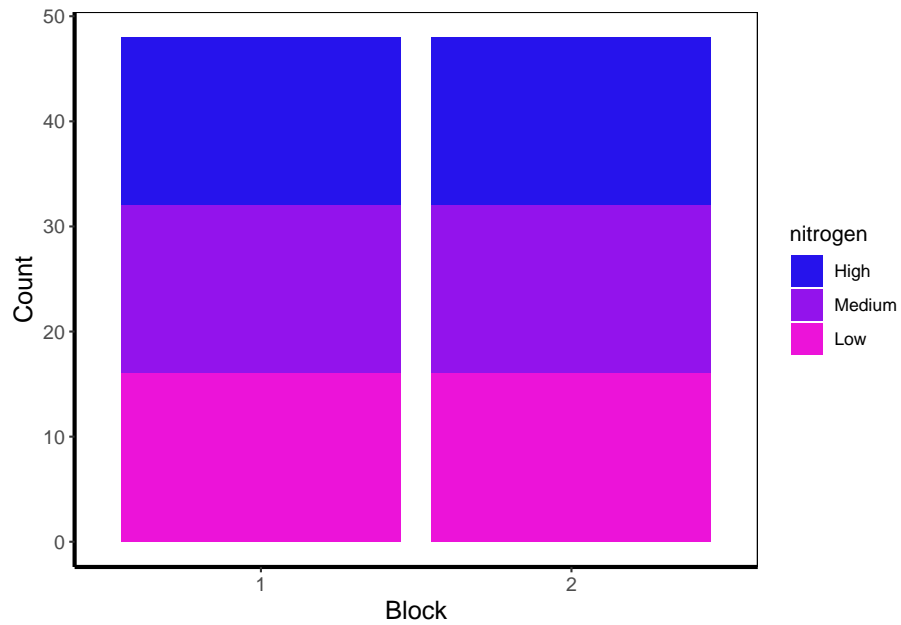
The barchart shows the numbers of observations in each block, with each bar split according to the number of observations in each nitrogen concentration. In this case they are equal because the dataset (and experimental design) was balanced.

5.4.7   Quantile lines

While we can draw a straight line, perhaps we would also like to include the descriptive nature of a boxplot, except using continuous data. We can use quantile lines in such cases. Note that for quantiles to be calculated `ggplot2` requires the installation of the package `quantreg`.

```r
library(quantreg)

ggplot(aes(x = weight, y = shootarea), data = flower) +
    geom_point(size = 0.5, alpha = 0.6) +
    geom_quantile(colour = "darkgrey", size = 1) +
    labs(y = "Shoot Area", x = "Weight") +
    theme_rbook()
```
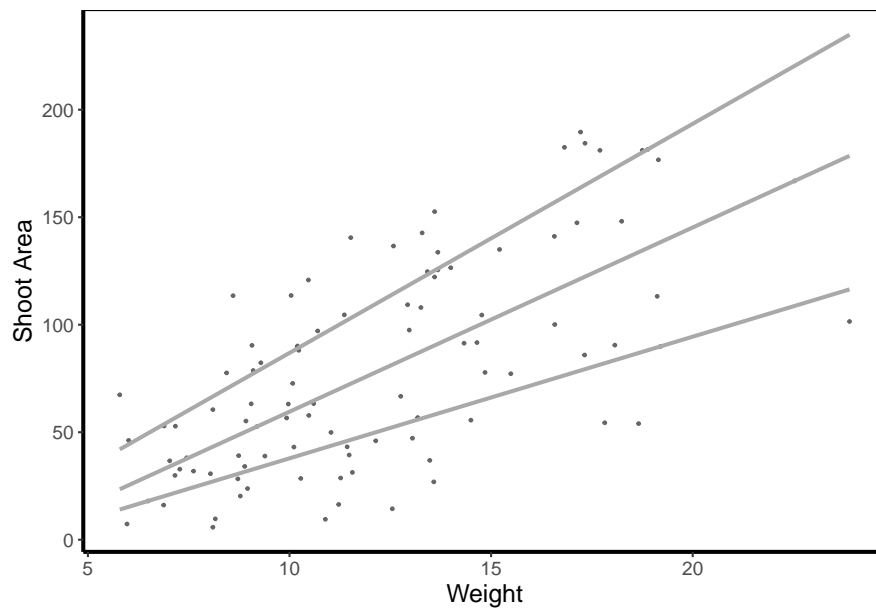
5.4.8 Heatmap

Heatmaps are a great tool to visualise spatial patterns. `ggplot2` can easily handle such data using `geom_bin2d()` to allow us to see if our data is more (or less) clustered.

```
ggplot(aes(x = leafarea, y = shootarea), data = flower) +
    geom_bin2d() +
    labs(y = "Shoot Area", x = "Leaf Area") +
    coord_cartesian(xlim = c(5,25)) +
    theme_rbook()
```
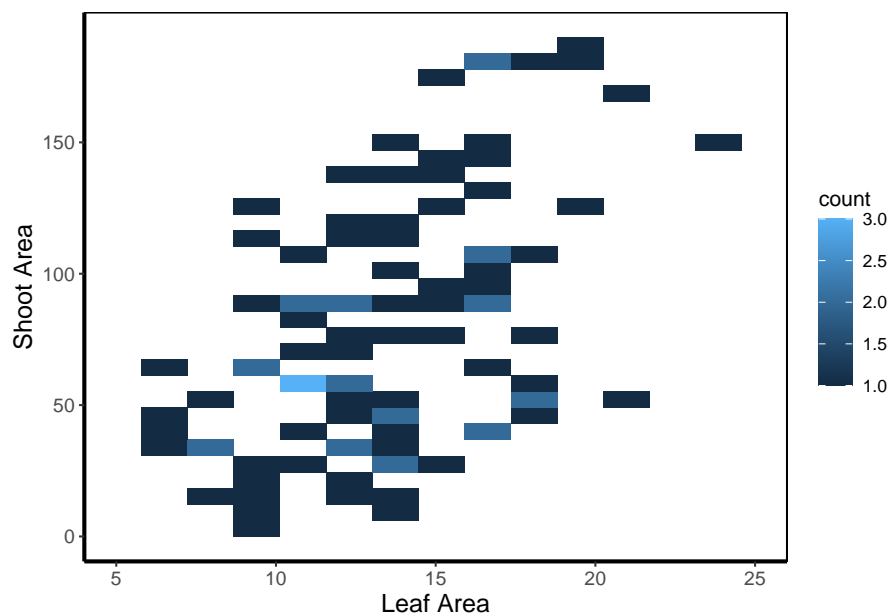
In this example, lighter blue squares show combinations of leaf area and shoot area where we have more data, and dark blue shows the converse.

5.4.9 Hex map

A similar version to `geom_density2d()` is `geom_hex()`. The only difference between the two is that the squares are replaced with hexagons. Note that `geom_hex()` requires you to first install an additional package called `hexbin`.

```r
library(hexbin)

ggplot(aes(x = leafarea, y = shootarea), data = flower) +
    geom_hex() +
    labs(y = "Shoot Area", x = "Leaf Area") +
    coord_cartesian(xlim = c(5,25)) +
    theme_rbook()
```
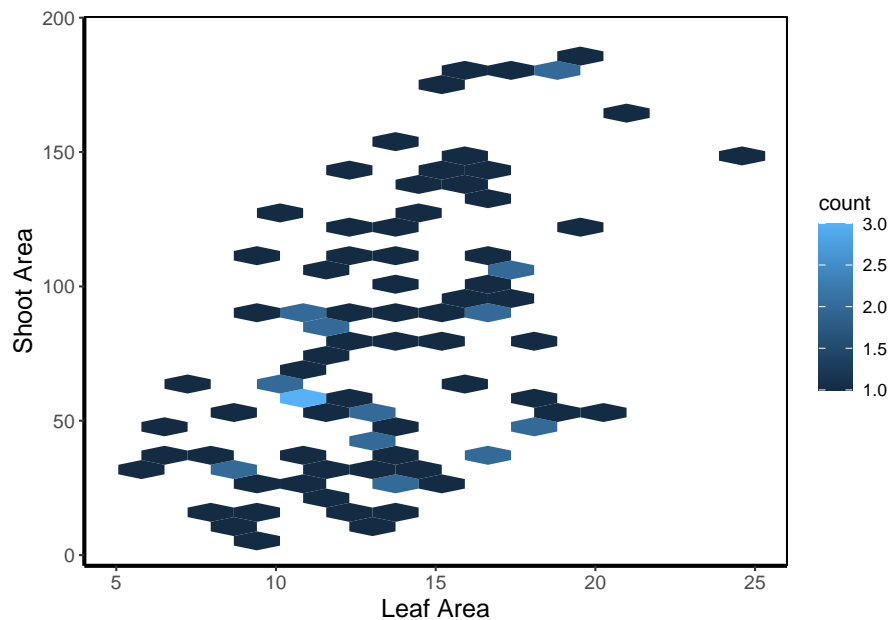

5.4.10 Contour map

Similar to a heatmap we can make a contour map using `geom_density_2d()`. The way to read this figure is much the same way as you'd read a topographical map showing mountains or peaks. The central polygon represents the space (amongst shoot and leaf area) where there are most observations. As you "step" down this mountain to the next

line, we step down in the number of counts. Think of this as showing where points are most clustered, as in `geom_bin2d()`.

```
ggplot(aes(x = leafarea, y = shootarea), data = flower) +
    geom_density2d() +
    labs(y = "Shoot Area", x = "Leaf Area") +
    coord_cartesian(xlim = c(5,25)) +
    theme_rbook()
```
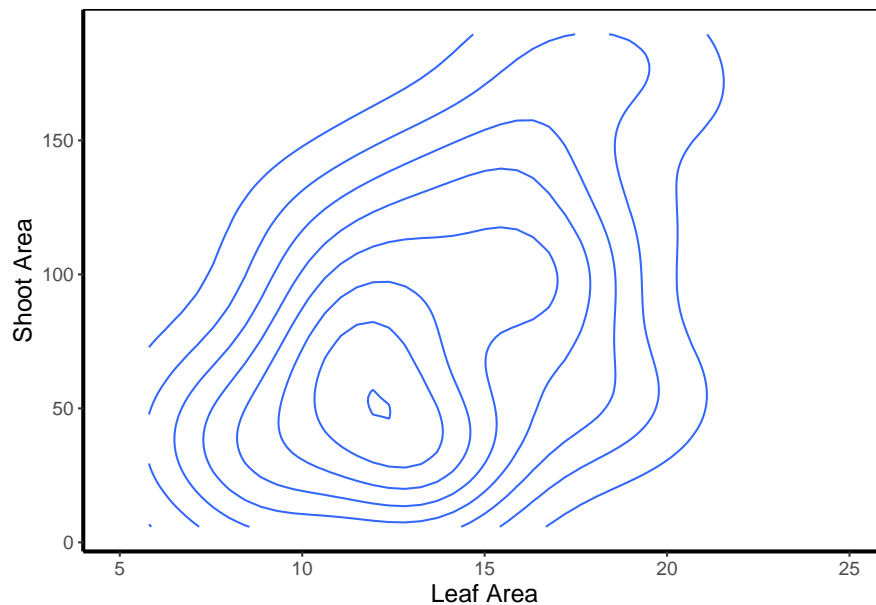

We can then expand on this using the statistics layer, via `stat_nameOfStatistic`. For instance, we can use the calculated "level" (representing the height of contour) to fill in our figure. To do so, we'll swap out `geom_density_2d()` for `stat_density_2d()` which will allow us to colour in the contour map.

```
ggplot(aes(x = leafarea, y = shootarea), data = flower) +
    stat_density_2d(aes(fill = stat(level)), geom = "polygon") +
    labs(y = "Shoot Area", x = "Leaf Area") +
    coord_cartesian(xlim = c(5,25)) +
    theme_rbook()
```

```
## Warning: `stat(level)` was deprecated in ggplot2 3.4.0.
## i Please use `after_stat(level)` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated.
```
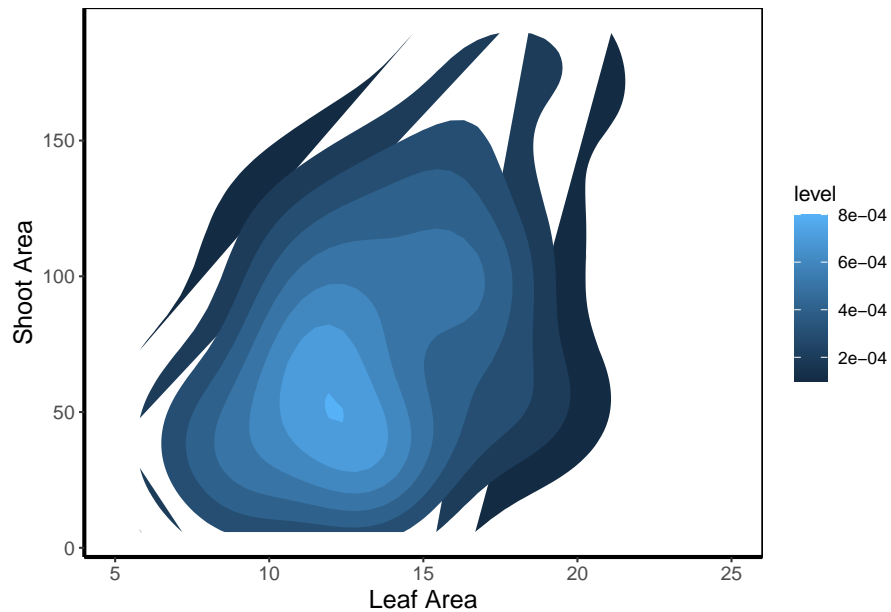
In this case it doesn't look too pretty given that we don't have a wide spread of data leading to partly formed polygons.

5.4.11   Cleveland dotplot

The cleveland dotplot is a figure well suited to detect unusual observations (outliers?) in our data. The y axis is simply the order in which the data appears in the dataset. The x axis is our variable of interest. Points far to the left or far to the right are indicative of outliers. `ggplot2` does not have a geom specifically designed for a cleveland dotplot, though because of the grammar of graphics framework, we can build it ourselves.

We do this by first using the function `rownames()`. This function returns the row name of each row. In most data frames, the row names are simply numbers. Once we have this information, we can convert this to numbers using `as.numeric()`. After doing this we have a vector of numbers from 1 to 96. Keep in mind that such techniques can be used for any sort of figure we want to make. We'll use the same technique to add horizontal lines for each row of data to replicate the traditional look of a dotchart.

```
ggplot(flower) +
   geom_hline(aes(yintercept = as.numeric(rownames(flower))), linetype = 2,
            colour = "lightgrey") +
 geom_point(aes(x = leafarea, y = as.numeric(rownames(flower)))) +
 labs(y = "Order in Dataset", x = bquote("Area of leaf"~(cm^2))) +
   theme_rbook()
```
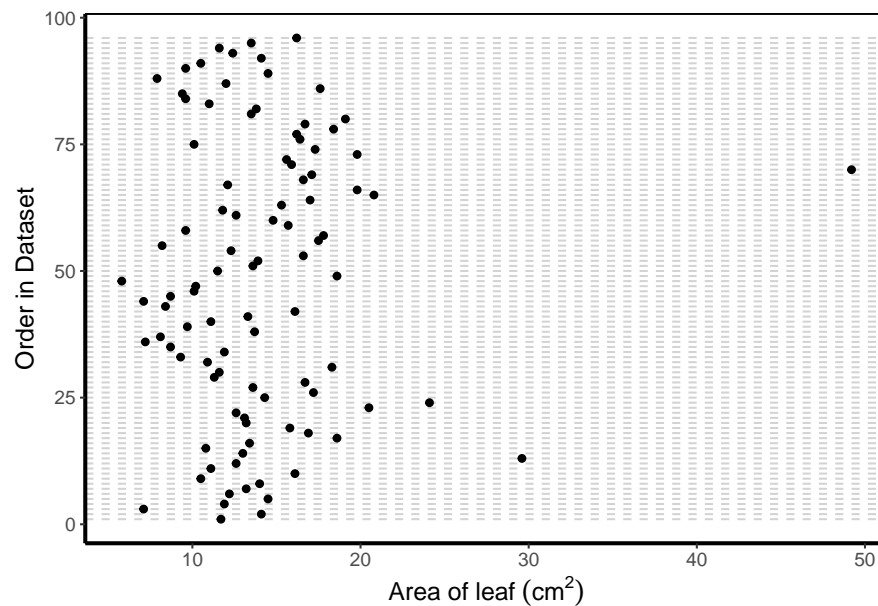
5.4.12 Pairs plot

For the final figure we will need an additional package, called `GGally`. Within `GGally` is a function called `ggpairs()` which is equivalent to `pairs()` (part of base R). Both functions produce a scatterplot matrix of all variables of interest. Running diagonally down from left to right are plots showing distributions; either as stacked barcharts or as density plots, in this case coloured according to nitrogen concentration. The bottom plots show the plotted relationships, with variables on the top as the x axis, and the variables on the right plotted as the y axis. For example, the very bottom left plots shows the relationship between number of flowers on the y axis and treatment on the x axis, coloured according to nitrogen concentration. Conversely, the upper plots show either the correlation between variables, when both variables are continuous and the correlation can be calculated, boxplots when the plot is of a factor variable and a continuous variable, or as a stacked barchart when both variables are factors.

Such figures are a fantastic way to quickly show relationships in the dataset. They can of course be used to plot only part of the dataset, for instance using `flowers[ ,1:5]` to plot the first five columns of the dataset.

We've included two additional arguments in the code below. `cardinality_threshold =` forces `GGally` to plot all of the variables in our dataset (where it normally wants fewer variables) as well as `progress =` to avoid a progress bar appearing when we run the code.

```r
library(GGally)
flower$block <- factor(flower$block)

ggpairs(flower, aes(colour = nitrogen), cardinality_threshold = NULL, progress = FALSE) +
    theme_rbook()
```
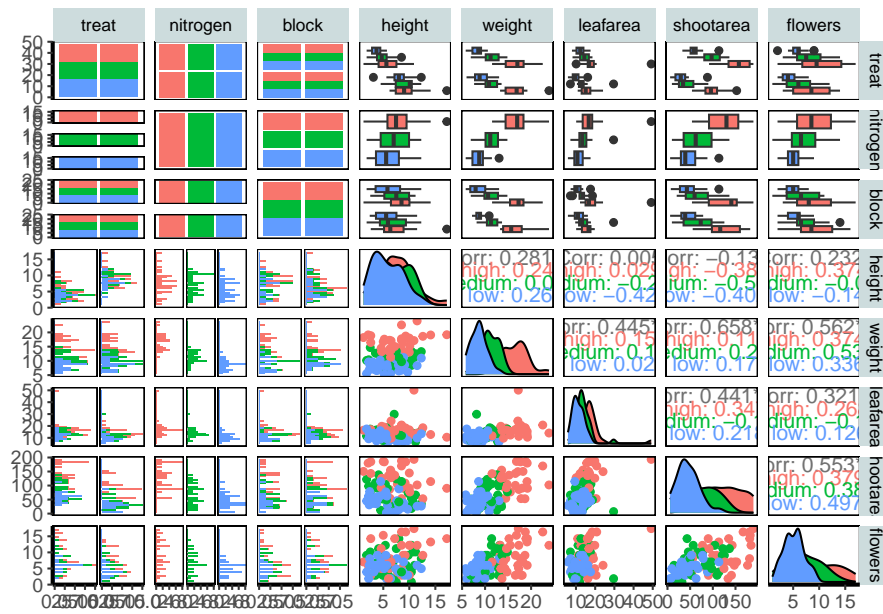
5.5   Exercise 5

Congratulations, you've reached the end of Chapter 5! Perhaps now's a good time to practice some of what you've learned. You can find an exercise we've prepared for you (and our solutions) on the course website.

Chapter 6

Simple Statistics in R

> Note: The content of this Chapter is a very old draft which we've included here as a placeholder. Although you may still find some of the the content useful we are in the process of substantially rewriting this Chapter.

In addition to R's powerful data manipulation and graphics facilities, R includes a host of procedures which you can use to analyse your data. Many of these procedures are included with the base installation of R, however, even more can be installed with packages available from the CRAN website (see Chapter 1 for more details). All of the analyses described in this Chapter can be carried out without installing additional packages.

6.1 One and two sample tests

The two main functions for these types of tests are the `t.test()` and `wilcox.test()` that perform t tests and Wilcoxon's signed rank test respectively. Both of these tests can be applied to one and two sample analyses as well as paired data.

As an example of a one sample t test we will use the `trees` dataset which is included with R. To access this in-built dataset we can use the `data()` function. This data set provides measurements of the diameter, height and volume of timber in 31 felled black cherry trees (see `?trees` for more detail).

```
data(trees)
str(trees)
## 'data.frame':    31 obs. of  3 variables:
##  $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
##  $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
##  $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
summary(trees)
```

```
##       Girth              Height        Volume
##   Min.   : 8.30   Min.    :63   Min.    :10.20
##   1st Qu.:11.05   1st Qu.:72   1st Qu.:19.40
##   Median :12.90   Median :76   Median :24.20
##   Mean    :13.25   Mean    :76   Mean    :30.17
##   3rd Qu.:15.25   3rd Qu.:80   3rd Qu.:37.30
##   Max.    :20.60   Max.    :87   Max.    :77.00
```

If we wanted to test whether the mean height of black cherry trees in this sample is
equal to 70 ft (`mu = 70`), assuming these data are normally distributed, we can use a
`t.test()` to do so.

```
t.test(trees$Height, mu = 70)
##
##  One Sample t-test
##
## data:  trees$Height
## t = 5.2429, df = 30, p-value = 1.173e-05
## alternative hypothesis: true mean is not equal to 70
## 95 percent confidence interval:
##   73.6628 78.3372
## sample estimates:
## mean of x
##        76
```

The above summary has a fairly logical layout and includes the name of the test that
we have asked for (`One Sample t-test`), which data has been used (`data:   trees$Height`),
the t statistic, degrees of freedom and associated p value (`t = 5.2`, `df = 30`, `p-value =`
`1e-05`). It also states the alternative hypothesis (`alternative hypothesis: true mean is`
`not equal to 70`) which tells us this is a two sided test (as we have both equal and not
equal to), the 95% confidence interval for the mean (`95 percent confidence interval:73.66`
`78.34`) and also an estimate of the mean (`sample estimates: mean of x : 76`). In the above
example, the p value is very small and therefore we would reject the null hypothesis
and conclude that the mean height of our sample of black cherry trees is not equal to
70 ft.

The function `t.test()` also has a number of additional arguments which can be used
for one-sample tests. You can specify that a one sided test is required by using either
`alternative = "greater"` or `alternative = "less` arguments which tests whether the sample
mean is greater or less than the mean specified. For example, to test whether our sample
mean is greater than 70 ft.

```
t.test(trees$Height, mu = 70, alternative = "greater")
##
##   One Sample t-test
##
## data:  trees$Height
## t = 5.2429, df = 30, p-value = 5.866e-06
## alternative hypothesis: true mean is greater than 70
## 95 percent confidence interval:
##   74.05764      Inf
## sample estimates:
## mean of x
##        76
```

You can also change the confidence level used for estimating the confidence intervals using the argument `conf.level = 0.99`. If specified in this way, 99% confidence intervals would be estimated.

Although t tests are fairly robust against small departures from normality you may wish to use a rank based method such as the Wilcoxon's signed rank test. In R, this is done in almost exactly the same way as the t test but using the `wilcox.test()` function.

```
wilcox.test(trees$Height, mu = 70)
## Warning in wilcox.test.default(trees$Height, mu = 70): cannot compute exact
## p-value with ties
## Warning in wilcox.test.default(trees$Height, mu = 70): cannot compute exact
## p-value with zeroes
##
##   Wilcoxon signed rank test with continuity correction
##
## data:  trees$Height
## V = 419.5, p-value = 0.0001229
## alternative hypothesis: true location is not equal to 70
```

Don't worry too much about the warning message, R is just letting you know that your sample contained a number of values which were the same and therefore it was not possible to calculate an exact p value. This is only really a problem with small sample sizes. You can also use the arguments `alternative = "greater"` and `alternative = "less"`.

In our one sample test it's always a good idea to examine your data for departures from normality, rather than just assuming everything is OK. Perhaps the simplest way to assess normality is the 'quantile-quantile plot'. This graph plots the ranked sample quantiles from your distribution against a similar number of ranked quantiles taken from a normal distribution. If your data are normally distributed then the plot of your data points will be in a straight line. Departures from normality will show up as a curve or s-shape in your data points. Judging just how much departure is acceptable comes with a little bit of practice.

To construct a Q-Q plot you need to use both the `qqnorm()` and `qqline()` functions. The `lty = 2` argument changes the line to a dashed line.

```
qqnorm(trees$Height)
qqline(trees$Height, lty = 2)
```

Normal Q–Q Plot
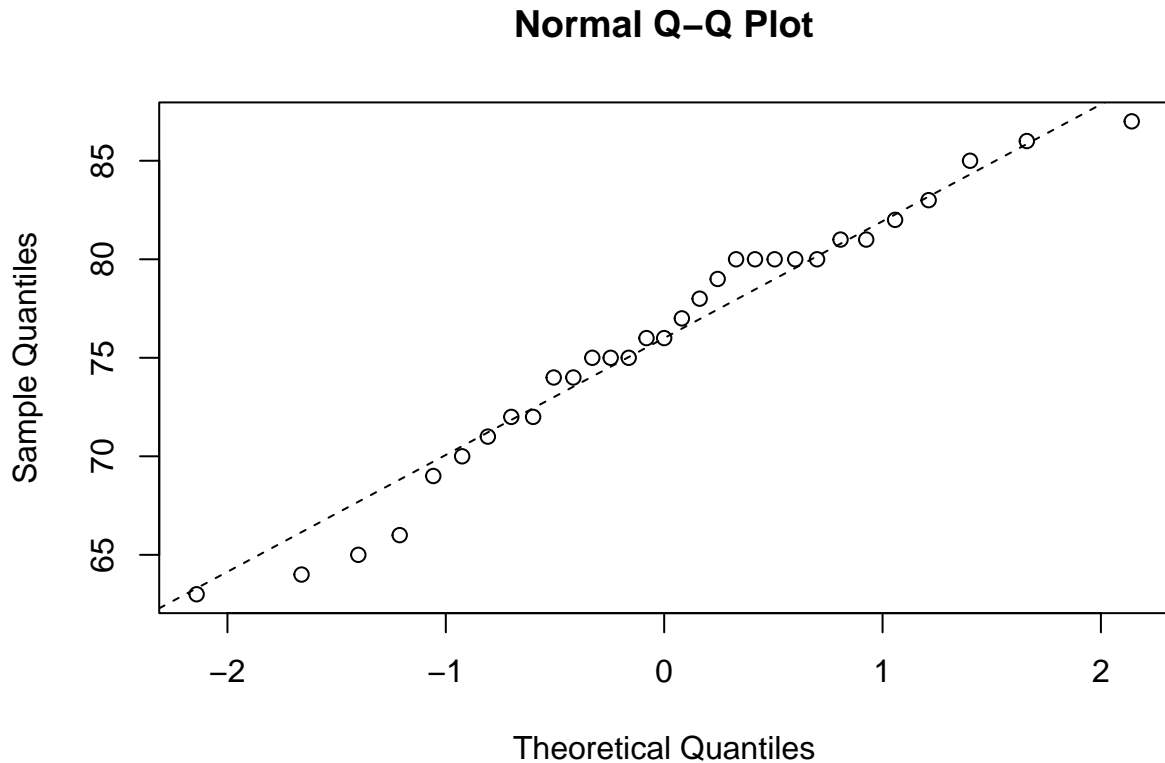
If you insist on performing a specific test for normality you can use the function `shapiro.test()` which performs a Shapiro – Wilks test of normality.

```
shapiro.test(trees$Height)
##
##  Shapiro-Wilk normality test
##
## data:  trees$Height
## W = 0.96545, p-value = 0.4034
```

In the example above, the p value $= 0.4$ which suggests that there is no evidence to reject the null hypothesis and we can therefore assume these data are normally distributed.

In addition to one-sample tests, both the `t.test()` and `wilcox.test()` functions can be used to test for differences between two samples. A two sample t test is used to test the null hypothesis that the two samples come from distributions with the same mean (i.e. the means are not different). For example, a study was conducted to test whether

'seeding' clouds with dimethylsulphate alters the moisture content of clouds. Ten random clouds were 'seeded' with a further ten 'unseeded'. The dataset can be found in the `atmosphere.txt` data file located in the `data/` directory on the Rbook github page.

```
atmos <- read.table('data/atmosphere.txt', header = TRUE)
str(atmos)
## 'data.frame':    20 obs. of  2 variables:
##  $ moisture : num  301 302 299 316 307 ...
##  $ treatment: chr  "seeded" "seeded" "seeded" "seeded" ...
```

As with our previous data frame (`flowers`), these data are in the long format. The column `moisture` contains the moisture content measured in each cloud and the column `treatment` identifies whether the cloud was `seeded` or `unseeded`. To perform a two-sample t test

```
t.test(atmos$moisture ~ atmos$treatment)
##
##  Welch Two Sample t-test
##
## data:  atmos$moisture by atmos$treatment
## t = 2.5404, df = 16.807, p-value = 0.02125
## alternative hypothesis: true difference in means between group seeded and group unseeded is not equal
## 95 percent confidence interval:
##    1.446433 15.693567
## sample estimates:
##    mean in group seeded mean in group unseeded
##                  303.63                 295.06
```

Notice the use of the formula method (`atmos$moisture ~ atmos$treatment`, which can be read as 'the moisture described by treatment') to specify the test. You can also use other methods depending on the format of the data frame. Use `?t.test` for further details. The details of the output are similar to the one-sample t test. The Welch's variant of the t test is used by default and does not assume that the variances of the two samples are equal. If you are sure the variances in the two samples are the same, you can specify this using the `var.equal = TRUE` argument

```
t.test(atmos$moisture ~ atmos$treatment, var.equal = TRUE)
##
##  Two Sample t-test
##
## data:  atmos$moisture by atmos$treatment
## t = 2.5404, df = 18, p-value = 0.02051
## alternative hypothesis: true difference in means between group seeded and group unseeded is not equal
```

```
## 95 percent confidence interval:
##    1.482679 15.657321
## sample estimates:
##    mean in group seeded mean in group unseeded
##                   303.63                 295.06
```

To test whether the assumption of equal variances is valid you can perform an *F*-test on the ratio of the group variances using the `var.test()` function.

```
var.test(atmos$moisture ~ atmos$treatment)
##
##  F test to compare two variances
##
## data:  atmos$moisture by atmos$treatment
## F = 0.57919, num df = 9, denom df = 9, p-value = 0.4283
## alternative hypothesis: true ratio of variances is not equal to 1
## 95 percent confidence interval:
##  0.1438623 2.3318107
## sample estimates:
## ratio of variances
##           0.5791888
```

As the *p* value is greater than 0.05, there is no evidence to suggest that the variances are unequal. Note however, that the *F*-test is sensitive to departures from normality and should not be used with data which is not normal. See the `car` package for alternatives.

The non-parametric two-sample Wilcoxon test (also known as a Mann-Whitney U test) can be performed using the same formula method:

```
wilcox.test(atmos$moisture ~ atmos$treatment)
##
##  Wilcoxon rank sum exact test
##
## data:  atmos$moisture by atmos$treatment
## W = 79, p-value = 0.02881
## alternative hypothesis: true location shift is not equal to 0
```

You can also use the `t.test()` and `wilcox.test()` functions to test paired data. Paired data are where there are two measurements on the same experimental unit (individual, site etc) and essentially tests for whether the mean difference between the paired observations is equal to zero. For example, the `pollution` dataset gives the biodiversity score of aquatic invertebrates collected using kick samples in 17 different rivers. These data are paired because two samples were taken on each river, one upstream of a paper mill and one downstream.

```
pollution <- read.table('data/pollution.txt', header = TRUE)
str(pollution)
## 'data.frame':    16 obs. of  2 variables:
##  $ down: int  20 15 10 5 20 15 10 5 20 15 ...
##  $ up  : int  23 16 10 4 22 15 12 7 21 16 ...
```

Note, in this case these data are in the wide format with upstream and downstream values in separate columns (see Chapter 3 on how to convert to long format if you want). To conduct a paired t test use the `paired = TRUE` argument.

```
t.test(pollution$down, pollution$up, paired = TRUE)
##
##  Paired t-test
##
## data:  pollution$down and pollution$up
## t = -3.0502, df = 15, p-value = 0.0081
## alternative hypothesis: true mean difference is not equal to 0
## 95 percent confidence interval:
##  -1.4864388 -0.2635612
## sample estimates:
## mean difference
##          -0.875
```

The output is almost identical to that of a one-sample t test. It is also possible to perform a non-parametric matched-pairs Wilcoxon test in the same way

```
wilcox.test(pollution$down, pollution$up, paired = TRUE)
## Warning in wilcox.test.default(pollution$down, pollution$up, paired = TRUE):
## cannot compute exact p-value with ties
## Warning in wilcox.test.default(pollution$down, pollution$up, paired = TRUE):
## cannot compute exact p-value with zeroes
##
##  Wilcoxon signed rank test with continuity correction
##
## data:  pollution$down and pollution$up
## V = 8, p-value = 0.01406
## alternative hypothesis: true location shift is not equal to 0
```

The function `prop.test()` can be used to compare two or more proportions. For example, a company wishes to test the effectiveness of an advertising campaign for a particular brand of cat food. The company commissions two polls, one before the advertising campaign and one after, with each poll asking cat owners whether they would buy this brand of cat food. The results are given in the table below

	before	after
would buy	45	71
would not buy	35	32

From the table above we can conclude that 56% of cat owners would buy the cat food before the campaign compared to 69% after. But, has the advertising campaign been a success?

The `prop.test()` function has two main arguments which are given as two vectors. The first vector contains the number of positive outcomes and the second vector the total numbers for each group. So to perform the test we first need to define these vectors

```
buy <- c(45,71)                         # creates a vector of positive outcomes
total <-c((45 + 35), (71 + 32))    # creates a vector of total numbers
prop.test(buy, total)                   # perform the test
##
##   2-sample test for equality of proportions with continuity correction
##
## data:  buy out of total
## X-squared = 2.598, df = 1, p-value = 0.107
## alternative hypothesis: two.sided
## 95 percent confidence interval:
##   -0.27865200  0.02501122
## sample estimates:
##    prop 1    prop 2
## 0.5625000 0.6893204
```

There is no evidence to support that the advertising campaign has changed cat owners opinions of the cat food ($p = 0.1$). Use `?prop.test` to explore additional uses of the binomial proportions test.

We could also analyse the count data in the above example as a Chi-square contingency table. The simplest method is to convert the tabulated table into a 2 x 2 matrix using the `matrix()` function (note, there are many alternative methods of constructing a table like this).

```
buyers <- matrix(c(45, 35, 71, 32), nrow = 2)
buyers
##      [,1] [,2]
## [1,]   45   71
## [2,]   35   32
```

Notice that you enter the data column wise into the matrix and then specify the number of rows using `nrow =`.

We can also change the row names and column names from the defaults to make it look more like a table (you don't really need to do this to perform a Chi-square test)

```
colnames(buyers) <- c("before", "after")
rownames(buyers) <- c("buy", "notbuy")
buyers
##         before after
## buy         45    71
## notbuy      35    32
```

You can then perform a Chi-square test to test whether the number of cat owners buying the cat food is independent of the advertising campaign using the `chisq.test()` function. In this example the only argument is our matrix of counts.

```
chisq.test(buyers)
##
##  Pearson's Chi-squared test with Yates' continuity correction
##
## data:  buyers
## X-squared = 2.598, df = 1, p-value = 0.107
```

There is no evidence ($p = 0.107$) to suggest that we should reject the null hypothesis that the number of cat owners buying the cat food is independent of the advertising campaign. You may have spotted that for a 2x2 table, this test is exactly equivalent to the `prop.test()`. You can also use the `chisq.test()` function on raw (untabulated) data and to test for goodness of fit (see `?chisq.test` for more details).

6.2 Correlation

In R, the Pearson's product-moment correlation coefficient between two continuous variables can be estimated using the `cor()` function. Using the `trees` data set again, we can determine the correlation coefficient of the association between tree `Height` and `Volume`.

```
data(trees)
str(trees)
## 'data.frame':    31 obs. of  3 variables:
##  $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
##  $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
##  $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
cor(trees$Height, trees$Volume)
## [1] 0.5982497
```

or we can produce a matrix of correlation coefficients for all variables in a data frame

```
cor(trees)
##             Girth    Height    Volume
## Girth   1.0000000 0.5192801 0.9671194
## Height 0.5192801 1.0000000 0.5982497
## Volume 0.9671194 0.5982497 1.0000000
```

Note that the correlation coefficients are identical in each half of the matrix. Also, be aware that, although a matrix of coefficients can be useful, a little commonsense should be used when using cor() on data frames with numerous variables. It is not good practice to trawl through these types of matrices in the hope of finding large coefficients without having an *a priori* reason for doing so and remember the correlation coefficient assumes that associations are linear.

If you have missing values in the variables you are trying to correlate, cor() will return an error message (as will many functions in R). You will either have to remove these observations (be very careful if you do this) or tell R what to do when an observation is missing. A useful argument you can use with the cor() function is use = "complete.obs".

```
cor(trees, use = "complete.obs")
##             Girth    Height    Volume
## Girth   1.0000000 0.5192801 0.9671194
## Height 0.5192801 1.0000000 0.5982497
## Volume 0.9671194 0.5982497 1.0000000
```

The function cor() will return the correlation coefficient of two variables, but gives no indication whether the coefficient is significantly different from zero. To do this you need to use the function cor.test().

```
cor.test(trees$Height, trees$Volume)
##
##  Pearson's product-moment correlation
##
## data:  trees$Height and trees$Volume
## t = 4.0205, df = 29, p-value = 0.0003784
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.3095235 0.7859756
## sample estimates:
##       cor
## 0.5982497
```

Two non-parametric equivalents to Pearson correlation are available within the `cor.test()` function; Spearman's rank and Kendall's tau coefficient. To use either of these simply include the argument `method = "spearman"` or `method = "kendall"` depending on the test you wish to use. For example

```r
cor.test(trees$Height, trees$Volume, method = "spearman")
## Warning in cor.test.default(trees$Height, trees$Volume, method = "spearman"):
## Cannot compute exact p-value with ties
##
##  Spearman's rank correlation rho
##
## data:  trees$Height and trees$Volume
## S = 2089.6, p-value = 0.0006484
## alternative hypothesis: true rho is not equal to 0
## sample estimates:
##       rho
## 0.5787101
```

6.3   Simple linear modelling

Linear models are one of the most widely used models in statistics and data science. They are often thought of as simple models but they're very flexible and able to model a wide variety of experimental and survey designs. Many of the statistical approaches you may have used previously (such as linear regression, *t*-test, ANOVA, ANCOVA etc) can be expressed as a linear model so the good news is that you're probably already familiar with linear models (albeit indirectly). They also form the foundation of more complicated modelling approaches and are relatively easy to extended to incorporate additional complexity. During this section we'll learn how to fit some simple linear models using R and cover some of the more common applications. We won't go into any detail of the underlying linear modelling theory but rather focus on the practicalities of model fitting and R code.

The main function for fitting linear models in R is the `lm()` function (short for linear model!). The `lm()` function has many arguments but the most important is the first argument which specifies the model you want to fit using a *model formula* which typically takes the general form:

response variable ~ explanatory variable(s)

This model formula is simply read as

> *'variation in the response variable modelled as a function (~) of the explanatory variable(s)'.*

The response variable is also commonly known as the 'dependent variable' and the explanatory variables are sometimes referred to as 'independent variables' (or less frequently as 'predictor variables'). There is also an additional term in our model formula which represents the variation in our response variable **not** explained by our explanatory variables but you don't need to specify this when using the `lm()` function.

As mentioned above, many of the statistical 'tests' you might have previously used can be expressed as a linear model. For example, if we wanted to perform a bivariate linear regression between a response variable (`y`) and a single continuous explanatory variable (`x`) our model formula would simply be

```
y ~ x
```

On the other hand, if we wanted to use an ANOVA to test whether the group means of a response variable (`y`) were different between a three level factor (`x`) our model formula would look like

```
y ~ x
```

OK, hang on, they both look identical, what gives? In addition to the model formula, the type of linear model you fit is also determined by the type of data in your explanatory variable(s) (i.e. what class of data). If your explanatory variable is continuous then you will fit a bivariate linear regression. If your explanatory variable is a factor (i.e. categorical data) you will fit an ANOVA type model.

You can also increase the complexity of your linear model by including additional explanatory variables in your model formula. For example, if we wanted to fit a two-way ANOVA both of our explanatory variables `x` and `z` would need to be factors and separated by a `+` symbol

```
y ~ x + z
```

If we wanted to perform a factorial ANOVA to identify an interaction between both explanatory variables we would separate our explanatory variables with a `:` symbol whilst also including our main effects in our model formula

```
y ~ x + z + x:z
```

or by using the equivalent shortcut notation

```
y ~ x * z
```

It's important that you get comfortable with using model formula (and we've only given the briefest of explanations above) when using the `lm()` function (and other functions) as it's remarkably easy to specifiy a model which is either nonsense or isn't the model you really wanted to fit. A summary table of various linear model formula and equivalent R code given below.

Traditional name	Model formula	R code
Bivariate regression	Y ~ X1 (continuous)	`lm(Y ~ X)`
One-way ANOVA	Y ~ X1 (categorical)	`lm(Y ~ X)`
Two-way ANOVA	Y ~ X1 (cat) + X2(cat)	`lm(Y ~ X1 + X2)`
ANCOVA	Y ~ X1 (cat) + X2(cont)	`lm(Y ~ X1 + X2)`
Multiple regression	Y ~ X1 (cont) + X2(cont)	`lm(Y ~ X1 + X2)`
Factorial ANOVA	Y ~ X1 (cat) * X2(cat)	`lm(Y ~ X1 * X2)` or `lm(Y ~ X1 + X2 + X1:X2)`

OK, time for an example. The data file `smoking.txt` summarises the results of a study investigating the possible relationship between mortality rate and smoking across 25 occupational groups in the UK. The variable `occupational.group` specifies the different occupational groups studied, the `risk.group` variable indicates the relative risk to lung disease for the various occupational groups and `smoking` is an index of the average number of cigarettes smoked each day (relative to the number smoked across all occupations). The variable `mortality` is an index of the death rate from lung cancer in each group (relative to the death rate across all occupational groups). In this data set, the response variable is `mortality` and the potential explanatory variables are `smoking` which is numeric and `risk.group` which is a three level factor. The first thing to do is import our data file using the `read.table()` function as usual and assign the data to an object called `smoke`. You can find a link to download these data here.

```
smoke <- read.table('data/smoking.txt', header = TRUE, sep = "\t",
                                  stringsAsFactors = TRUE)
str(smoke, vec.len = 2)
## 'data.frame':    25 obs. of  4 variables:
##  $ occupational.group: Factor w/ 25 levels "Administrators",..: 9 14 2 11 10 ...
##  $ risk.group        : Factor w/ 3 levels "high","low","medium": 2 1 1 3 1 ...
##  $ smoking           : int  77 137 117 94 116 ...
##  $ mortality         : int  84 116 123 128 155 ...
# vec.len argument to limited number of ' first elements' to display
```

Next, let's investigate the relationship between the `mortality` and `smoking` variables by plotting a scatter plot. We can use either the `ggplot2` package or base R graphics to do this. We'll use `ggplot2` this time and our old friend the `ggplot()` function.

```r
library(ggplot2)
ggplot(mapping = aes(x = smoking, y = mortality), data = smoke) +
    geom_point()
```
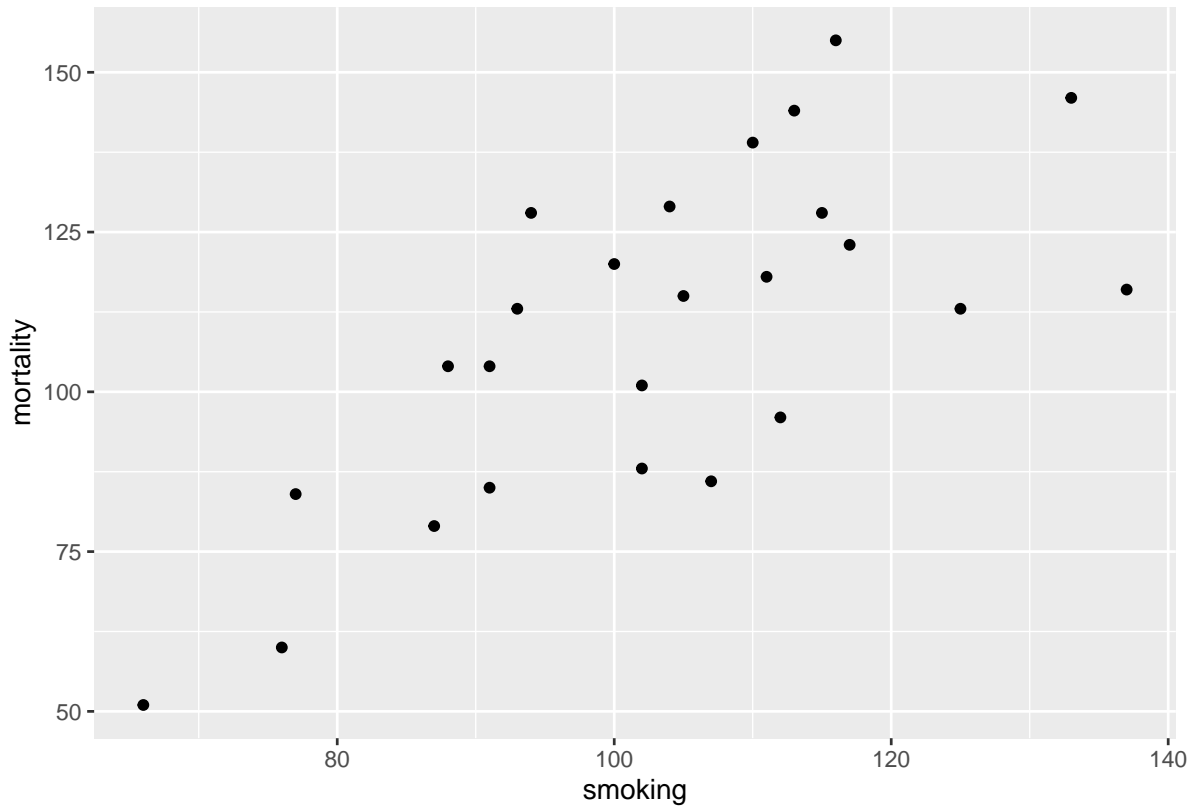

The plot does suggest that there is a positive relationship between the smoking index and mortality index.

To fit a simple linear model to these data we will use the `lm()` function and include our model formula `mortality ~ smoking` and assign the results to an object called `smoke_lm`.

```r
smoke_lm <- lm(mortality ~ smoking, data = smoke)
```

Notice that we have not used the `$` notation to specify the variables in our model formula, instead we've used the `data = smoke` argument. Although the `$` notation will work (i.e. `smoke$mortality ~ smoke$smoking`) it will more than likely cause you problems later on and should be avoided. In fact, we would go as far to suggest that if any function has a `data =` argument you should **always** use it. How do you know if a function has a `data =` argument? Just look in the associated help file.

Perhaps somewhat confusingly (at least at first) it appears that nothing much has happened, you don't automatically get the voluminous output that you normally get with other statistical packages. In fact, what R does, is store the output of the analysis in what is known as a `lm` class object (which we have called `smoke_lm`) from which you

are able to extract exactly what you want using other functions. If you're brave, you can examine the structure of the `smoke_lm` model object using the `str()` function.

```
str(smoke_lm)
## List of 12
##  $ coefficients : Named num [1:2] -2.89 1.09
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "smoking"
##  $ residuals    : Named num [1:25] 3.15 -30.11 -1.36 28.66 31.73 ...
##   ..- attr(*, "names")= chr [1:25] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:25] -545 -91.63 2.72 26.99 35.56 ...
##   ..- attr(*, "names")= chr [1:25] "(Intercept)" "smoking" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:25] 80.9 146.1 124.4 99.3 123.3 ...
##   ..- attr(*, "names")= chr [1:25] "1" "2" "3" "4" ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:25, 1:2] -5 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:25] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "(Intercept)" "smoking"
##   .. ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.2 1.46
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 23
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = mortality ~ smoking, data = smoke)
##  $ terms        :Classes 'terms', 'formula'  language mortality ~ smoking
##   .. ..- attr(*, "variables")= language list(mortality, smoking)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "mortality" "smoking"
##   .. .. .. ..$ : chr "smoking"
##   .. ..- attr(*, "term.labels")= chr "smoking"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(mortality, smoking)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "mortality" "smoking"
##  $ model        :'data.frame':   25 obs. of  2 variables:
##   ..$ mortality: int [1:25] 84 116 123 128 155 101 118 113 104 88 ...
##   ..$ smoking  : int [1:25] 77 137 117 94 116 102 111 93 88 102 ...
```

```
##    ..- attr(*, "terms")=Classes 'terms', 'formula'  language mortality ~ smoking
##    .. .. ..- attr(*, "variables")= language list(mortality, smoking)
##    .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##    .. .. .. ..- attr(*, "dimnames")=List of 2
##    .. .. .. .. ..$ : chr [1:2] "mortality" "smoking"
##    .. .. .. .. ..$ : chr "smoking"
##    .. .. ..- attr(*, "term.labels")= chr "smoking"
##    .. .. ..- attr(*, "order")= int 1
##    .. .. ..- attr(*, "intercept")= int 1
##    .. .. ..- attr(*, "response")= int 1
##    .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##    .. .. ..- attr(*, "predvars")= language list(mortality, smoking)
##    .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##    .. .. .. ..- attr(*, "names")= chr [1:2] "mortality" "smoking"
##  - attr(*, "class")= chr "lm"
```

To obtain a summary of our analysis we can use the `summary()` function on our `smoke_lm`
model object.

```
summary(smoke_lm)
##
## Call:
## lm(formula = mortality ~ smoking, data = smoke)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -30.107 -17.892   3.145  14.132  31.732
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -2.8853    23.0337  -0.125    0.901
## smoking       1.0875     0.2209   4.922 5.66e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 18.62 on 23 degrees of freedom
## Multiple R-squared:  0.513,  Adjusted R-squared:  0.4918
## F-statistic: 24.23 on 1 and 23 DF,  p-value: 5.658e-05
```

This shows you everything you need to know about the parameter estimates (intercept
and slope), their standard errors and associated t statistics and p values. The estimate
for the Intercept suggests that when the relative smoking index is 0 the relative mortality
rate is `-2.885`! The p value associated with the intercept tests the null hypothesis that
the intercept is equal to zero. As the p value is large we fail to reject this null hypothesis.

The `smoking` parameter estimate (`1.0875`) is the estimate of the slope and suggests that for every unit increase in the average number of cigarettes smoked each day the mortality risk index increases by 1.0875. The p value associated with the `smoking` parameter tests whether the slope of this relationship is equal to zero (i.e. no relationship). As our p value is small we reject this null hypothesis and therefore the slope is different from zero and therefore there is a significant relationship. The summary table also includes other important information such as the coefficient of determination (R^2), adjusted R^2 , F statistic, associated degrees of freedom and p value. This information is a condensed form of an ANOVA table which you can see by using the `anova()` function.

```
anova(smoke_lm)
## Analysis of Variance Table
##
## Response: mortality
##           Df Sum Sq Mean Sq F value    Pr(>F)
## smoking    1 8395.7  8395.7  24.228 5.658e-05 ***
## Residuals 23 7970.3   346.5
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Now let's fit another linear model, but this time we will use the `risk.group` variable as our explanatory variable. Remember the `risk.group` variable is a factor and so our linear model will be equivalent to an ANOVA type analysis. We will be testing the null hypothesis that there is no difference in the mean mortality rate between the `low`, `medium` and `high` groups. We fit the model in exactly the same way as before.

```
smoke_risk_lm <- lm(mortality ~ risk.group, data = smoke)
```

Again, we can produce an ANOVA table using the `anova()` function

```
anova(smoke_risk_lm)
## Analysis of Variance Table
##
## Response: mortality
##            Df  Sum Sq Mean Sq F value    Pr(>F)
## risk.group  2 11514.4  5757.2  26.107 1.554e-06 ***
## Residuals  22  4851.6   220.5
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The results presented in the ANOVA table suggest that we can reject the null hypothesis (very small p value) and therefore the mean mortality rate index is different between `low`, `medium` and `high` risk groups.

As we did with our first linear model we can also produce a summary of the estimated parameters using the `summary()` function.

```
summary(smoke_risk_lm)
##
## Call:
## lm(formula = mortality ~ risk.group, data = smoke)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -26.17 -11.45   4.00   9.00  26.83
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)        135.00       5.25  25.713  < 2e-16 ***
## risk.grouplow      -57.83       8.02  -7.211 3.16e-07 ***
## risk.groupmedium   -27.55       6.90  -3.992 0.000615 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 14.85 on 22 degrees of freedom
## Multiple R-squared:  0.7036, Adjusted R-squared:  0.6766
## F-statistic: 26.11 on 2 and 22 DF,  p-value: 1.554e-06
```

In the summary table the Intercept is set to the first level of `risk.group` (`high`) as this occurs first alphabetically. Therefore, the estimated mean mortality index for `high` risk individuals is `135`. The estimates for `risk.grouplow` and `risk.groupmedium` are mean differences from the intercept (`high` group). So the mortality index for the `low` group is `135` - `57.83` = `77.17` and for the `medium` group is `135` - `27.55` = `107.45`. The t values and p values in the summary table are associated with testing specific hypotheses. The p value associated with the intercept tests the null hypothesis that the mean mortality index for the `high` group is equal to zero. To be honest this is not a particularly meaningful hypothesis to test but we can reject it anyway as we have a very small p value. The p value for the `risk.grouplow` parameter tests the null hypothesis that the mean difference between `high` and `low` risk groups is equal to zero (i.e. there is no difference). Again we reject this null hypothesis and conclude that the means are different between these two groups. Similarly, the p value for `risk.groupmedium` tests the null hypothesis that the mean difference between `high` and `medium` groups is equal to zero which we also reject.

Don't worry too much if you find the output from the `summary()` function a little confusing. Its takes a bit of practice and experience to be able to make sense of all the numbers. Remember though, the more complicated your model is, the more complicated your interpretion will be. And always remember, a model that you can't interpret is not worth fitting (most of the time!).

Another approach to interpreting your model output is to plot a graph of your data and then add the fitted model to this plot. Let's go back to the first linear model we fitted

(`smoke_lm`). We can add the fitted line to our previous plot using the `ggplot2` package and the `geom_smooth` geom. We can easily include the standard errors by specifying the `se = TRUE` argument.

```
ggplot(mapping = aes(x = smoking, y = mortality), data = smoke) +
    geom_point() +
    geom_smooth(method = "lm", se = TRUE)
```
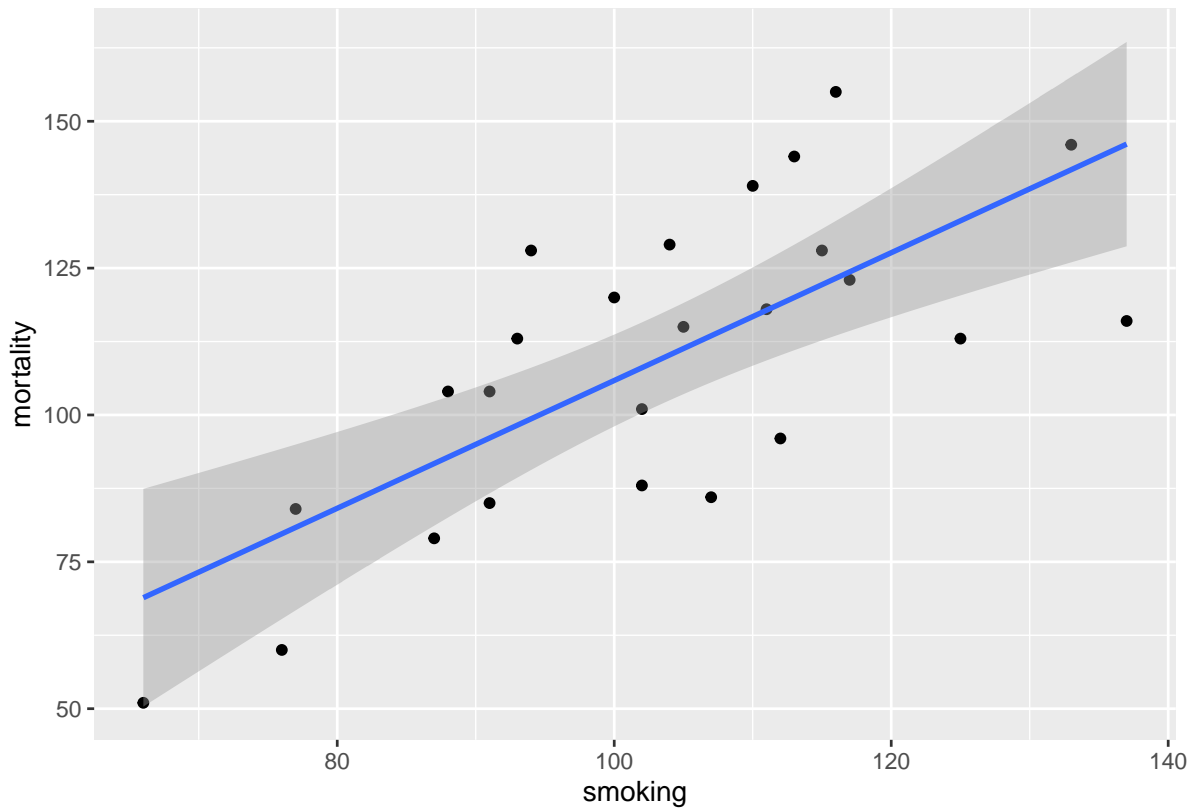

You can also do this with R's base graphics. Note though that the fitted line extends beyond the data which is not great practice. If you want to prevent this you can generate predicted values from the model using the `predict()` function within the range of your data and then add these values to the plot using the `lines()` function (not shown).

```
plot(smoke$smoking, smoke$mortality, xlab = "smoking rate", ylab = " mortality rate")
abline(smoke_lm, col = "red")
```
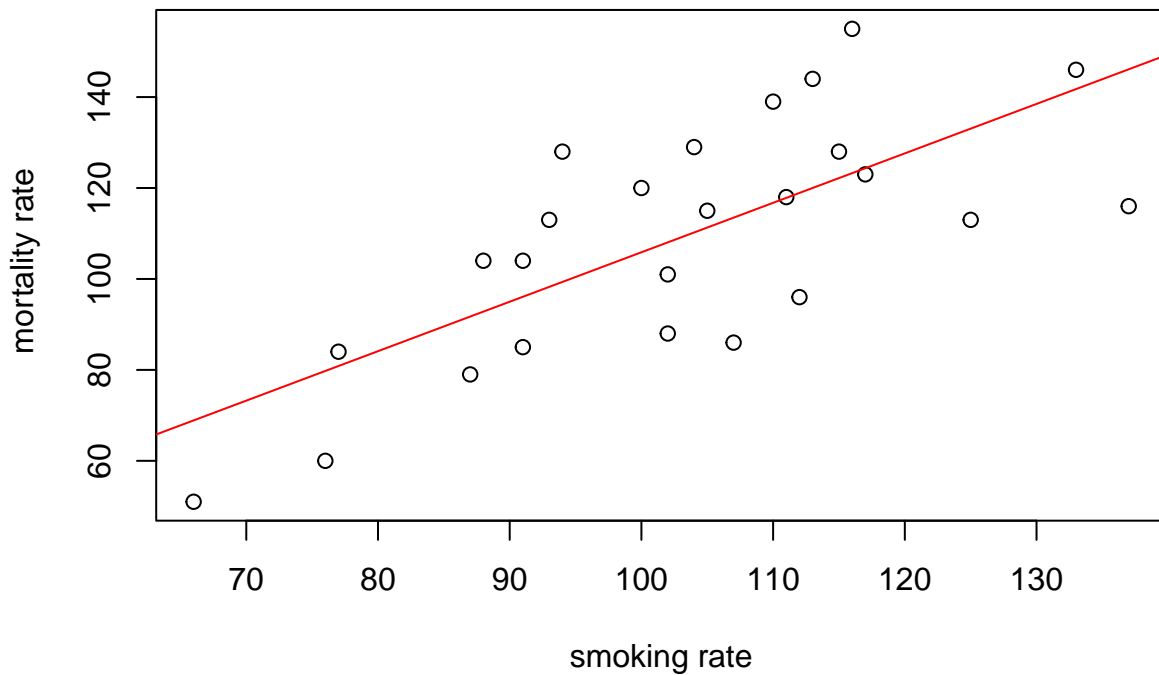
Before we sit back and relax and admire our model (or go write that high impact paper your supervisor/boss has been harassing you about) our work is not finished. It's vitally important to check the underlying assumptions of your linear model. Two of the most important assumption are equal variances (homogeneity of variance) and normality of residuals. To check for equal variances we can construct a graph of residuals versus fitted values. We can do this by first extracting the residuals and fitted values from our model object using the `resid()` and `fitted()` functions.

```r
smoke_res <- resid(smoke_lm)
smoke_fit <- fitted(smoke_lm)
```

And then plot them using `ggplot` or base R graphics.

```r
ggplot(mapping = aes(x = smoke_fit, y = smoke_res)) +
    geom_point() +
    geom_hline(yintercept = 0, colour = "red", linetype = "dashed")
```
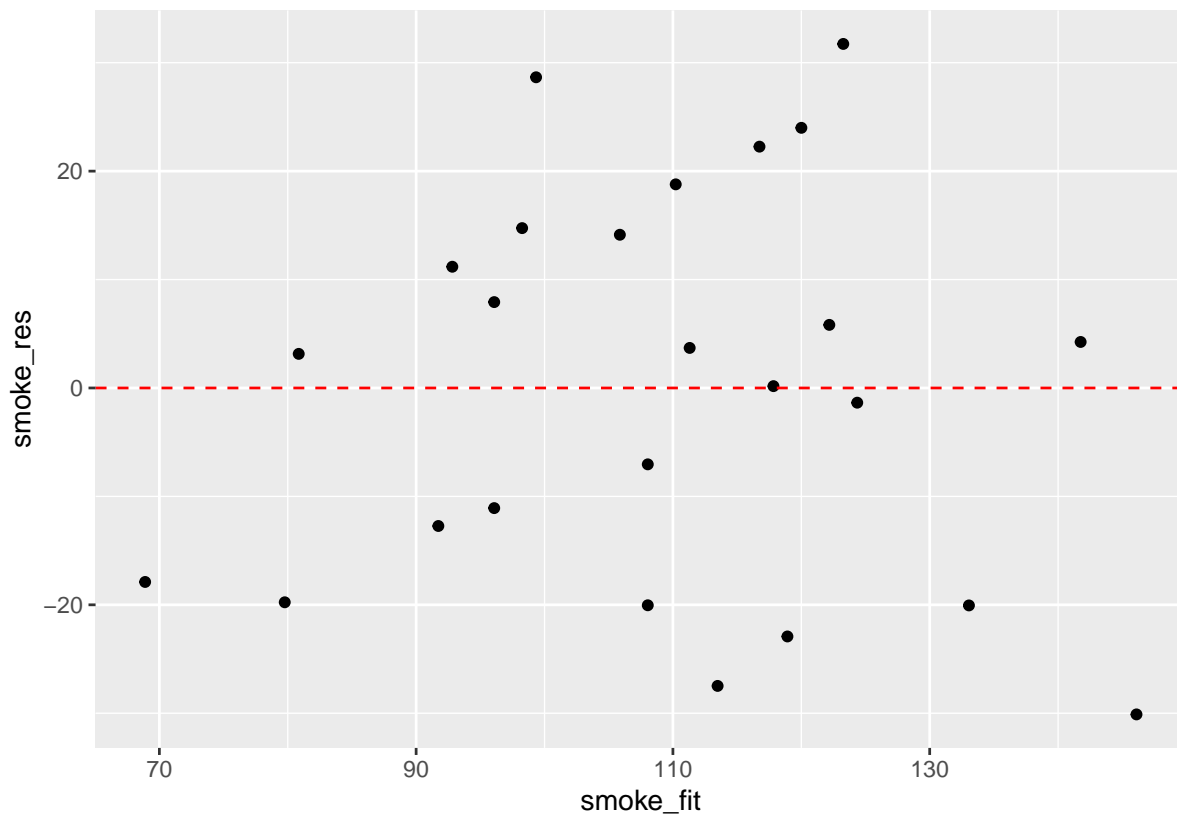
It takes a little practice to interpret these types of graph, but what you are looking for is no pattern or structure in your residuals. What you definitely don't want to see is the scatter increasing around the zero line (red dashed line) as the fitted values get bigger (this has been described as looking like a trumpet, a wedge of cheese or even a slice of pizza) which would indicate unequal variances (heteroscedacity).

To check for normality of residuals we can use our old friend the Q-Q plot using the residuals stored in the `smoke_res` object we created earlier.

```
ggplot(mapping = aes(sample = smoke_res)) +
    stat_qq() +
    stat_qq_line()
```
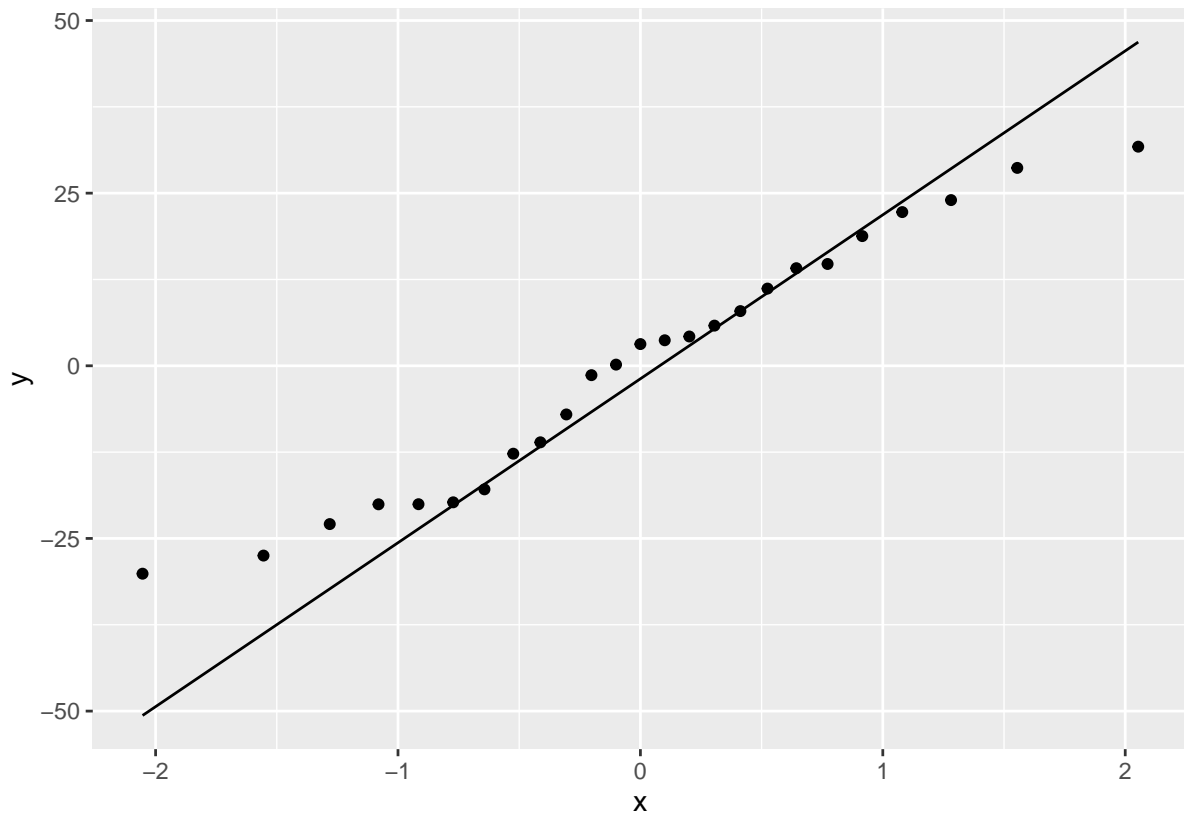
Or the same plot with base graphics.

```r
qqnorm(smoke_res)
qqline(smoke_res)
```

Normal Q–Q Plot
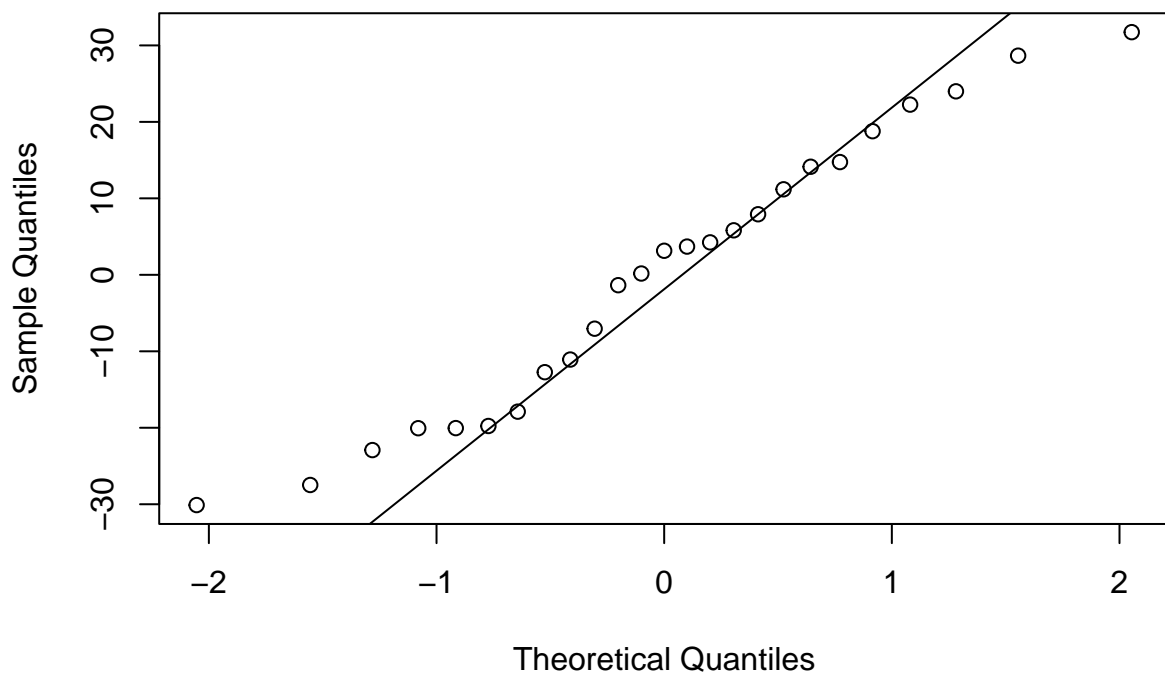
Alternatively, you can get R to do most of the hard work by using the `plot()` function on the model object `smoke_lm`. Before we do this we should tell R that we want to plot four graphs in the same plotting window in RStudio using the `par(mfrow = c(2,2))`. This command splits the plotting window into 2 rows and 2 columns.

```
par(mfrow = c(2,2))
plot(smoke_lm)
```
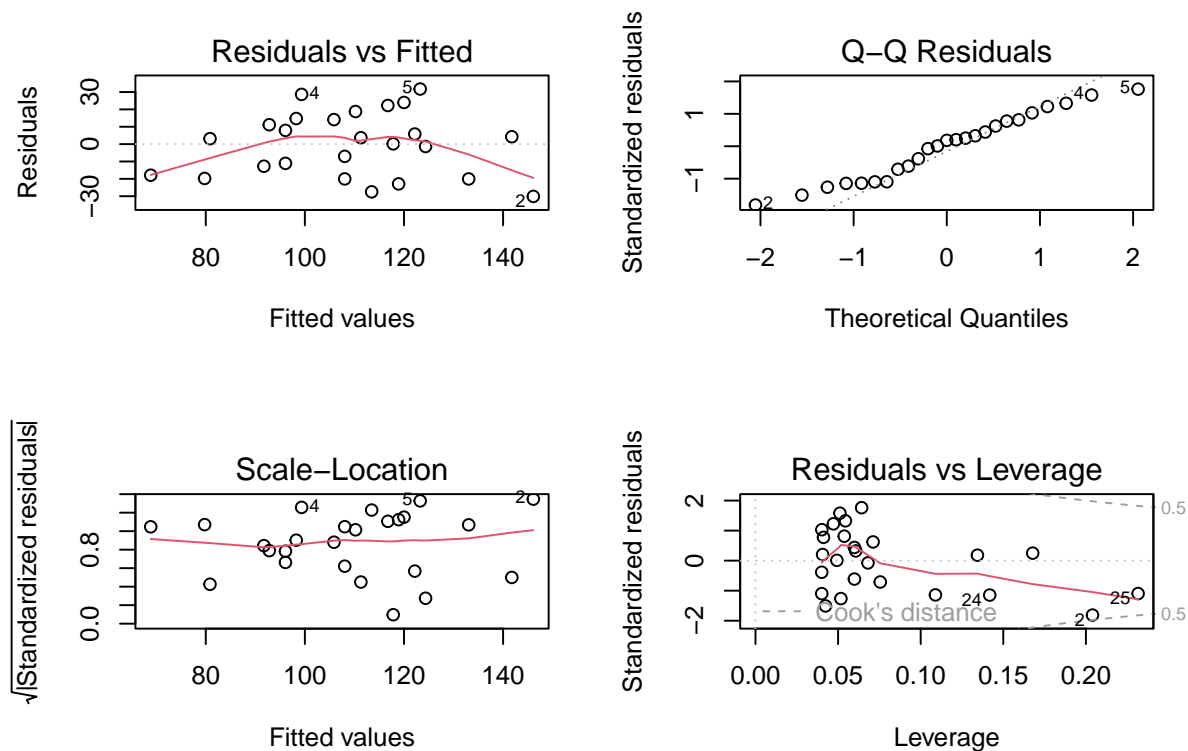

The first two graphs (top left and top right) are the same residual versus fitted and Q-Q plots we produced before. The third graph (bottom left) is the same as the first but plotted on a different scale (the absolute value of the square root of the standardised residuals) and again you are looking for no pattern or structure in the data points. The fourth graph (bottom right) gives you an indication whether any of your observations are having a large influence (Cook's distance) on your regression coefficient estimates. Levearge identifies observations which have unusually large values in their explanatory variables.

You can also produce these diagnostic plots using `ggplot` by installing the package `ggfortify` and using the `autoplot()` function.

```
library(ggfortify)
autoplot(smoke_lm, which = 1:6, ncol = 2, label.size = 3)
```
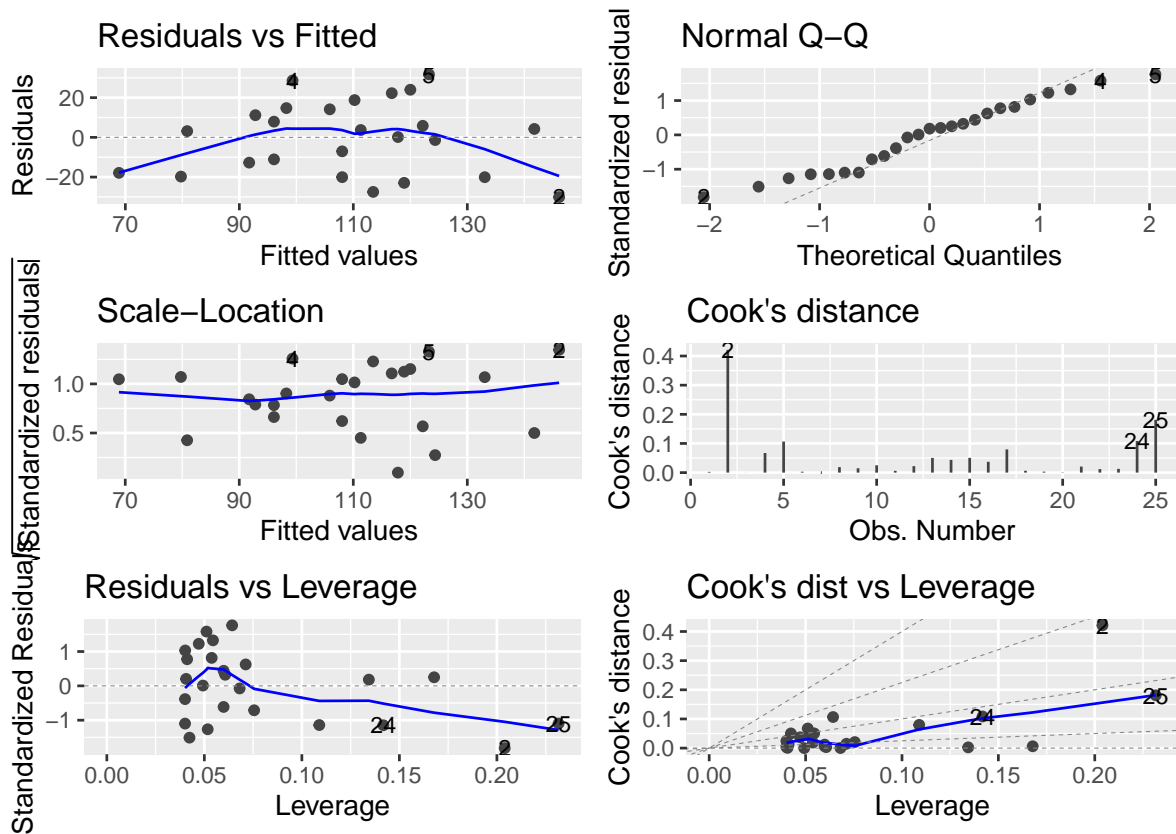
What you do about influential data points or data points with high leverage is up to you. If you would like to examine the effect of removing one of these points on the parameter estimates you can use the `update()` function. Let's remove data point 2 (miners, mortality = 116 and smoking = 137) and store the results in a new object called `smoke_lm2`. Note, we do this to demonstrate the use of the `update()` function. You should think long and hard about removing any data point(s) and if you do you should **always** report this and justify your reasoning.

```
smoke_lm2 <- update(smoke_lm, subset = -2)
summary(smoke_lm2)
##
## Call:
## lm(formula = mortality ~ smoking, data = smoke, subset = -2)
##
## Residuals:
##      Min       1Q    Median       3Q       Max
## -29.7425 -11.6920   -0.4745   13.6141   28.7587
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -20.0755    23.5798  -0.851    0.404
## smoking       1.2693     0.2297   5.526 1.49e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 17.62 on 22 degrees of freedom
## Multiple R-squared:  0.5813, Adjusted R-squared:  0.5622
## F-statistic: 30.54 on 1 and 22 DF,  p-value: 1.488e-05
```

There are numerous other functions which are useful for producing diagnostic plots. For example, `rstandard()` and `rstudent()` returns the standardised and studentised residuals. The function `dffits()` expresses how much an observation influences the associated fitted value and the function `dfbetas()` gives the change in the estimated parameters if an observation is excluded, relative to its standard error (intercept is the solid line and slope is the dashed line in the example below). The solid bold line in the same graph represents the Cook's distance. Examples of how to use these functions are given below.

```r
par(mfrow=c(2,2))
plot(dffits(smoke_lm), type = "l")
plot(rstudent(smoke_lm))
matplot(dfbetas(smoke_lm), type = "l", col = "black")
lines(sqrt(cooks.distance(smoke_lm)), lwd = 2)
```
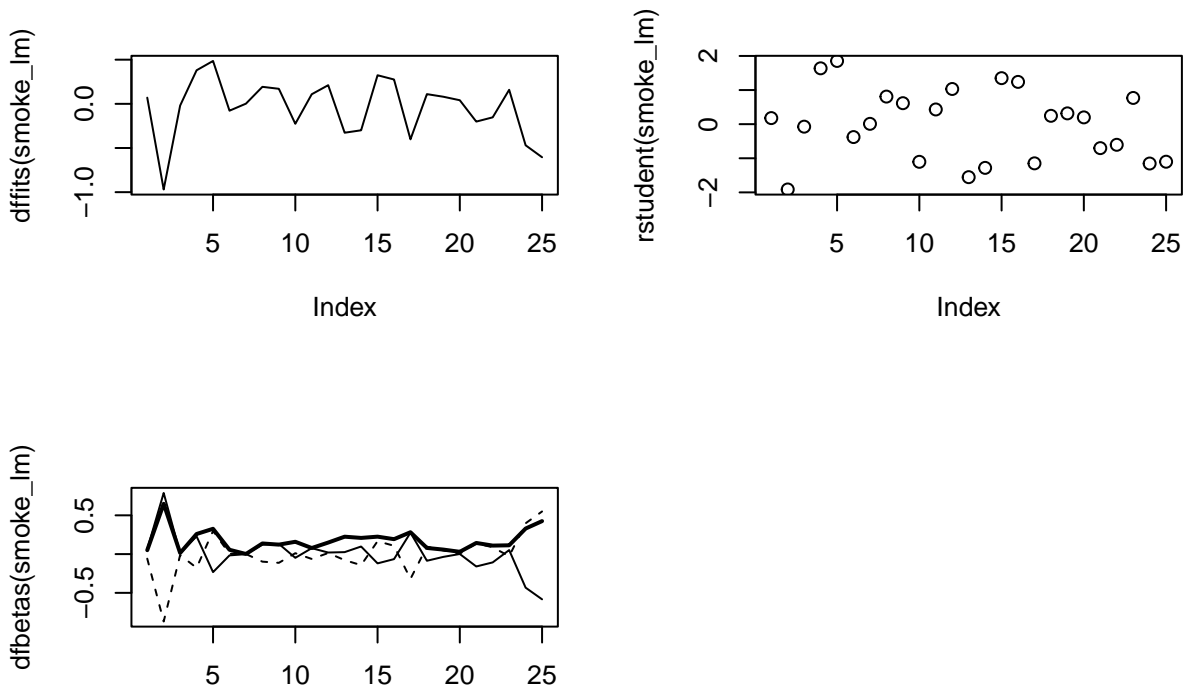

6.4 Other modelling approaches

As with most things R related, a complete description of the variety and flexibility of different statistical analyses you can perform is beyond the scope of this introductory

text. Further information can be found in any of the excellent documents referred to in Chapter 2. A table of some of the more common statistical functions is given below to get you started.

R function	Use
`glm()`	Fit a generalised linear model with a specific error structure specified using the `family =` argument (Poisson, binomial, gamma)
`gam()`	Fit a generalised additive model. The R package `mgcv` must be loaded
`lme()` & `nlme()`	Fit linear and non-linear mixed effects models. The R package `nlme` must be loaded
`lmer()`	Fit linear and generalised linear and non-linear mixed effects models. The package `lme4` must be **installed** and loaded
`gls()`	Fit generalised least squares models. The R package `nlme` must be loaded
`kruskal.test()`	Performs a Kruskal-Wallis rank sum test
`friedman.test()`	Performs a Friedman's test

6.5   Exercise 6

Congratulations, you've reached the end of Chapter 6! Perhaps now's a good time to practice some of what you've learned. You can find an exercise we've prepared for you (and our solutions) on the course website.

Chapter 7

Programming in R

After learning the basics, programming in R is the next big step towards attaining coding nirvana. In R this freedom is gained by being able to wield and manipulate code to get it to do exactly what you want, rather than contort, bend and restrict yourself to what others have built. It's worth remembering that many packages are written with the goal of solving a problem the author had. Your problem may be similar, but not exactly the same. In such cases you could contort, bend and restrict your data so that it works with the package, or you could build your own functions that solve your specific problems.

But there are already a vast number of R packages available, surely more than enough to cover everything you could possibly want to do? Why then, would you ever need to use R as a programming language? Why not just stick to the functions from a package? Well, in some cases you'll want to customise those existing functions to suit your specific needs. Or you may want to implement an approach that's hot off the press or you've come up with an entirely novel idea, which means there won't be any pre-existing packages that work for you. Both of these are not particularly common early-on when you start using R, but you may well find as you progress, the reliance on existing functions becomes a little limiting. In addition, if you ever start using more advanced statistical approaches (i.e. Bayesian inference coded through Stan or JAGS) then an understanding of basic programming constructs, especially loops, becomes fundamental.

In this Chapter we'll explore the basics of the programming side of R. Even if you never create your own function, you will at the very least use functions on a daily basis. Pulling back the curtain and showing the nitty-gritty of how these work will hopefully, in the worst case, help your confidence and in the best case, provide a starting point for doing some clever coding.

7.1   Looking behind the curtain

A good way to start learning to program in R is to see what others have done. We can start by briefly peeking behind the curtain. In Chapter 5 we made use of the `theme_classic()` function when customising our `ggplot` figures. With many functions in R, if you want to have a quick glance at the machinery behind the scenes, we can simply

write the function name but without the `()`. This is the same trick we used in Chapter 5 to alter the `theme_classic()` style of ggplot2 to make `theme_rbook()`.

Note that to view the source code of base R packages (those that come with R) requires some additional steps which we won't cover here (see this link if you're interested), but for most other packages that you install yourself, generally entering the function name without `()` will show the source code of the function.

```
theme_classic
## function (base_size = 11, base_family = "", base_line_size = base_size/22,
##     base_rect_size = base_size/22)
## {
##     theme_bw(base_size = base_size, base_family = base_family,
##         base_line_size = base_line_size, base_rect_size = base_rect_size) %+replace%
##         theme(panel.border = element_blank(), panel.grid.major = element_blank(),
##             panel.grid.minor = element_blank(), axis.line = element_line(colour = "black",
##                 linewidth = rel(1)), strip.background = element_rect(fill = "white",
##                 colour = "black", linewidth = rel(2)), complete = TRUE)
## }
## <bytecode: 0x7f8d36b0fbd8>
## <environment: namespace:ggplot2>
```

What we see above is the underlying code for this particular function. As we did in Chapter 5, we could copy and paste this into our own script and make any changes we deemed necessary. This approach isn't limited to ggplot2 functions and can be used for other functions as well, although tread carefully and test the changes you've made.

Don't worry overly if most of the code contained in functions doesn't make sense immediately. This will be especially true if you are new to R, in which case it seems incredibly intimidating. To help with that, we'll begin by making our own functions in R in the next section.

7.2   Functions in R

Functions are your loyal servants, waiting patiently to do your bidding to the best of their ability. They're made with the utmost care and attention … though sometimes may end up being something of a Frankenstein's monster - with an extra limb or two and a head put on backwards. But no matter how ugly they may be they're completely faithful to you.

They're also very stupid.

If we asked you to go to the supermarket to get us some ingredients to make *Francesinha*, even if you don't know what the heck that is, you'd be able to guess and bring at least *something* back. Or you could decide to make something else. Or you could ask a celebrity chef for help. Or you could pull out your phone and search online for what

Francesinha is. The point is, even if we didn't give you enough information to do the task, you're intelligent enough to, at the very least, try to find a work around.

If instead, we asked our loyal function to do the same, it would listen intently to our request, stand still for a few milliseconds, compose itself, and then start shouting `Error: 'data' must be a data frame, or other object ....`. It would then repeat this every single time we asked it to do the job. The point here, is that code and functions are not intelligent. They cannot find workarounds. It's totally reliant on you, to tell it very explicitly what it needs to do step by step.

Remember two things: the intelligence of code comes from the coder, not the computer and functions need exact instructions to work.

To prevent functions from being *too* stupid you must provide the information the function needs in order for it to function. As with the *Francesinha* example, if we'd supplied a recipe list to the function, it would have managed just fine. We call this "fulfilling an argument". The vast majority of functions require the user to fulfill at least one argument.

This can be illustrated in the pseudocode below. When we make a function we can specify what arguments the user must fulfill (e.g. `argument1` and `argument2`), as well as what to do once it has this information (`expression`):

```
nameOfFunction <- function(argument1, argument2, ...) {expression}
```

The first thing to note is that we've used the function `function()` to create a new function called `nameOfFunction`. To walk through the above code; we're creating a function called `nameOfFunction`. Within the round brackets we specify what information (i.e. arguments) the function requires to run (as many or as few as needed). These arguments are then passed to the expression part of the function. The expression can be any valid R command or set of R commands and is usually contained between a pair of braces `{ }` (if a function is only one line long you can omit the braces). Once you run the above code, you can then use your new function by typing:

```
nameOfFunction(argument1, argument2)
```

Confused? Let's work through an example to help clear things up.

First we are going to create a data frame called `city`, where columns `porto`, `aberdeen`, `nairobi`, and `genoa` are filled with 100 random values drawn from a bag (using the `rnorm()` function to draw random values from a Normal distribution with mean 0 and standard deviation of 1). We also include a "problem", for us to solve later, by including 10 `NA` values within the `nairobi` column (using `rep(NA, 10)`).

```r
city <- data.frame(
  porto = rnorm(100),
  aberdeen = rnorm(100),
  nairobi = c(rep(NA, 10), rnorm(90)),
  genoa = rnorm(100)
)
```

Let's say that you want to multiply the values in the variables Porto and Aberdeen and create a new object called porto_aberdeen. We can do this "by hand" using:

```r
porto_aberdeen <- city$porto * city$aberdeen
```

We've now created an object called porto_aberdeen by multiplying the vectors city$porto and city$aberdeen. Simple. If this was all we needed to do, we can stop here. R works with vectors, so doing these kinds of operations in R is actually much simpler than other programming languages, where this type of code might require loops (we say that R is a vectorised language). Something to keep in mind for later is that doing these kinds of operations with loops can be much slower compared to vectorisation.

But what if we want to repeat this multiplication many times? Let's say we wanted to multiply columns porto and aberdeen, aberdeen and genoa, and nairobi and genoa. In this case we could copy and paste the code, replacing the relevant information.

```r
porto_aberdeen <- city$porto * city$aberdeen
aberdeen_genoa <- city$aberdeen * city$aberdeen
nairobi_genoa <- city$nairobi * city$genoa
```

While this approach works, it's easy to make mistakes. In fact, here we've "forgotten" to change aberdeen to genoa in the second line of code when copying and pasting. This is where writing a function comes in handy. If we were to write this as a function, there is only one source of potential error (within the function itself) instead of many copy-pasted lines of code (which we also cut down on by using a function).

In this case, we're using some fairly trivial code where it's maybe hard to make a genuine mistake. But what if we increased the complexity?

```r
city$porto * city$aberdeen / city$porto + (city$porto * 10^(city$aberdeen))
                - city$aberdeen - (city$porto * sqrt(city$aberdeen + 10))
```

Now imagine having to copy and paste this three times, and in each case having to change the porto and aberdeen variables (especially if we had to do it more than three times).

What we could do instead is generalise our code for x and y columns instead of naming specific cities. If we did this, we could recycle the x * y code. Whenever we wanted to multiple columns together, we assign a city to either x or y. We'll assign the multiplication to the objects `porto_aberdeen` and `aberdeen_nairobi` so we can come back to them later.

```r
# Assign x and y values
x <- city$porto
y <- city$aberdeen

# Use multiplication code
porto_aberdeen <- x * y

# Assign new x and y values
x <- city$aberdeen
y <- city$nairobi

# Reuse multiplication code
aberdeen_nairobi <- x * y
```

This is essentially what a function does. OK down to business, let's call our new function `multiply_columns()` and define it with two arguments, x and y. In the function code we simply return the value of x * y using the `return()` function. Using the `return()` function is not strictly necessary in this example as R will automatically return the value of the last line of code in our function. We include it here to make this explicit.

```r
multiply_columns <- function(x, y) {
  return(x * y)
}
```

Now that we've defined our function we can use it. Let's use the function to multiple the columns `city$porto` and `city$aberdeen` and assign the result to a new object called `porto_aberdeen_func`.

```r
porto_aberdeen_func <- multiply_columns(x = city$porto, y = city$aberdeen)
porto_aberdeen_func
##   [1] -0.0506197327 -0.4432141350  0.3108201105 -0.2919239091 -1.1366253825
##   [6] -0.3399143242  0.0386067903 -0.0544747104  0.0006411174 -2.1307848846
##  [11] -1.2609183635  0.1331581979 -3.3295965306  0.5764768687  0.0781312978
##  [16] -0.0437793593 -0.3253823630 -0.3572483727 -0.2340859154  0.0339204099
##  [21]  0.5439459974  0.6183965126 -0.2428913302  0.0403144399  0.0435016842
##  [26]  0.7049567566 -0.1453578744  0.3484937168 -3.7316204837 -0.0088017265
```

```
##  [31] -0.2376784823 -0.0902000868  0.1179349916 -0.0215942052 -0.3865748654
##  [36] -0.3914259779 -0.3329847718 -0.0355957735  4.0772656249  0.2729821809
##  [41] -0.1704451143  0.1002800649  0.6687341403 -0.4311306371 -2.0069960118
##  [46] -1.0100703945  0.1429819387 -0.2577061965  0.0234090642 -0.2903305085
##  [51]  1.5468207129  0.5075457731 -0.6612056716  0.7181899311 -0.0240039699
##  [56]  0.1513975807 -0.0747482740 -1.4085669960 -0.5782803178  0.4623300714
##  [61]  0.5931229059  1.6677806429 -0.6167697646 -0.0605397617 -0.6233045972
##  [66] -0.0808670626  1.1602354255 -1.1427446696  2.5554826788  0.1857713171
##  [71] -1.5619418626  0.2050820137  0.2140668501  0.6101150976  0.5710091209
##  [76]  0.3790092719 -0.1167624827 -0.8736133682  0.0613002222  0.3692599284
##  [81] -0.0178162316 -0.8446248106  0.1913479908 -0.4570263770  3.3978891988
##  [86]  0.2635802481  0.2436495629 -0.6944883151  1.0399681629  0.7000514854
##  [91]  1.5209518248 -0.6275459443  1.3351369461  0.1898882544  1.1237113915
##  [96] -0.3917062564  1.1856465899 -0.1463636917  0.0132100567  0.0496237979
```

If we're only interested in multiplying `city$porto` and `city$aberdeen`, it would be overkill to create a function to do something once. However, the benefit of creating a function is that we now have that function added to our environment which we can use as often as we like. We also have the code to create the function, meaning we can use it in completely new projects, reducing the amount of code that has to be written (and retested) from scratch each time. As a rule of thumb, you should consider writing a function whenever you've copied and pasted a block of code more than twice.

To satisfy ourselves that the function has worked properly, we can compare the `porto_aberdeen` variable with our new variable `porto_aberdeen_func` using the `identical()` function. The `identical()` function tests whether two objects are *exactly* identical and returns either a `TRUE` or `FALSE` value. Use `?identical` if you want to know more about this function.

```
identical(porto_aberdeen, porto_aberdeen_func)
## [1] TRUE
```

And we confirm that the function has produced the same result as when we do the calculation manually. We recommend getting into a habit of checking that the function you've created works the way you think it has.

Now let's use our `multiply_columns()` function to multiply columns `aberdeen` and `nairobi`. Notice now that argument `x` is given the value `city$aberdeen` and `y` the value `city$nairobi`.

```
aberdeen_nairobi_func <- multiply_columns(x = city$aberdeen, y = city$nairobi)
aberdeen_nairobi_func
##  [1]          NA          NA          NA          NA          NA
##  [6]          NA          NA          NA          NA          NA
## [11] -1.818835640  0.069272301  0.018907325 -0.348491292  0.713220342
```

```
##  [16]   0.617257230 -4.594103324   1.688394991 -0.279758221   0.008449800
##  [21]  -0.336269721   0.284820505   0.001846761   0.015733845 -0.378288644
##  [26]   0.829339248 -1.504604685   0.404052839 -0.233627807 -0.159035890
##  [31]  -1.638663797   0.947715298 -0.960145855 -0.020679843   1.362055806
##  [36]  -0.648757823 -0.023641242   0.180527916 -0.017370698 -0.101936670
##  [41]  -0.159893010 -0.638490853   1.765985608   0.010745759 -1.240070109
##  [46]  -1.700468333 -0.183591396 -0.601803446 -0.216330706 -0.371921944
##  [51]  -0.234472164 -0.043227622 -1.536227571 -0.583180162 -0.182803701
##  [56]  -0.579301422 -0.100975934 -1.570994085 -0.204840043   2.201432248
##  [61]   1.015200239   0.832711440   0.203847374 -0.331274664 -1.088656144
##  [66]  -0.392745474   2.224547956   1.189232601   1.108834420   0.345535354
##  [71]  -0.664589864   1.046810343   0.046499349 -0.989949412 -0.107914610
##  [76]   0.376635421 -0.087812899 -0.826459387 -0.057487138   0.121719362
##  [81]  -0.137320600   0.328071801   0.022542978 -0.389179273   0.191388409
##  [86]   0.119479952 -0.847346250 -0.955426387   0.497962007   2.322256787
##  [91]   1.539839313 -0.623016575 -1.777796553   0.965421151   3.169734617
##  [96]   0.096019019 -0.320122350 -0.003816275   0.074307573 -0.142337877
```

So far so good. All we've really done is wrapped the code x * y into a function, where we ask the user to specify what their x and y variables are.

Now let's add a little complexity. If you look at the output of nairobi_genoa some of the calculations have produced NA values. This is because of those NA values we included in nairobi when we created the city data frame. Despite these NA values, the function appeared to have worked but it gave us no indication that there might be a problem. In such cases we may prefer if it had warned us that something was wrong. How can we get the function to let us know when NA values are produced? Here's one way.

```r
multiply_columns <- function(x, y) {
  temp_var <- x * y
  if (any(is.na(temp_var))) {
    warning("The function has produced NAs")
    return(temp_var)
  } else {
    return(temp_var)
  }
}


aberdeen_nairobi_func <- multiply_columns(city$aberdeen, city$nairobi)
## Warning in multiply_columns(city$aberdeen, city$nairobi): The function has
## produced NAs
porto_aberdeen_func <- multiply_columns(city$porto, city$aberdeen)
```

The core of our function is still the same. We still have x * y, but we've now got an extra six lines of code. Namely, we've included some conditional statements, if and

else, to test whether any NAs have been produced and if they have we display a warning message to the user. The next section of this Chapter will explain how these work and how to use them.

7.3   Conditional statements

x * y does not apply any logic. It merely takes the value of x and multiplies it by the value of y. Conditional statements are how you inject some logic into your code. The most commonly used conditional statement is if. Whenever you see an if statement, read it as *'If X is TRUE, do a thing'*. Including an else statement simply extends the logic to *'If X is TRUE, do a thing, or else do something different'*.

Both the if and else statements allow you to run sections of code, depending on a condition is either TRUE or FALSE. The pseudocode below shows you the general form.

```
if (condition) {
Code executed when condition is TRUE
} else {
Code executed when condition is FALSE
}
```

To delve into this a bit more, we can use an old programmer joke to set up a problem.

———————————————

A programmer's partner says: *'Please go to the store and buy a carton of milk and if they have eggs, get six.'*

The programmer returned with 6 cartons of milk.

When the partner sees this, and exclaims *'Why the heck did you buy six cartons of milk?'*

The programmer replied *'They had eggs'*.

———————————————

At the risk of explaining a joke, the conditional statement here is whether or not the store had eggs. If coded as per the original request, the programmer should bring 6 cartons of milk if the store had eggs (condition = TRUE), or else bring 1 carton of milk if there weren't any eggs (condition = FALSE). In R this is coded as:

```
eggs <- TRUE # Whether there were eggs in the store

  if (eggs == TRUE) { # If there are eggs
  n.milk <- 6 # Get 6 cartons of milk
    } else { # If there are not eggs
  n.milk <- 1 # Get 1 carton of milk
  }
```

We can then check `n.milk` to see how many milk cartons they returned with.

```
n.milk
## [1] 6
```

And just like the joke, our R code has missed that the condition was to determine whether or not to buy eggs, not more milk (this is actually a loose example of the Winograd Scheme, designed to test the *intelligence* of artificial intelligence by whether it can reason what the intended referent of a sentence is).

We could code the exact same egg-milk joke conditional statement using an `ifelse()` function.

```
eggs <- TRUE
n.milk <- ifelse(eggs == TRUE, yes = 6, no = 1)
```

This `ifelse()` function is doing exactly the same as the more fleshed out version from earlier, but is now condensed down into a single line of code. It has the added benefit of working on vectors as opposed to single values (more on this later when we introduce loops). The logic is read in the same way; "If there are eggs, assign a value of 6 to `n.milk`, if there isn't any eggs, assign the value 1 to `n.milk`".

We can check again to make sure the logic is still returning 6 cartons of milk:

```
n.milk
## [1] 6
```

Currently we'd have to copy and paste code if we wanted to change if eggs were in the store or not. We learned above how to avoid lots of copy and pasting by creating a function. Just as with the simple `x * y` expression in our previous `multiply_columns()` function, the logical statements above are straightforward to code and well suited to be turned into a function. How about we do just that and wrap this logical statement up in a function?

```
milk <- function(eggs) {
  if (eggs == TRUE) {
    6
  } else {
    1
  }
}
```

We've now created a function called `milk()` where the only argument is `eggs`. The user of the function specifies if eggs is either `TRUE` or `FALSE`, and the function will then use a conditional statement to determine how many cartons of milk are returned.

Let's quickly try:

```
milk(eggs = TRUE)
## [1] 6
```

And the joke is maintained. Notice in this case we have actually specified that we are fulfilling the `eggs` argument (`eggs = TRUE`). In some functions, as with ours here, when a function only has a single argument we can be lazy and not name which argument we are fulfilling. In reality, it's generally viewed as better practice to explicitly state which arguments you are fulfilling to avoid potential mistakes.

OK, lets go back to the `multiply_columns()` function we created above and explain how we've used conditional statements to warn the user if `NA` values are produced when we multiple any two columns together.

```
multiply_columns <- function(x, y) {
  temp_var <- x * y
  if (any(is.na(temp_var))) {
    warning("The function has produced NAs")
    return(temp_var)
  } else {
    return(temp_var)
  }
}
```

In this new version of the function we still use `x * y` as before but this time we've assigned the values from this calculation to a temporary vector called `temp_var` so we can use it in our conditional statements. Note, this `temp_var` variable is *local* to our function and will not exist outside of the function due something called R's scoping rules. We then use an `if` statement to determine whether our `temp_var` variable contains any `NA` values. The way this works is that we first use the `is.na()` function to test whether each value in our `temp_var` variable is an `NA`. The `is.na()` function returns `TRUE` if the value is an `NA`

and FALSE if the value isn't an NA. We then nest the is.na(temp_var) function inside the function any() to test whether **any** of the values returned by is.na(temp_var) are TRUE. If at least one value is TRUE the any() function will return a TRUE. So, if there are any NA values in our temp_var variable the condition for the if() function will be TRUE whereas if there are no NA values present then the condition will be FALSE. If the condition is TRUE the warning() function generates a warning message for the user and then returns the temp_var variable. If the condition is FALSE the code below the else statement is executed which just returns the temp_var variable.

So if we run our modified multiple_columns() function on the columns city$aberdeen and city$nairobi (which contains NAs) we will receive an warning message.

```
aberdeen_nairobi_func <- multiply_columns(city$aberdeen, city$nairobi)
## Warning in multiply_columns(city$aberdeen, city$nairobi): The function has
## produced NAs
```

Whereas if we multiple two columns that don't contain NA values we don't receive a warning message.

```
porto_aberdeen_func <- multiply_columns(city$porto, city$aberdeen)
```

7.4 Combining logical operators

The functions that we've created so far have been perfectly suited for what we need, though they have been fairly simplistic. Let's try creating a function that has a little more complexity to it. We'll make a function to determine if today is going to be a good day or not based on two criteria. The first criteria will depend on the day of the week (Friday or not) and the second will be whether or not your code is working (TRUE or FALSE). To accomplish this, we'll be using if and else statements. The complexity will come from if statements immediately following the relevant else statement. We'll use such conditional statements four times to achieve all combinations of it being a Friday or not, and if your code is working or not.

```
good.day <- function(code.working, day) {
  if (code.working == TRUE && day == "Friday") {
    "BEST. DAY. EVER. Stop while you are ahead and go to the pub!"
  } else if (code.working == FALSE && day == "Friday") {
    "Oh well, but at least it's Friday! Pub time!"
  } else if (code.working == TRUE && day != "Friday") {
    "So close to a good day... shame it's not a Friday"
  } else if (code.working == FALSE && day != "Friday") {
    "Hello darkness."
```

```
  }
}

good.day(code.working = TRUE, day = "Friday")
## [1] "BEST. DAY. EVER. Stop while you are ahead and go to the pub!"

good.day(FALSE, "Tuesday")
## [1] "Hello darkness."
```

Notice that we never specified what to do if the day was not a Friday? That's because, for this function, the only thing that matters is whether or not it's Friday.

We've also been using logical operators whenever we've used `if` statements. Logical operators are the final piece of the logical conditions jigsaw. Below is a table which summarises operators. The first two are logical operators and the final six are relational operators. You can use any of these when you make your own functions (or loops).

Operator	Technical Description	What it means	Example
&&	Logical AND	Both conditions must be met	`if(cond1 == test && cond2 == test)`
||	Logical OR	Either condition must be met	`if(cond1 == test || cond2 == test)`
<	Less than	X is less than Y	`if(X < Y)`
>	Greater than	X is greater than Y	`if(X > Y)`
<=	Less than or equal to	X is less/equal to Y	`if(X <= Y)`
>=	Greater than or equal to	X is greater/equal to Y	`if(X >= Y)`
==	Equal to	X is equal to Y	`if(X == Y)`
!=	Not equal to	X is not equal to Y	`if(X != Y)`

7.5   Loops

R is very good at performing repetitive tasks. If we want a set of operations to be repeated several times we use what's known as a loop. When you create a loop, R will execute the instructions in the loop a specified number of times or until a specified condition is met. There are three main types of loop in R: the *for* loop, the *while* loop and the *repeat* loop.

Loops are one of the staples of all programming languages, not just R, and can be a powerful tool (although in our opinion, used far too frequently when writing R code).

7.5.1   For loop

The most commonly used loop structure when you want to repeat a task a defined number of times is the `for` loop. The most basic example of a `for` loop is:

```r
for (i in 1:5) {
  print(i)
}
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

But what's the code actually doing? This is a dynamic bit of code were an index `i` is iteratively replaced by each value in the vector `1:5`. Let's break it down. Because the first value in our sequence (`1:5`) is `1`, the loop starts by replacing `i` with `1` and runs everything between the `{ }`. Loops conventionally use `i` as the counter, short for iteration, but you are free to use whatever you like, even your pet's name, it really does not matter (except when using nested loops, in which case the counters must be called different things, like `SenorWhiskers` and `HerrFlufferkins`.

So, if we were to do the first iteration of the loop manually

```r
i <- 1
print(i)
## [1] 1
```

Once this first iteration is complete, the for loop *loops* back to the beginning and replaces `i` with the next value in our `1:5` sequence (`2` in this case):

```r
i <- 2
print(i)
## [1] 2
```

This process is then repeated until the loop reaches the final value in the sequence (`5` in this example) after which point it stops.

To reinforce how `for` loops work and introduce you to a valuable feature of loops, we'll alter our counter within the loop. This can be used, for example, if we're using a loop to iterate through a vector but want to select the next row (or any other value). To show this we'll simply add 1 to the value of our index every time we iterate our loop.

```
for (i in 1:5) {
  print(i + 1)
}
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
```

As in the previous loop, the first value in our sequence is 1. The loop begins by replacing i with 1, but this time we've specified that a value of 1 must be added to i in the expression resulting in a value of 1 + 1.

```
i <- 1
i + 1
## [1] 2
```

As before, once the iteration is complete, the loop moves onto the next value in the sequence and replaces i with the next value (2 in this case) so that i + 1 becomes 2 + 1.

```
i <- 2
i + 1
## [1] 3
```

And so on. We think you get the idea! In essence this is all a for loop is doing and nothing more.

Whilst above we have been using simple addition in the body of the loop, you can also combine loops with functions.

Let's go back to our data frame city. Previously in the Chapter we created a function to multiply two columns and used it to create our porto_aberdeen, aberdeen_nairobi, and nairobi_genoa objects. We could have used a loop for this. Let's remind ourselves what our data look like and the code for the multiple_columns() function.

```
# Recreating our dataset
city <- data.frame(
  porto = rnorm(100),
  aberdeen = rnorm(100),
  nairobi = c(rep(NA, 10), rnorm(90)),
  genoa = rnorm(100)
)
```

```
# Our function
multiply_columns <- function(x, y) {
  temp <- x * y
  if (any(is.na(temp))) {
    warning("The function has produced NAs")
    return(temp)
  } else {
    return(temp)
  }
}
```

To use a list to iterate over these columns we need to first create an empty list (remember lists?) which we call temp (short for temporary) which will be used to store the output of the for loop.

```
temp <- list()
for (i in 1:(ncol(city) - 1)) {
  temp[[i]] <- multiply_columns(x = city[, i], y = city[, i + 1])
}
## Warning in multiply_columns(x = city[, i], y = city[, i + 1]): The function has
## produced NAs

## Warning in multiply_columns(x = city[, i], y = city[, i + 1]): The function has
## produced NAs
```

When we specify our for loop notice how we subtracted 1 from ncol(city). The ncol() function returns the number of columns in our city data frame which is 4 and so our loop runs from i = 1 to i = 4 - 1 which is i = 3. We'll come back to why we need to subtract 1 from this in a minute.

So in the first iteration of the loop i takes on the value 1. The multiply_columns() function multiplies the city[, 1] (porto) and city[, 1 + 1] (aberdeen) columns and stores it in the temp[[1]] which is the first element of the temp list.

The second iteration of the loop i takes on the value 2. The multiply_columns() function multiplies the city[, 2] (aberdeen) and city[, 2 + 1] (nairobi) columns and stores it in the temp[[2]] which is the second element of the temp list.

The third and final iteration of the loop i takes on the value 3. The multiply_columns() function multiplies the city[, 3] (nairobi) and city[, 3 + 1] (genoa) columns and stores it in the temp[[3]] which is the third element of the temp list.

So can you see why we used ncol(city) - 1 when we first set up our loop? As we have four columns in our city data frame if we didn't use ncol(city) - 1 then eventually we'd try to add the 4th column with the non-existent 5th column.

Again, it's a good idea to test that we are getting something sensible from our loop (remember, check, check and check again!). To do this we can use the `identical()` function to compare the variables we created `by` `hand` with each iteration of the loop manually.

```r
porto_aberdeen_func <- multiply_columns(city$porto, city$aberdeen)
i <- 1
identical(multiply_columns(city[, i], city[, i + 1]), porto_aberdeen_func)
## [1] TRUE

aberdeen_nairobi_func <- multiply_columns(city$aberdeen, city$nairobi)
## Warning in multiply_columns(city$aberdeen, city$nairobi): The function has
## produced NAs
i <- 2
identical(multiply_columns(city[, i], city[, i + 1]), aberdeen_nairobi_func)
## Warning in multiply_columns(city[, i], city[, i + 1]): The function has
## produced NAs
## [1] TRUE
```

If you can follow the examples above, you'll be in a good spot to begin writing some of your own for loops. That said there are other types of loops available to you.

7.5.2   While loop

Another type of loop that you may use (albeit less frequently) is the `while` loop. The `while` loop is used when you want to keep looping until a specific logical condition is satisfied (contrast this with the `for` loop which will always iterate through an entire sequence).

The basic structure of the while loop is:

```r
while(logical_condition){ expression }
```

A simple example of a while loop is:

```r
i <- 0
while (i <= 4) {
  i <- i + 1
  print(i)
}
## [1] 1
## [1] 2
```

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

Here the loop will only continue to pass values to the main body of the loop (the `expression` body) when `i` is less than or equal to 4 (specified using the `<=` operator in this example). Once `i` is greater than 4 the loop will stop.

There is another, very rarely used type of loop; the `repeat` loop. The `repeat` loop has no conditional check so can keep iterating indefinitely (meaning a break, or "stop here", has to be coded into it). It's worthwhile being aware of it's existence, but for now we don't think you need to worry about it; the `for` and `while` loops will see you through the vast majority of your looping needs.

7.5.3 When to use a loop?

Loops are fairly commonly used, though sometimes a little overused in our opinion. Equivalent tasks can be performed with functions, which are often more efficient than loops. Though this raises the question when should you use a loop?

In general loops are implemented inefficiently in R and should be avoided when better alternatives exist, especially when you're working with large datasets. However, loop are sometimes the only way to achieve the result we want.

Some examples of when using loops can be appropriate:

- Some simulations (e.g. the Ricker model can, in part, be built using loops)

- Recursive relationships (a relationship which depends on the value of the previous relationship ["to understand recursion, you must understand recursion"])

- More complex problems (e.g., how long since the last badger was seen at site j, given a pine marten was seen at time t, at the same location j as the badger, where the pine marten was detected in a specific 6 hour period, but exclude badgers seen 30 minutes before the pine marten arrival, repeated for all pine marten detections)

- While loops (keep jumping until you've reached the moon)

7.5.4 If not loops, then what?

In short, use the apply family of functions; `apply()`, `lapply()`, `tapply()`, `sapply()`, `vapply()`, and `mapply()`. The apply functions can often do the tasks of most "home-brewed" loops, sometimes faster (though that won't really be an issue for most people) but more importantly with a much lower risk of error. A strategy to have in the back of your mind which may be useful is; for every loop you make, try to remake it using an apply function (often `lapply` or `sapply` will work). If you can, use the apply version. There's

nothing worse than realising there was a small, tiny, seemingly meaningless mistake in a loop which weeks, months or years down the line has propagated into a huge mess. We strongly recommend trying to use the apply functions whenever possible.

lapply

Your go to apply function will often be `lapply()` at least in the beginning. The way that `lapply()` works, and the reason it is often a good alternative to for loops, is that it will go through each element in a list and perform a task (i.e. run a function). It has the added benefit that it will output the results as a list - something you'd have to otherwise code yourself into a loop.

An `lapply()` has the following structure:

```
lapply(X, FUN)
```

Here x is the vector which we want to do *something* to. FUN stands for how much fun this is (just kidding!). It's also short for "function".

Let's start with a simple demonstration first. Let's use the `lapply()` function create a sequence from 1 to 5 and add 1 to each observation (just like we did when we used a for loop):

```
lapply(0:4, function(a) {a + 1})
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 4
##
## [[5]]
## [1] 5
```

Notice that we need to specify our sequence as `0:4` to get the output 1 ,2 ,3 ,4 , 5 as we are adding 1 to each element of the sequence. See what happens if you use 1:5 instead.

Equivalently, we could have defined the function first and then used the function in `lapply()`.

```r
add_fun <- function(a) {a + 1}
lapply(0:4, add_fun)
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 4
##
## [[5]]
## [1] 5
```

The `sapply()` function does the same thing as `lapply()` but instead of storing the results as a list, it stores them as a vector.

```r
sapply(0:4, function(a) {a + 1})
## [1] 1 2 3 4 5
```

As you can see, in both cases, we get exactly the same results as when we used the for loop.

7.6 Exercise 7

Congratulations, you've reached the end of Chapter 7! Perhaps now's a good time to practice some of what you've learned. You can find an exercise we've prepared for you (and our solutions) on the course website.

Chapter 8

Reproducible reports with R markdown

This chapter will introduce you to creating reproducible reports using R markdown to encourage best (or better) practice to facilitate open science. It will first describe what R markdown is and why you might want to consider using it, describe how to create an R markdown document using RStudio and then how to convert this document to a html or pdf formatted report. During this Chapter you will learn about the different components of an R markdown document, how to format text, graphics and tables within the document and finally how to avoid some of the common difficulties using R markdown.

8.1   What is R markdown?

R markdown is a simple and easy to use plain text language used to combine your R code, results from your data analysis (including plots and tables) and written commentary into a single nicely formatted and reproducible document (like a report, publication, thesis chapter or a web page like this one).

Technically, R markdown is a variant of another language (yet another language!) called Markdown and both are a type of 'markup' language. A markup language simply provides a way of creating an easy to read plain text file which can incorporate formatted text, images, headers and links to other documents. Don't worry about the details for the moment, although if you're interested you can find more information about markup languages here. Actually, if it makes you feel any better all of you will have been exposed to a markup language before, as most of the internet content you digest every day is underpinned by a markup language called HTML (**H**yper**t**ext **M**arkup **L**anguage). Anyway, the main point is that R markdown is very easy to learn (much, much easier than HTML) and when used with RStudio it's ridiculously easy to integrate into your workflow to produce feature rich content (so why wouldn't you?!).

8.2 Why use R markdown?

During the previous Chapters we talked a lot about conducting your research in a robust and reproducible manner to facilitate open science. In a nutshell, open science is about doing all we can to make our data, methods, results and inferences transparent and available to everyone. Some of the main tenets of open science are described here and include:

- Transparency in experimental methodology, observation, collection of data and analytical methods.
- Public availability and re-usability of scientific data
- Public accessibility and transparency of scientific communication
- Using web-based tools to facilitate scientific collaboration

By now all of you will (hopefully) be using R to explore and analyse your interesting data. As such, you're already well along the road to making your analysis more reproducible, transparent and shareable. However, perhaps your current workflow looks something like this:
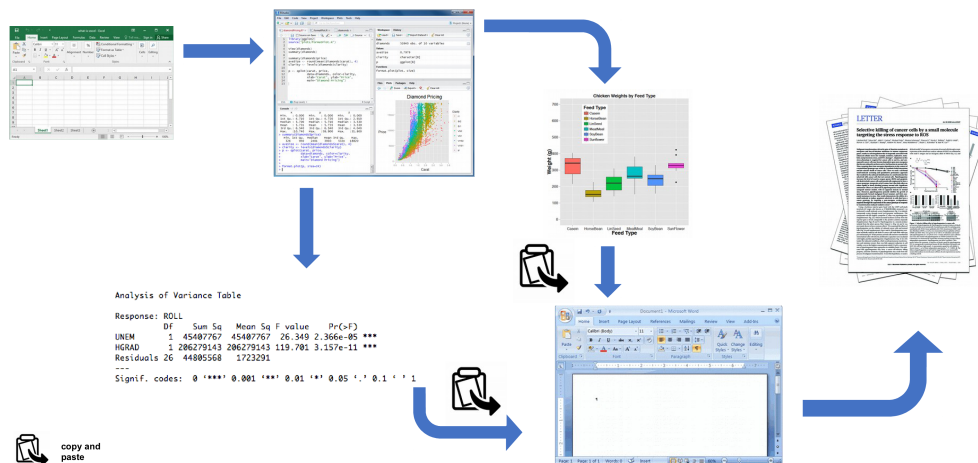
Your data is imported from your favourite spreadsheet software into RStudio (or R), you write your R code to explore and analyse your data, you save plots as external files, copy tables of analysis output and then manually combine all of this and your written prose into a single MS Word document (maybe a paper or thesis chapter). Whilst there is nothing particularly wrong with this approach (and it's certainly better than using point and click software to analyse your data) there are some limitations:

- It's not particularly reproducible. Because this workflow separates your R code from the final document there are multiple opportunities for undocumented decisions to be made (which plots did you use? what analysis did/didn't you include? etc).

- It's inefficient. If you need to go back and change something (create a new plot or update your analysis etc) you will need to create or amend multiple documents increasing the risk of mistakes creeping into your workflow.

- It's difficult to maintain. If your analysis changes you again need to update multiple files and documents.

- It can be difficult to decide what to share with others. Do you share all of your code (initial data exploration, model validation etc) or just the code specific to your final document? It's quite a common (and bad!) practice for researchers to maintain two R scripts, one used for the actual analysis and one to share with the final paper or thesis chapter. This can be both time consuming and confusing and should be avoided.

Perhaps a more efficient and robust workflow would look something like this:
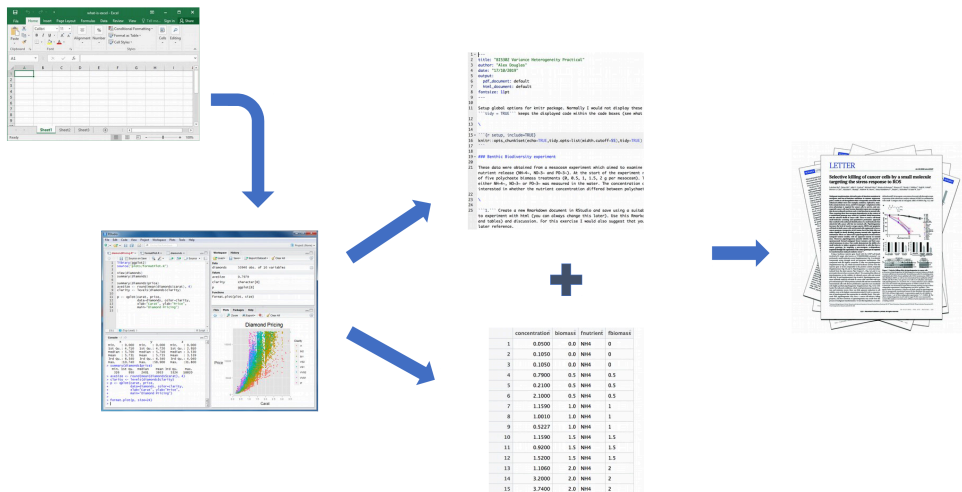
Your data is imported into RStudio (or R) as before but this time all of the R code you used to analyse your data, produce your plots and your written text (Introduction, Materials and Methods, Discussion etc) is contained within a single R markdown document which is then used (along with your data) to automatically create your final document. This is exactly what R markdown allows you to do.

Some of the advantages of using R markdown include:

- Explicitly links your data with your R code and output creating a fully reproducible workflow. **ALL** of the R code used to explore, summarise and analyse your data can be included in a single easy to read document. You can decide what to include in your final document (as you will learn below) but all of your R code can be included in the R markdown document.

- You can create a wide variety of output formats (pdf, html web pages, MS Word and many others) from a single R markdown document which enhances both collaboration and communication.

- Enhances transparency of your research. Your data and R markdown file can be included with your publication or thesis chapter as supplementary material or hosted on a GitHub repository (see the GitHub Chapter).

- Increases the efficiency of your workflow. If you need to modify or extend your current analysis you just need to update your R markdown document and these changes will automatically be included in your final document.

8.3   Get started with R markdown

To use R markdown you will first need to install the `rmarkdown` package in RStudio (or in the R console if you're not using RStudio) and any package dependencies. You can find instructions on how to do this for both Windows and Mac OSX operating systems here. If you would like to create pdf documents (or MS Word documents) from your R markdown file you will also need to install a version of LaTeX on your computer. If you've not installed LaTeX before, we recommend that you install TinyTeX. Again, instructions on how to do this can be found here.

8.4   Create an R markdown document

Right, time to create your first R markdown document. Within RStudio, click on the menu `File -> New File -> R Markdown...`. In the pop up window, give the document a 'Title' and enter the 'Author' information (your name) and select HTML as the default output. We can change all of this later so don't worry about it for the moment.
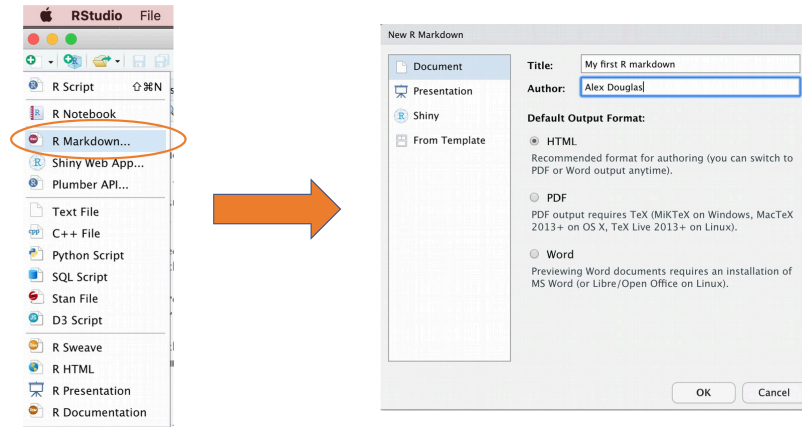
You will notice that when your new R markdown document is created it includes some example R markdown code. Normally you would just highlight and delete everything in the document except the information at the top between the --- delimiters (this is called the YAML header which we will discuss in a bit) and then start writing your own code. However, just for now we will use this document to practice converting R markdown to both html and pdf formats and check everything is working.

Once you've created your R markdown document it's good practice to save this file somewhere convenient. You can do this by selecting `File -> Save` from RStudio menu

(or use the keyboard shortcut ctrl + s on Windows or cmd + s on a Mac) and enter an appropriate file name (maybe call it `my_first_rmarkdown`). Notice the file extension of your new R markdown file is `.Rmd`.

Now, to convert your `.Rmd` file to a HTML document click on the little black triangle next to the `Knit` icon at the top of the source window and select `knit to HTML`.
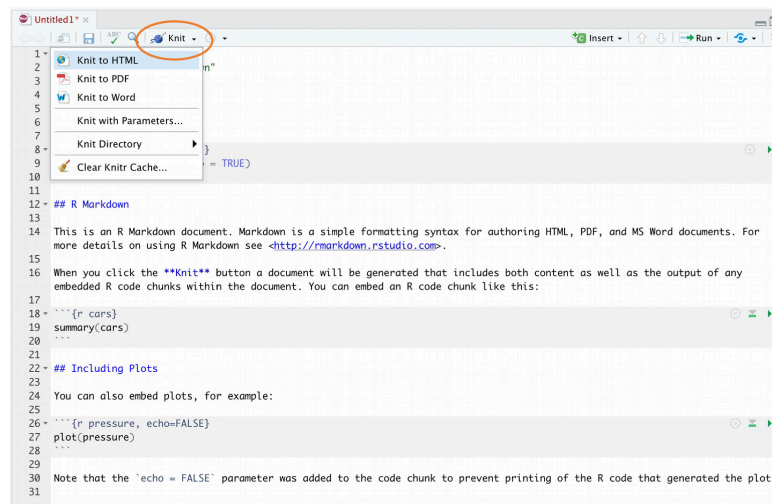
RStudio will now 'knit' (or render) your `.Rmd` file into a HTML file. Notice that there is a new `R Markdown` tab in your console window which provides you with information on the rendering process and will also display any errors if something goes wrong.

If everything went smoothly a new HTML file will have been created and saved in the same directory as your `.Rmd` file (ours will be called `my_first_rmarkdown.html`). To view this document simply double click on the file to open in a browser (like Chrome or Firefox) to display the rendered content. RStudio will also display a preview of the rendered file in a new window for you to check out (your window might look slightly different if you're using a Windows computer).
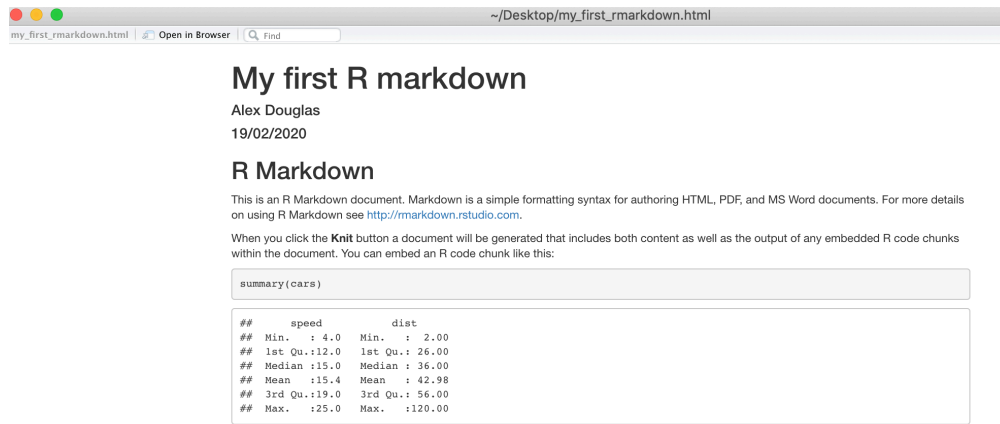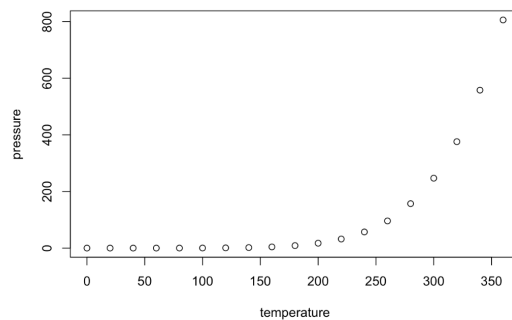
Great, you've just rendered your first R markdown document. If you want to knit your `.Rmd` file to a pdf document then all you need to do is choose `knit to PDF` instead of `knit to HTML` when you click on the `knit` icon. This will create a file called `my_first_rmarkdown.pdf` which you can double click to open. Give it a go!

You can also knit an `.Rmd` file using the command line in the console rather than by clicking on the knit icon. To do this, just use the `render()` function from the `rmarkdown` package as shown below. Again, you can change the output format using the `output_format=` argument as well as many other options.

```
library(rmarkdown)

render('my_first_rmarkdown.Rmd', output_format = 'html_document')

# alternatively if you don't want to load the rmarkdown package

rmarkdown::render('my_first_rmarkdown.Rmd', output_format = 'html_document')

# see ?render for more options
```

8.5   R markdown anatomy

OK, now that you can render an R markdown file in RStudio into both HTML and pdf formats let's take a closer look at the different components of a typical R markdown document. Normally each R markdown document is composed of 3 main components: 1) a YAML header, 2) formatted text and 3) one or more code chunks.
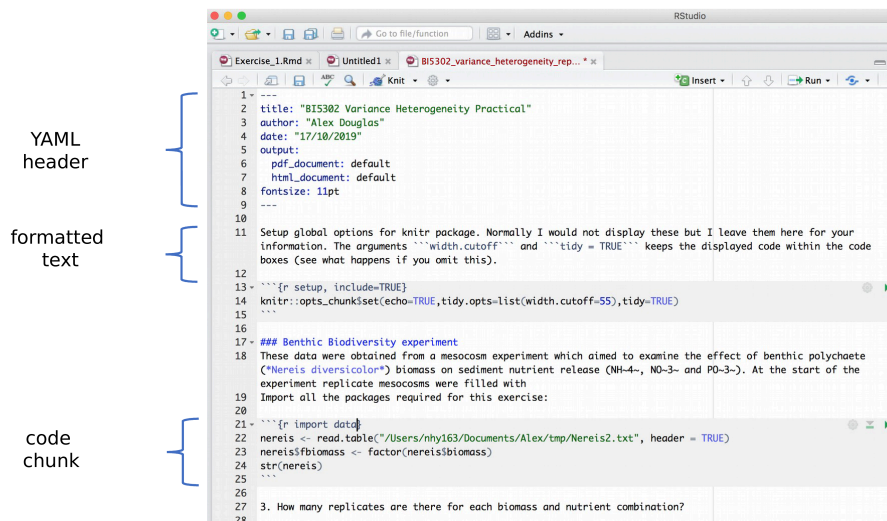
8.5.1   YAML header

YAML stands for '**Y**AML **A**in't **M**arkup **L**anguage' (it's an 'in' joke!) and this optional component contains the metadata and options for the entire document such as the author name, date, output format, etc. The YAML header is surrounded before and after by a --- on its own line. In RStudio a minimal YAML header is automatically created for you when you create a new R markdown document as we did above but you can change this any time. A simple YAML header may look something like this:

```
---
title: My first R markdown document
author: Jane Doe
date: March 01, 2020
output: html_document
---
```

In the YAML header above the output format is set to HTML. If you would like to change the output to pdf format then you can change it from `output: html_document` to

output: `pdf_document` (you can also set more than one output format if you like). You can also change the default font and font size for the whole document and even include fancy options such as a table of contents and inline references and a bibliography. If you want to explore the plethora of other options see here. Just a note of caution, many of the options you can specify in the YAML header will work with both HTML and pdf formatted documents, but not all. If you need multiple output formats for your R markdown document check whether your YAML options are compatible between these formats. Also, indentation in the YAML header has a meaning, so be careful when aligning text. For example, if you want to include a table of contents you would modify the `output:` field in the YAML header as follows

```
---
title: My first R markdown document
author: Jane Doe
date: March 01, 2020
output:
  html_document:
    toc: yes
---
```

8.5.2   Formatted text

As mentioned above, one of the great things about R markdown is that you don't need to rely on your word processor to bring your R code, analysis and writing together. R markdown is able to render (almost) all of the text formatting that you are likely to need such as italics, bold, strike-through, super and subscript as well as bulleted and numbered lists, headers and footers, images, links to other documents or web pages and also equations. However, in contrast to your familiar *What-You-See-Is-What-You-Get* (WYSIWYG) word processing software you don't see the final formatted text in your R markdown document (as you would in MS Word), rather you need to 'markup' the formatting in your text ready to be rendered in your output document. At first, this might seem like a right pain in the proverbial but it's actually very easy to do and also has many advantages (do you find yourself spending more time on making your text look pretty in MS Word rather than writing good content?!).

Here is an example of marking up text formatting in an R markdown document

```
#### Benthic Biodiversity experiment
These data were obtained from a mesocosm experiment which aimed to examine the effect
of benthic polychaete (*Nereis diversicolor*) biomass on sediment nutrient
(NH~4~, NO~3~ and PO~3~) release. At the start of the experiment 15 replicate mesocosms
were filled with 20 cm^2^ of **homogenised** marine sediment and assigned to one of five
polychaete biomass treatments (0, 0.5, 1, 1.5, 2 g per mesocosm).
```

which would look like this in the final rendered document (can you spot the markups?)

Benthic Biodiversity experiment

These data were obtained from a mesocosm experiment which aimed to examine the effect of benthic polychaete (*Nereis diversicolor*) biomass on sediment nutrient (NH_4, NO_3 and PO_3) release. At the start of the experiment replicate mesocosms were filled with 20 cm^2 of **homogenised** marine sediment and assigned to one of five polychaete biomass treatments (0, 0.5, 1, 1.5, 2 g per mesocosm).

Emphasis

Some of the most common R markdown syntax for providing emphasis and formatting text is given below.

Goal	R markdown	output
bold text	`**mytext**`	**mytext**
italic text	`*mytext*`	*mytext*
strikethrough	`~~mytext~~`	~~mytext~~
superscript	`mytext^2^`	mytext^2
subscript	`mytext~2~`	mytext$_2$

Interestingly there is no underline R markdown syntax by default. We think this is because bold and italics are used to emphasise content (a semantic meaning) whereas an underline is considered a stylistic element (there may well be other reasons). If you really want to underline text you can use `<span style="text-decoration:underline">underline this text</span>` for HTML output or `$\text{\underline{This sentence underlined using \LaTeX}}$` for pdf output. We just avoid underlining!

White space and line breaks

One of the things that can be confusing for new users of R markdown is the use of spaces and carriage returns (the enter key on your keyboard). In R markdown multiple spaces within the text are generally ignored as are carriage returns. For example this R markdown text

```
These      data were      obtained from a
mesocosm experiment which    aimed to examine the
effect
of        benthic polychaete (*Nereis diversicolor*) biomass.
```

will be rendered as

> These data were obtained from a mesocosm experiment which aimed to
> examine the effect of benthic polychaete (*Nereis diversicolor*) biomass.

This is generally a good thing (no more random multiple spaces in your text). If you want your text to start on a new line then you can simply add two blank spaces at the end of the preceding line

> These data were obtained from a
> mesocosm experiment which aimed to examine the
> effect benthic polychaete (*Nereis diversicolor*) biomass.

If you really want multiple spaces within your text then you can use the **N**on **b**reaking **sp**ace tag ` `

```
These       data were         obtained from a
mesocosm experiment which     aimed to examine the
effect         benthic polychaete (*Nereis diversicolor*) biomass.
```

> These   data were   obtained from a
> mesocosm experiment which   aimed to examine the
> effect   benthic polychaete (*Nereis diversicolor*) biomass.

Headings

You can add headings and subheadings to your R markdown document by using the `#` symbol at the beginning of the line. You can decrease the size of the headings by simply adding more `#` symbols. For example

```
# Benthic Biodiversity experiment
## Benthic Biodiversity experiment
### Benthic Biodiversity experiment
#### Benthic Biodiversity experiment
##### Benthic Biodiversity experiment
###### Benthic Biodiversity experiment
```

results in headings in decreasing size order

Benthic Biodiversity experiment

Benthic Biodiversity experiment

Benthic Biodiversity experiment

Benthic Biodiversity experiment

Benthic Biodiversity experiment

Comments

As you can see above the meaning of the `#` symbol is different when formatting text in an R markdown document compared to a standard R script (which is used to included a comment - remember?!). You can, however, use a `#` symbol to comment code inside a code chunk as usual (more about this in a bit). If you want to include a comment in your R markdown document outside a code chunk which won't be included in the final rendered document then enclose your comment between `<!--` and `-->`.

```
<!--
this is an example of how to format a comment using R markdown.
-->
```

Lists

If you want to create a bullet point list of text you can format an unordered list with sub items. Notice that the sub-items need to be indented.

```
- item 1
- item 2
    + sub-item 2
    + sub-item 3
- item 3
- item 4
```

- item 1
- item 2

  - sub-item 2
  - sub-item 3
- item 3
- item 4

If you need an ordered list

```
1. item 1
2. item 2
    + sub-item 2
    + sub-item 3
3. item 3
4. item 4
```

1. item 1
2. item 2

   - sub-item 2
   - sub-item 3
3. item 3
4. item 4

Images

Another useful feature is the ability to embed images and links to web pages (or other documents) into your R markdown document. You can include images into your R markdown document in a number of different ways. Perhaps the simplest method is to use

```
![Cute grey kitten](images/Cute_grey_kitten.jpg)
```

resulting in:

The code above will only work if the image file (`Cute_grey_kitten.jpg`) is in the right place relative to where you saved your `.Rmd` file. In the example above the image file is in a sub directory (folder) called `images` in the directory where we saved our `my_first_rmarkdown.Rmd` file. You can embed images saved in many different file types but perhaps the most common are `.jpg` and `.png`.

We think a more flexible way of including images in your document is to use the `include_graphics()` function from the `knitr` package as this gives finer control over the alignment and image size (it also works more or less the same with both HTML and pdf output formats). However, to do this you will need to include this R code in a 'code chunk' which we haven't covered yet. Despite this we'll leave the code here for later reference. This code center aligns the image and scales it to 50% of it's original size. See `?include_graphics` for more options.

```
```{r, echo=FALSE, fig.align='center', out.width='50%'}
library(knitr)
include_graphics("images/Cute_grey_kitten.jpg")
```
```

Figure 8.1: Cute grey kitten

Links

In addition to images you can also include links to webpages or other links in your document. Use the following syntax to create a clickable link to an existing webpage. The link text goes between the square brackets and the URL for the webpage between the round brackets immediately after.

```
You can include a text for your clickable [link](https://www.worldwildlife.org)
```

which gives you:

You can include a text for your clickable link

8.5.3   Code chunks

Now to the heart of the matter. To include R code into your R markdown document you simply place your code into a 'code chunk'. All code chunks start and end with three backticks ```. Note, these are also known as 'grave accents' or 'back quotes' and are not the same as an apostrophe! On most keyboards you can find the backtick on the same key as tilde (~).

```{r}
Any valid R code goes here
```

You can insert a code chunk by either typing the chunk delimiters ```{r} and ``` manually or use the RStudio toolbar (the Insert button) or by clicking on the menu `Code -> Insert Chunk`. Perhaps an even better way is to get familiar with the keyboard shortcuts Ctrl + Alt + I for Windows and Cmd + Option + I on MacOSX.

There are a many things you can do with code chunks: you can produce text output from your analysis, create tables and figures and insert images amongst other things. Within the code chunk you can place rules and arguments between the curly brackets `{}` that give you control over how your code is interpreted and output is rendered. These are known as chunk options. The only mandatory chunk option is the first argument which specifies which language you're using (`r` in our case but other languages are supported). Note, all of your chunk options must be written between the curly brackets on one line with no line breaks.

You can also specify an optional code chunk name (or label) which can be useful when trying to debug problems and when performing advanced document rendering. In the following block we name the code chunk `summary-stats`, create a dataframe (`dataf`) with two variables `x` and `y` and then use the `summary()` function to display some summary statistics . When we run the code chunk both the R code and the resulting output are displayed in the final document.

```
```{r, summary-stats}
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
dataf <- data.frame(x = x, y = y)

summary(dataf)
```
```

```
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
dataf <- data.frame(x = x, y = y)

summary(dataf)
##       x              y
##  Min.   : 1.00   Min.   : 1.00
##  1st Qu.: 3.25   1st Qu.: 3.25
##  Median : 5.50   Median : 5.50
##  Mean   : 5.50   Mean   : 5.50
##  3rd Qu.: 7.75   3rd Qu.: 7.75
##  Max.   :10.00   Max.   :10.00
```

When using chunk names make sure that you don't have duplicate chunk names in your R markdown document and avoid spaces and full stops as this may cause problems when you come to knit your document (We use a - to separate words in our chunk names).

If we wanted to only display the output of our R code (just the summary statistics for example) and not the code itself in our final document we can use the chunk option `echo=FALSE`

```
```{r, summary-stats, echo=FALSE}
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
dataf <- data.frame(x = x, y = y)
summary(dataf)
```
```

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

To display the R code but not the output use the `results='hide'` chunk option.

```
```{r, summary-stats, results='hide'}
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
dataf <- data.frame(x = x, y = y)
summary(dataf)
```
```

```
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
dataf <- data.frame(x = x, y = y)
summary(dataf)
```

Sometimes you may want to execute a code chunk without showing any output at all. You can suppress the entire output using the chunk option `include=FALSE`.

```
```{r, summary-stats, include=FALSE}
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
dataf <- data.frame(x = x, y = y)
summary(dataf)
```
```

There are a large number of chunk options documented here with a more condensed version here. Perhaps the most commonly used are summarised below with the default values shown.

Chunk option	default value	Function
echo	echo=TRUE	If FALSE, will not display the code in the final document
results	results='markup'	If 'hide', will not display the code's results in the final document. If 'hold', will delay displaying all output pieces until the end of the chunk. If 'asis', will pass through results without reformatting them.
include	include=TRUE	If FALSE, will run the chunk but not include the chunk in the final document.
eval	eval=TRUE	If FALSE, will not run the code in the code chunk.
message	message=TRUE	If FALSE, will not display any messages generated by the code.
warning	warning=TRUE	If FALSE, will not display any warning messages generated by the code.

8.5.4   Adding figures

By default, figures produced by R code will be placed immediately after the code chunk they were generated from. For example:

```{r, simple-plot}
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
dataf <- data.frame(x = x, y = y)
plot(dataf$x, dataf$y, xlab = "x axis", ylab = "y axis")
```

```r
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
dataf <- data.frame(x = x, y = y)
plot(dataf$x, dataf$y, xlab = "x axis", ylab = "y axis")
```
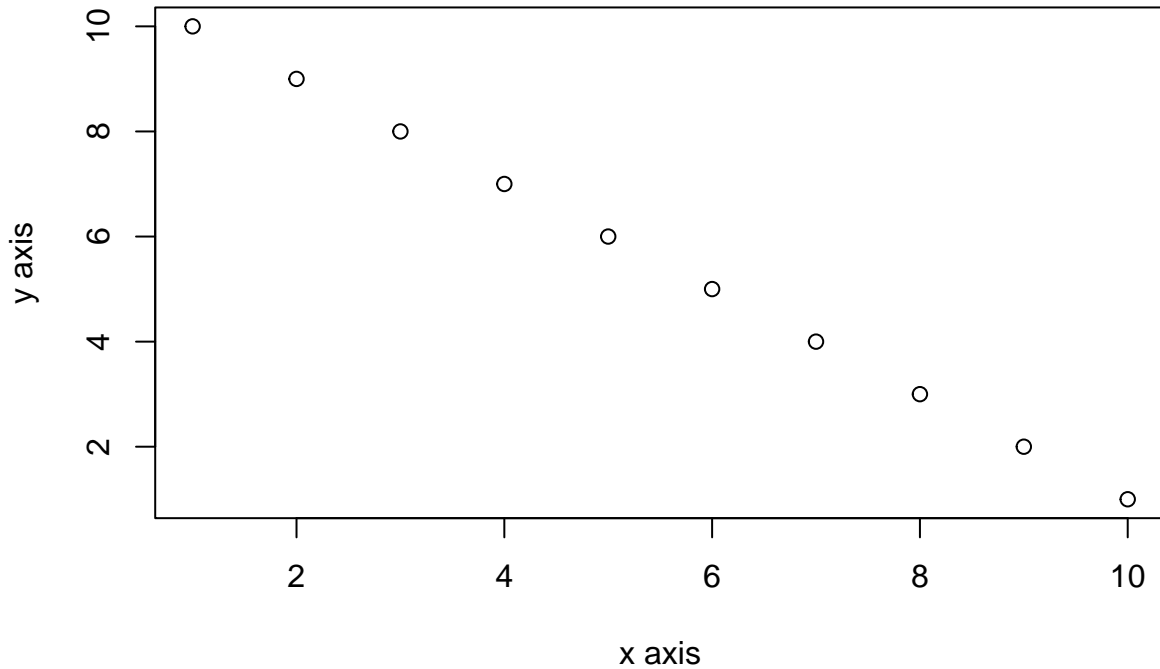

If you want to change the plot dimensions in the final document you can use the `fig.width=` and `fig.height=` chunk options (in inches!). You can also change the alignment of the figure using the `fig.align=` chunk option.

````r
```{r, simple-plot, fig.width=4, fig.height=3, fig.align='center'}
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
dataf <- data.frame(x = x, y = y)
plot(dataf$x, dataf$y, xlab = "x axis", ylab = "y axis")
```
````

```r
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
dataf <- data.frame(x = x, y = y)
plot(dataf$x, dataf$y, xlab = "x axis", ylab = "y axis")
```
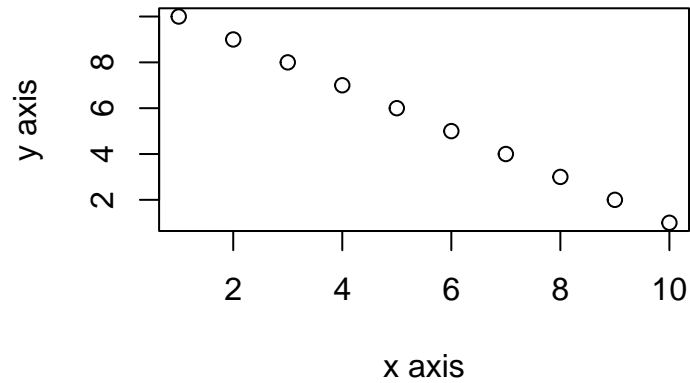
You can add a figure caption using the `fig.cap=` option.

```
```{r, simple-plot-cap, fig.cap="A simple plot", fig.align='center'}
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
dataf <- data.frame(x = x, y = y)
plot(dataf$x, dataf$y, xlab = "x axis", ylab = "y axis")
```
```

```r
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
dataf <- data.frame(x = x, y = y)
plot(dataf$x, dataf$y, xlab = "x axis", ylab = "y axis")
```

If you want to suppress the figure in the final document use the `fig.show='hide'` option.

```
```{r, simple-plot, fig.show='hide'}
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
dataf <- data.frame(x = x, y = y)
plot(dataf$x, dataf$y, xlab = "x axis", ylab = "y axis")
```
```

Figure 8.2: A simple plot

```r
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
dataf <- data.frame(x = x, y = y)
plot(dataf$x, dataf$y, xlab = "x axis", ylab = "y axis")
```
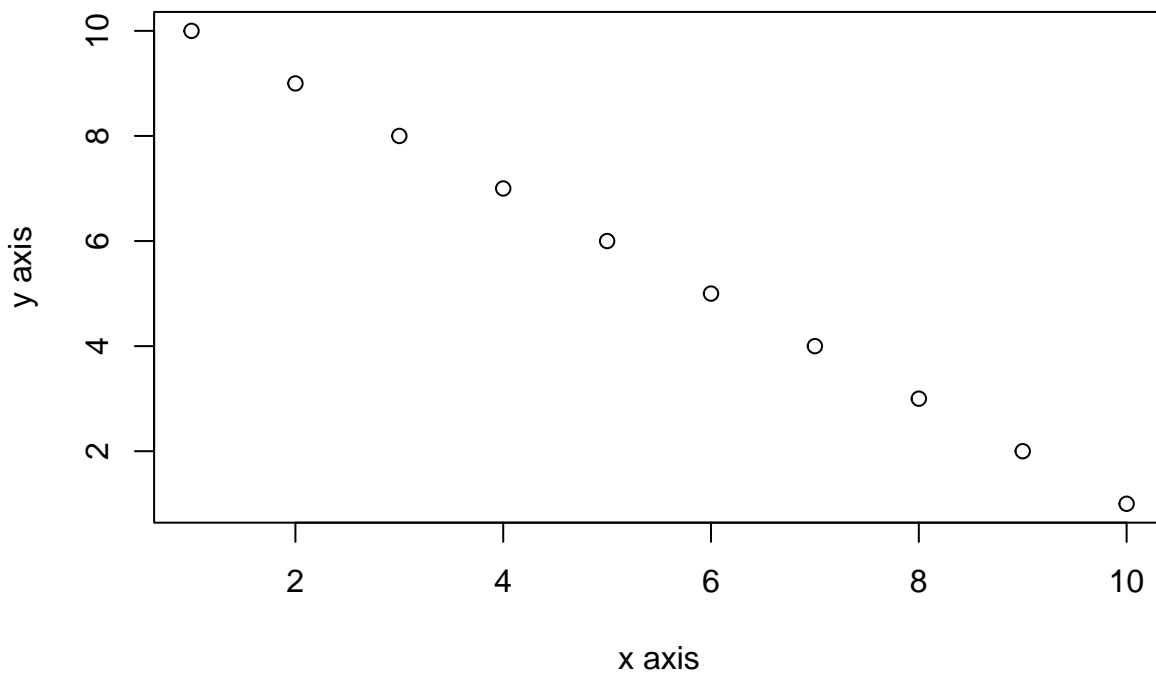
If you're using a package like `ggplot2` to create your plots then don't forget you will need to make the package available with the `library()` function in the code chunk (or in a preceding code chunk).

```r
```{r, simple-ggplot}
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
dataf <- data.frame(x = x, y = y)

library(ggplot2)
ggplot(dataf, aes(x = x, y = y)) +
  geom_point()
```
```

```r
x <- 1:10     # create an x variable
y <- 10:1     # create a y variable
dataf <- data.frame(x = x, y = y)

library(ggplot2)
ggplot(dataf, aes(x = x, y = y)) +
  geom_point()
```
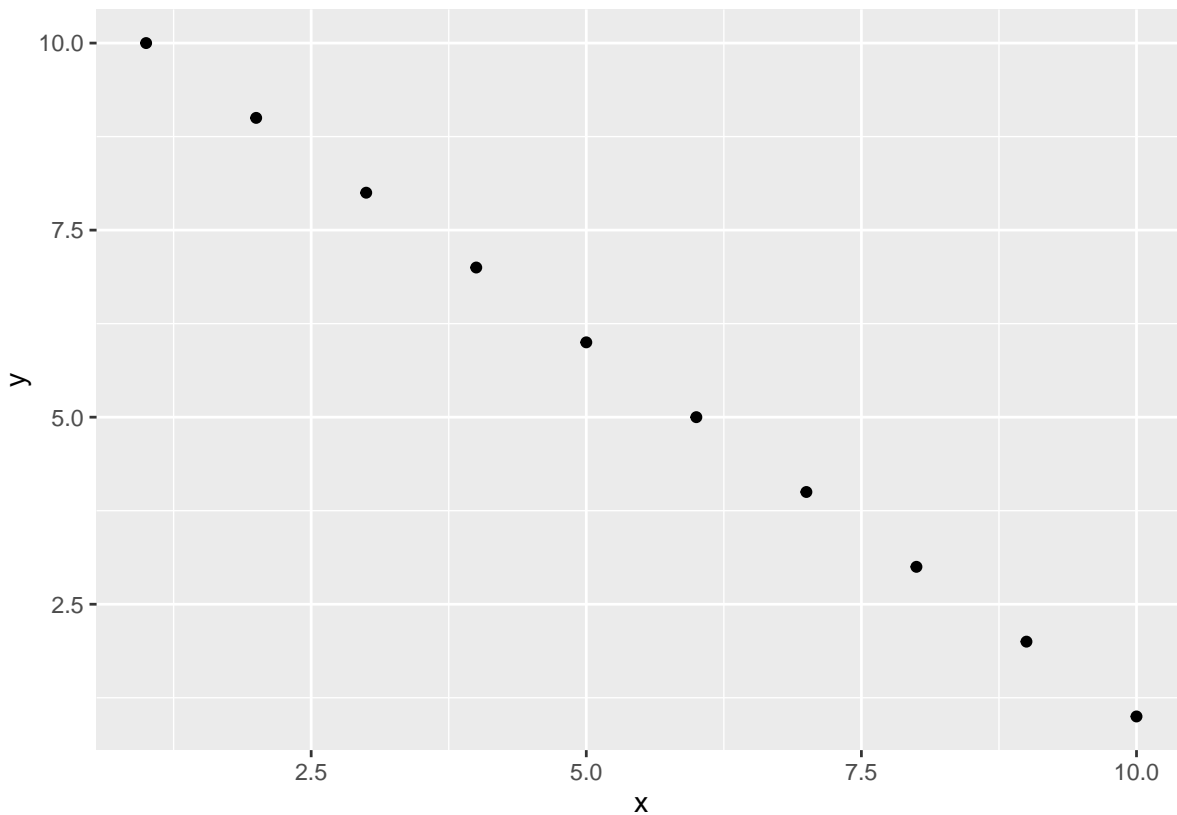

Again, there are a large number of chunk options specific to producing plots and figures.
See here for more details.

8.5.5   Adding tables

R markdown can print the contents of a dataframe as a table (or any other tabular
object such as a summary of model output) by including the name of the dataframe in
a code chunk. For example, to create a table of the first 10 rows of the inbuilt dataset
iris

```
```{r, ugly-table}
iris[1:10,]
```
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          1.4         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 4           4.6         3.1          1.5         0.2  setosa
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
```

But how ugly is that! You can create slightly nicer looking tables using native markdown syntax (this doesn't need to be in a code chunk).

```
|     x     |     y     |
|:---------:|:---------:|
|    1      |    5      |
|    2      |    4      |
|    3      |    3      |
|    4      |    2      |
|    5      |    1      |
```

x	y
1	5
2	4
3	3

x	y
4	2
5	1

The `:-------:` lets R markdown know that the line above should be treated as a header and the lines below as the body of the table. Alignment within the table is set by the position of the `:`. To center align use `:------:`, to left align `:------` and right align `------:`. Whilst it can be fun(!) to create tables with raw markup it's only practical for very small and simple tables.

The easiest way we know to include tables in an R markdown document is by using the `kable()` function from the `knitr` package (this should have already been installed when you installed the `rmarkdown` package). The `kable()` function can create tables for HTML, PDF and Word outputs.

To create a table of the first 10 rows of the `iris` dataframe using the `kable()` function simply write your code block as

```{r, kable-table}
library(knitr)
kable(iris[1:10,])
```

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
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa

The `kable()` function offers plenty of options to change the formatting of the table. For example, if we want to round numeric values to one decimal place use the `digits =` argument. To center justify the table contents use `align = 'c'` and to provide custom column headings use the `col.names =` argument. See `?knitr::kable` for more information.

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
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa

```{r, kable-table2}
kable(iris[1:10,], digits = 0, align = 'c',
      col.names = c('sepal length', 'sepal width',
      'petal length', 'petal width', 'species'))
```

sepal length	sepal width	petal length	petal width	species
5	4	1	0	setosa
5	3	1	0	setosa
5	3	1	0	setosa
5	3	2	0	setosa
5	4	1	0	setosa
5	4	2	0	setosa
5	3	1	0	setosa
5	3	2	0	setosa
4	3	1	0	setosa
5	3	2	0	setosa

You can further enhance the look of your `kable` tables using the `kableExtra` package (don't forget to install the package first!). See here for more details and a helpful tutorial.

```{r, kableExtra-table}
library(kableExtra)
kable(iris[1:10,]) %>%
    kable_styling(bootstrap_options = "striped", font_size = 10)
```

If you want even more control and customisation options for your tables take a look at the `pander` and `xtable` packages.

8.5.6   Inline R code

Up till now we've been writing and executing our R code in code chunks. Another great reason to use R markdown is that we can also include our R code directly within our text. This is known as 'inline code'. To include your code in your R markdown text you simply write `` `r write your code here` ``. This can come in really useful when you want to include summary statistics within your text. For example, we could describe the `iris` dataset as follows:

```
Morphological characteristics (variable names:  `r names(iris)[1:4]`) were measured from
`r nrow(iris)` *Iris sp.* plants from  `r length(levels(iris$Species))` different
species. The mean Sepal length was `r round(mean(iris$Sepal.Length), digits = 2)` mm.
```

which will be rendered as

> Morphological characteristics (variable names: Sepal.Length, Sepal.Width, Petal.Length, Petal.Width) were measured from 150 *iris* plants from 3 different species. The mean Sepal length was 5.84 mm.

The great thing about including inline R code in your text is that these values will automatically be updated if your data changes.

8.6   Some tips and tricks

Problem :

When rendering my R markdown document to pdf my code runs off the edge of the page.

Solution:

Add a global_options argument at the start of your .Rmd file in a code chunk:

```
```{r, global_options, include=FALSE}
knitr::opts_chunk$set(message=FALSE, tidy.opts=list(width.cutoff=60), tidy=TRUE)
```
```

This code chunk won't be displayed in the final document due to the `include = FALSE` argument and you should place the code chunk immediately after the YAML header to affect everything below that.

`tidy.opts = list(width.cutoff = 60), tidy=TRUE` defines the margin cutoff point and wraps text to the next line. Play around with this value to get it right (60-80 should be OK for most documents).

Problem:

When I load a package in my R markdown document my rendered output contains all of the startup messages and/or warnings.

Solution:

You can load all of your packages at the start of your R markdown document in a code chunk along with setting your global options.

```
```{r, global_options, include=FALSE}
knitr::opts_chunk$set(message=FALSE, warning=FALSE, tidy.opts=list(width.cutoff=60))
suppressPackageStartupMessages(library(ggplot2))
```
```

The `message=FALSE` and `warning=FALSE` arguments suppress messages and warnings. The `suppressPackageStartupMessages(library(ggplot2))` will load the `ggplot2` package but suppress startup messages.

Problem:

When rendering my R markdown document to pdf my tables and/or figures are split over two pages.

Solution:

Add a page break using the LateX `\pagebreak` notation before your offending table or figure.

Problem:

The code in my rendered document looks ugly!

Solution:

Add the argument `tidy=TRUE` to your global arguments. Sometimes, however, this can cause problems especially with correct code indentation.

```{r, global_options, include=FALSE}
knitr::opts_chunk$set(message=FALSE, tidy.opts=list(width.cutoff=60), tidy=TRUE)
```

8.7   Further Information

Although we've covered more than enough to get you quite far using R markdown, as with most things R related, we've really only had time to scratch the surface. Happily, there's a wealth of information available to you should you need to expand your knowledge and experience. A good place to start is the excellent free book written by the creator of R markdown *Yihui Xie* which you can find here.

Another useful and concise R markdown reference guide can be found here.

A quick and easy RStudio R markdown cheatsheet.

Chapter 9

Version control with Git and GitHub

This Chapter will introduce you to the basics of using a version control system to keep track of all your important R code and facilitate collaboration with colleagues and the wider world. This Chapter will focus on using the software 'Git' in combination with the web-based hosting service 'GitHub'. By the end of the Chapter, you will be able to install and configure Git and GitHub on your computer and setup and work with a version controlled project in RStudio. We won't be covering more advanced topics such as branching, forking and pull requests in much detail but we do give an overview later on in the Chapter.
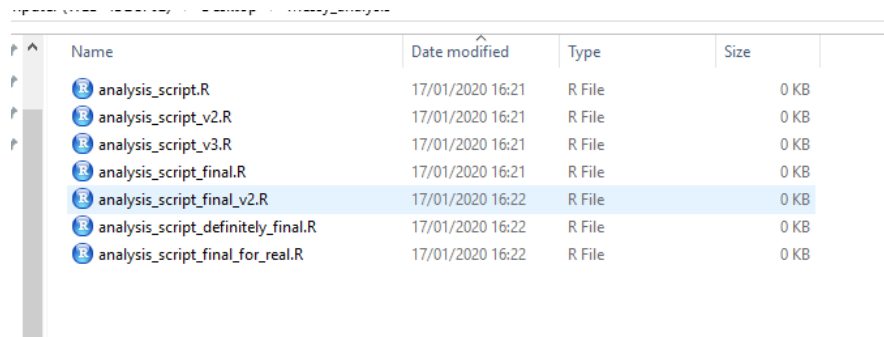
Just a few notes of caution. In this Chapter we'll be using RStudio to interface with Git as it gives you a nice friendly graphical user interface which generally makes life a little bit easier (and who doesn't want that?). However, one downside to using RStudio with Git is that RStudio only provides pretty basic Git functionality through its menu system. That's fine for most of what we'll be doing during this Chapter (although we will introduce a few Git commands as we go along) but if you really want to benefit from using Git's power you will need to learn some Git commands and syntax. This leads us on to our next point. We're not going to lie, Git can become a little bewildering and frustrating when you first start using it. This is mostly due to the terminology and liberal use of jargon associated with Git, but there's no hiding the fact that it's quite easy to get yourself and your Git repository into a pickle. Therefore, we've tried hard to keep things as straight forward as we can during this Chapter and as a result we do occasionally show you a couple of very 'un-Git' ways of doing things (mostly about reverting to previous versions of documents). Don't get hung up about this, there's no shame to using these low tech solutions and if it works then it works. Lastly, GitHub was not designed to host very large files and will warn you if you try to add files greater than 50 MB and block you adding files greater than 100 MB. If your project involves using large file sizes there are a few solutions but we find the easiest solution is to host these files elsewhere (Googledrive, Dropbox etc) and create a link to them in a README file or R markdown document on Github.

9.1   What is version control?

A Version Control System (VCS) keeps a record of all the changes you make to your files that make up a particular project and allows you to revert to previous versions of files if you need to. To put it another way, if you muck things up or accidentally lose important files you can easily roll back to a previous stage in your project to sort things out. Version control was originally designed for collaborative software development, but it's equally useful for scientific research and collaborations (although admittedly a lot of the terms, jargon and functionality are focused on the software development side). There are many different version control systems currently available, but we'll focus on using *Git*, because it's free and open source and it integrates nicely with RStudio. This means that its can easily become part of your usual workflow with minimal additional overhead.

9.2   Why use version control?

So why should you worry about version control? Well, first of all it helps avoid this (familiar?) situation when you're working on a project

Figure 9.1: You need version control

usually arising from this (familiar?) scenario
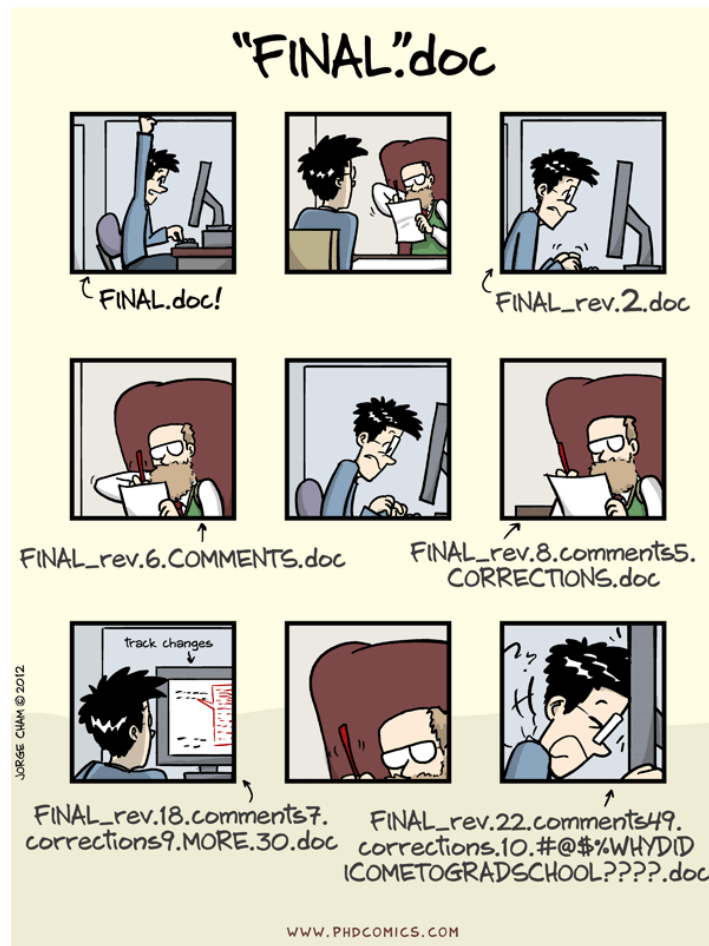
Version control automatically takes care of keeping a record of all the versions of a particular file and allows you to revert back to previous versions if you need to. Version control also helps you (especially the future you) keep track of all your files in a single place and it helps others (especially collaborators) review, contribute to and reuse your work through the GitHub website. Lastly, your files are always available from anywhere and on any computer, all you need is an internet connection.

9.3 What is Git and GitHub?

Git is a version control system originally developed by Linus Torvalds that lets you track changes to a set of files. These files can be any type of file including the menagerie of files that typically make up a data orientated project (.pdf, .Rmd, .docx, .txt, .jpg etc) although plain text files work the best. All the files that make up a project is called a **repository** (or just **repo**).

GitHub is a web-based hosting service for Git repositories which allows you to create a remote copy of your local version-controlled project. This can be used as a backup or archive of your project or make it accessible to you and to your colleagues so you can work collaboratively.

At the start of a project we typically (but not always) create a **remote** repository on GitHub, then **clone** (think of this as copying) this repository to our **local** computer (the one in front of you). This cloning is usually a one time event and you shouldn't need to clone this repository again unless you really muck things up. Once you have cloned your repository you can then work locally on your project as usual, creating and saving files for your data analysis (scripts, R markdown documents, figures etc). Along the way you can take snapshots (called **commits**) of these files after you've made important changes. We can then **push** these changes to the remote GitHub repository to make a backup or make available to our collaborators. If other people are working on the same project (**repository**), or maybe you're working on a different computer, you can **pull** any changes back to your local repository so everything is synchronised.
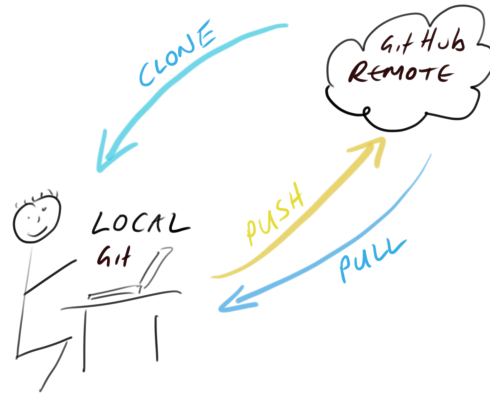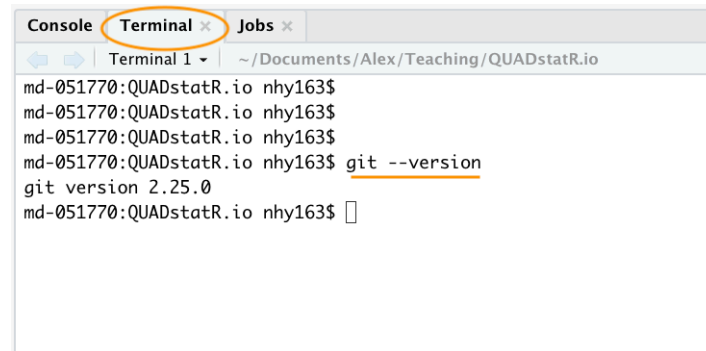
9.4 Getting started

This Chapter assumes that you have already installed the latest versions of R and RStudio. If you haven't done this yet you can find instructions here.

9.4.1 Install Git

To get started, you first need to install Git. If you're lucky you may already have Git installed (especially if you have a Mac or Linux computer). You can check if you already have Git installed by clicking on the Terminal tab in the Console window in RStudio and typing `git --version` (the space after the `git` command is important). If you see something that looks like `git version 2.25.0` (the version number may be different on your computer) then you already have Git installed (happy days). If you get an error (something like `git: command not found`) this means you don't have Git installed (yet!).

You can also do this check outside RStudio by opening up a separate Terminal if you want. On Windows go to the 'Start menu' and in the search bar (or run box) type `cmd` and press enter. On a Mac go to 'Applications' in Finder, click on the 'Utilities' folder and then on the 'Terminal' program. On a Linux machine simply open the Terminal (Ctrl+Alt+T often does it).

To install Git on a **Windows** computer we recommend you download and install Git for Windows (also known as 'Git Bash'). You can find the download file and installation instructions here.

For those of you using a **Mac** computer we recommend you download Git from here and install in the usual way (double click on the installer package once downloaded). If you've previously installed Xcode on your Mac and want to use a more up to date version of Git then you will need to follow a few more steps documented here. If you've never heard of Xcode then don't worry about it!

For those of you lucky enough to be working on a **Linux** machine you can simply use your OS package manager to install Git from the official repository. For Ubuntu Linux (or variants of) open your Terminal and type

```
sudo apt update
sudo apt install git
```

You will need administrative privileges to do this. For other versions of Linux see here for further installation instructions.

Whatever version of Git you're installing, once the installation has finished verify that the installation process has been successful by running the command `git --version` in the Terminal tab in RStudio (as described above). On some installations of Git (yes we're looking at you MS Windows) this may still produce an error as you will also need to setup RStudio so it can find the Git executable (described below).

9.4.2 Configure Git

After installing Git, you need to configure it so you can use it. Click on the Terminal tab in the Console window again and type the following:

```
git config --global user.email 'you@youremail.com'
```

```
git config --global user.name 'Your Name'
```

substituting `'Your Name'` for your actual name and `'you@youremail.com'` with your email address. We recommend you use your University email address (if you have one) as you will also use this address when you register for your GitHub account (coming up in a bit).

If this was successful, you should see no error messages from these commands. To verify that you have successfully configured Git type the following into the Terminal

```
git config --global --list
```

You should see both your `user.name` and `user.email` configured.

9.4.3 Configure RStudio

As you can see above, Git can be used from the command line, but it also integrates well with RStudio, providing a friendly graphical user interface. If you want to use RStudio's Git integration (we recommend you do - at least at the start), you need to check that the path to the Git executable is specified correctly. In RStudio, go to the menu `Tools -> Global Options -> Git/SVN` and make sure that 'Enable version control interface for RStudio projects' is ticked and that the 'Git executable:' path is correct for your installation. If it's not correct hit the `Browse...` button and navigate to where you installed git and click on the executable file. You will need to restart RStudio after doing this.
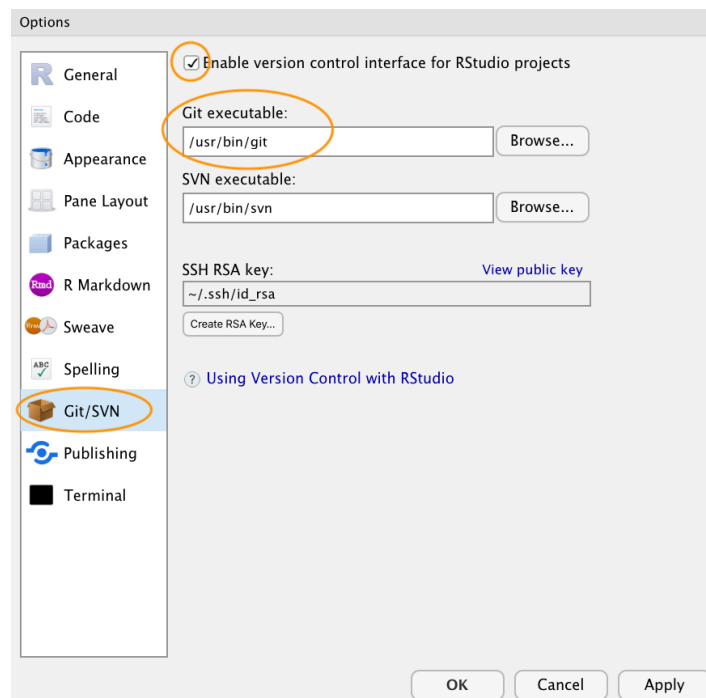
9.4.4   Register a GitHub account

If all you want to do is to keep track of files and file versions on your local computer then Git is sufficient. If however, you would like to make an off-site copy of your project or make it available to your collaborators then you will need a web-based hosting service for your Git repositories. This is where GitHub comes into play (there are also other services like GitLab, Bitbucket and Savannah). You can sign up for a free account on GitHub here. You will need to specify a username, an email address and a strong password. We suggest that you use your University email address (if you have one) as this will also allow you to apply for a free educator or researcher account later on which gives you some useful benefits (don't worry about this now though). When it comes to choosing a username we suggest you give this some thought. Choose a short(ish) rather than a long username, use all lowercase and hyphenate if you want to include multiple words, find a way of incorporating your actual name and lastly, choose a username that you will feel comfortable revealing to your future employer!

Next click on the 'Select a plan' (you may have to solve a simple puzzle first to verify you're human) and choose the 'Free Plan' option. Github will send an email to the email address you supplied for you to verify.

Once you've completed all those steps you should have both Git and GitHub setup up ready for you to use (Finally!).

9.5   Setting up a project in RStudio

Now that you're all set up, let's create your first version controlled RStudio project. There are a couple of different approaches you can use to do this. You can either setup a remote GitHub repository first then connect an RStudio project to this repository (we'll call this Option 1). Another option is to setup a local repository first and then link a remote GitHub repository to this repository (Option 2). You can also connect an existing project to a GitHub repository but we won't cover this here. We suggest that if you're completely new to Git and GitHub then use Option 1 as this approach sets up your local Git repository nicely and you can **push** and **pull** immediately. Option 2 requires a little more work and therefore there are more opportunities to go wrong. We will cover both of these options below.

9.5.1   Option 1 - GitHub first

To use the GitHub first approach you will first need to create a **repository (repo)** on GitHub. Go to your GitHub page and sign in if necessary. Click on the 'Repositories' tab at the top and then on the green 'New' button on the right
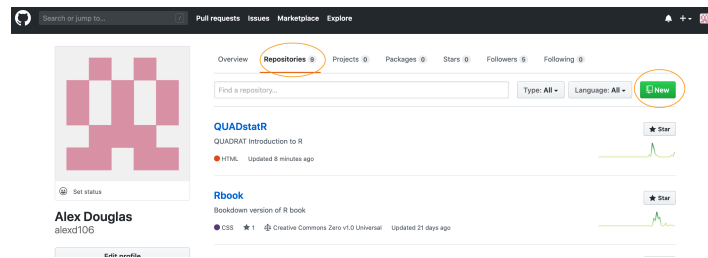
Give your new repo a name (let's call it `first_repo` for this Chapter), select 'Public', tick on the 'Initialize this repository with a README' (this is important) and then click on 'Create repository' (ignore the other options for now).

Your new GitHub repository will now be created. Notice the README has been rendered in GitHub and is in markdown (.md) format (see Chapter 8 on R markdown if this doesn't mean anything to you). Next click on the green 'Clone or Download' button and copy the `https//...` URL that pops up for later (either highlight it all and copy or click on the copy to clipboard icon to the right).
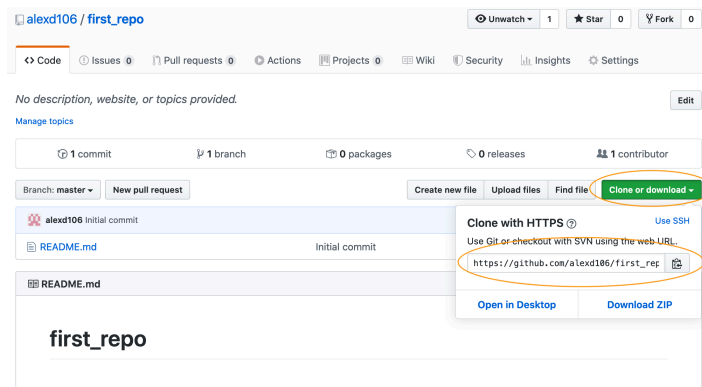
Ok, we now switch our attention to RStudio. In RStudio click on the `File -> New Project` menu. In the pop up window select `Version Control`.
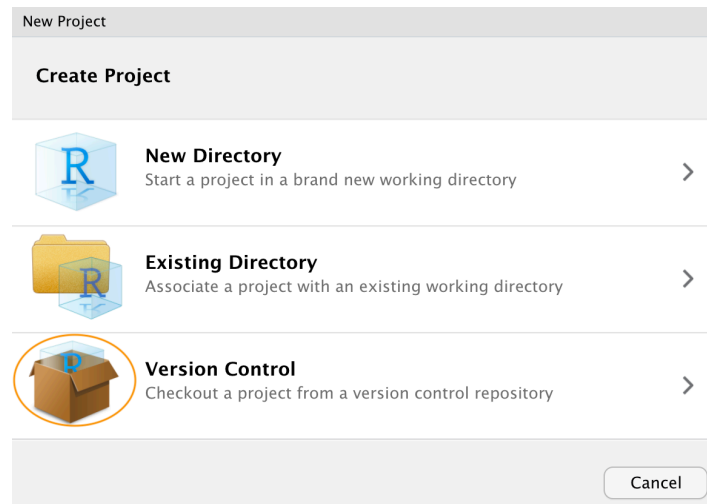
Now paste the the URL you previously copied from GitHub into the `Repository URL:` box. This should then automatically fill out the `Project Directory Name:` section with the correct repository name (it's important that the name of this directory has the same name as the repository you created in GitHub). You can then select where you want to create this directory by clicking on the `Browse` button opposite the `Create project as a subdirectory of:` option. Navigate to where you want the directory created and click OK. We also tick the `Open in new session` option.
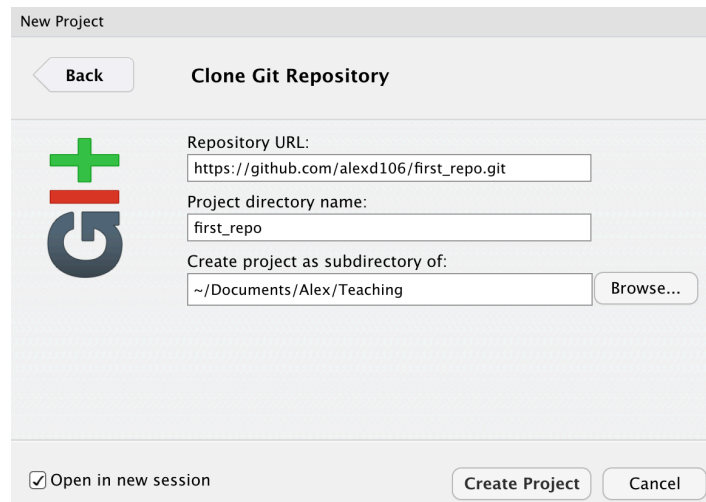
RStudio will now create a new directory with the same name as your repository on your local computer and will then **clone** your remote repository to this directory. The directory will contain three new files; `first_repo.Rproj` (or whatever you called your repository), `README.md` and `.gitignore`. You can check this out in the `Files` tab usually in the bottom right pane in RStudio. You will also have a `Git` tab in the top right pane with two files listed (we will come to this later on in the Chapter). That's it for Option 1, you now have a remote GitHub repository set up and this is linked to your local repository managed by RStudio. Any changes you make to files in this directory will be version controlled by Git.
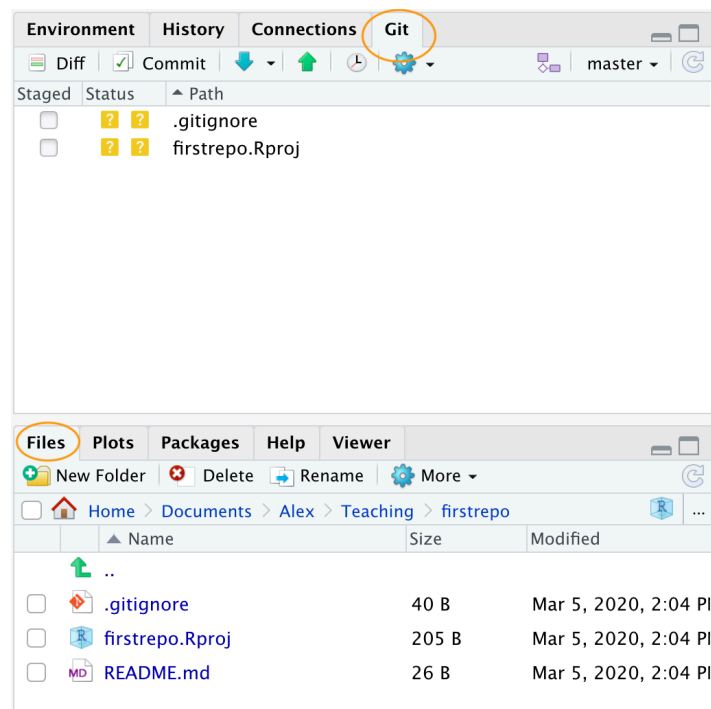
9.5.2 Option 2 - RStudio first

An alternative approach is to create a local RStudio project first and then link to a remote Github repository. As we mentioned before, this option is more involved than Option 1 so feel free to skip this now and come back later to it if you're interested. This option is also useful if you just want to create a local RStudio project linked to a local Git repository (i.e. no GitHub involved). If you want to do this then just follow the instructions below omitting the GitHub bit.

In RStudio click on the `File -> New Project` menu and select the `New Directory` option.
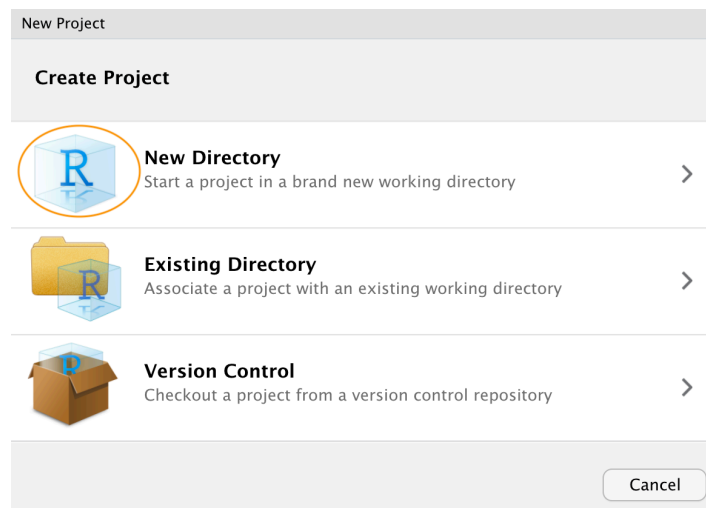
In the pop up window select the `New Project` option
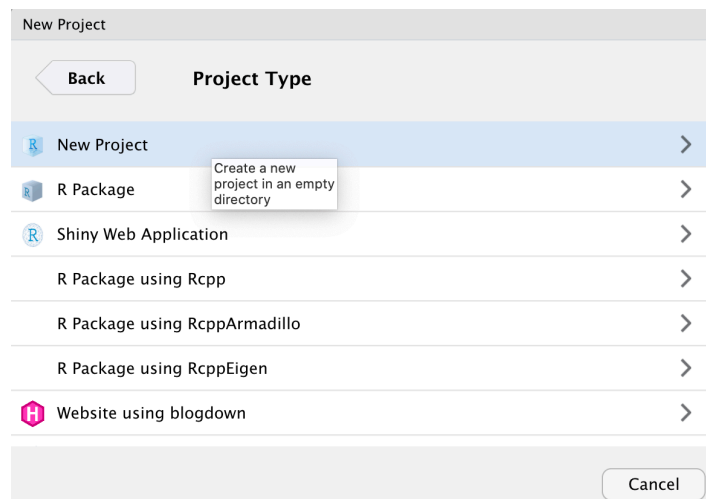
In the New Project window specify a `Directory name` (choose `second_repo` for this Chapter) and select where you would like to create this directory on you computer (click the `Browse` button). Make sure the `Create a git repository` option is ticked.
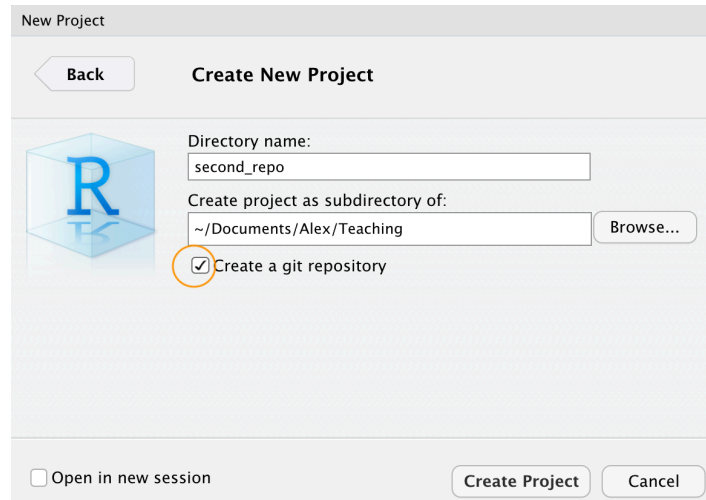
This will create a version controlled directory called `second_repo` on your computer that contains two files, `second_repo.Rproj` and `.gitignore` (there might also be a `.Rhistory` file but ignore this). You can check this by looking in the `Files` tab in RStudio (usually in the bottom right pane).
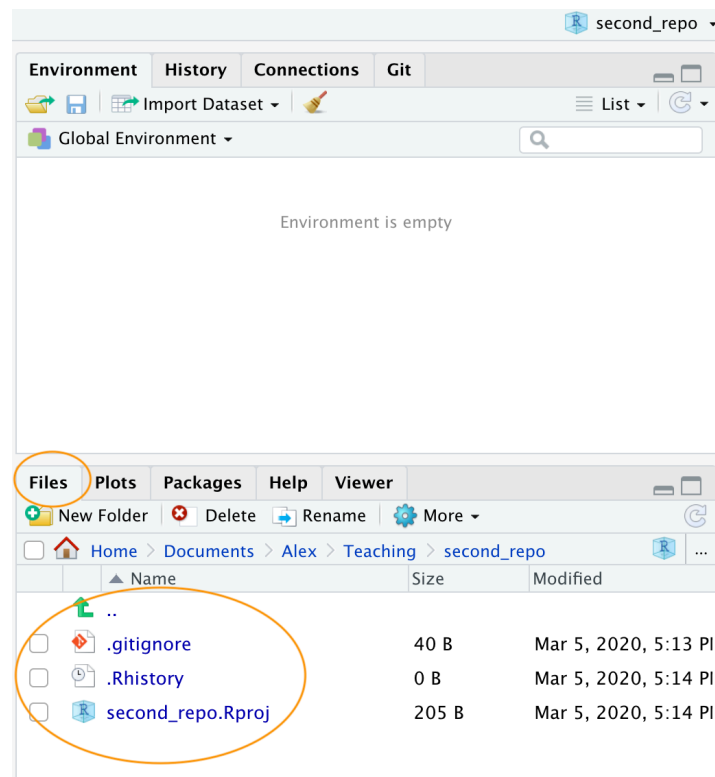
OK, before we go on to create a repository on GitHub we need to do one more thing - we need to place our `second_repo.Rproj` and `.gitignore`files under version control. Unfortunately we haven't covered this in detail yet so just follow the next few instructions (blindly!) and we'll revisit them in the Using Git section of this Chapter.

To get our two files under version control click on the 'Git' tab which is usually in the top tight pane in RStudio.
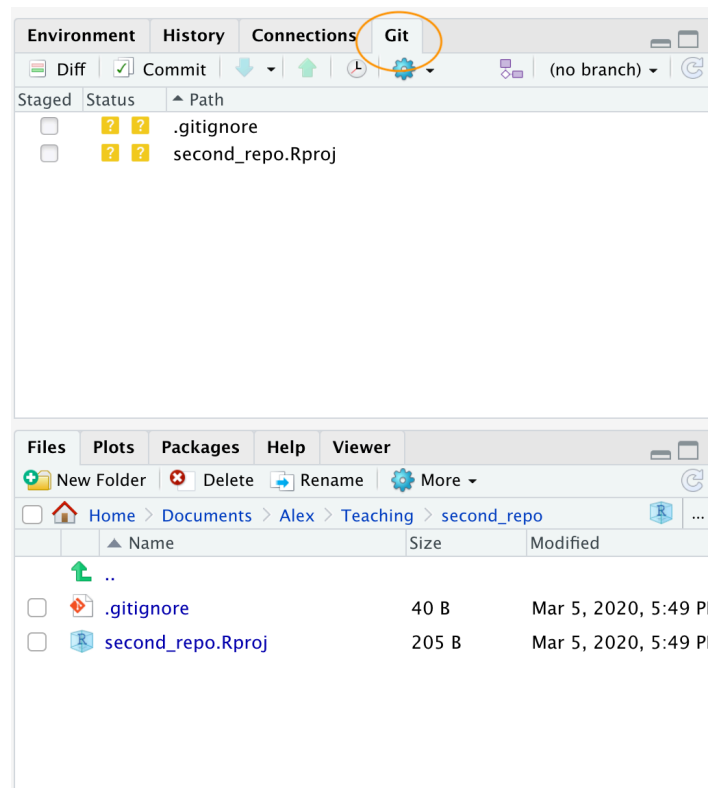
You can see that both files are listed. Next, tick the boxes under the 'Staged' column for both files and then click on the 'Commit' button.
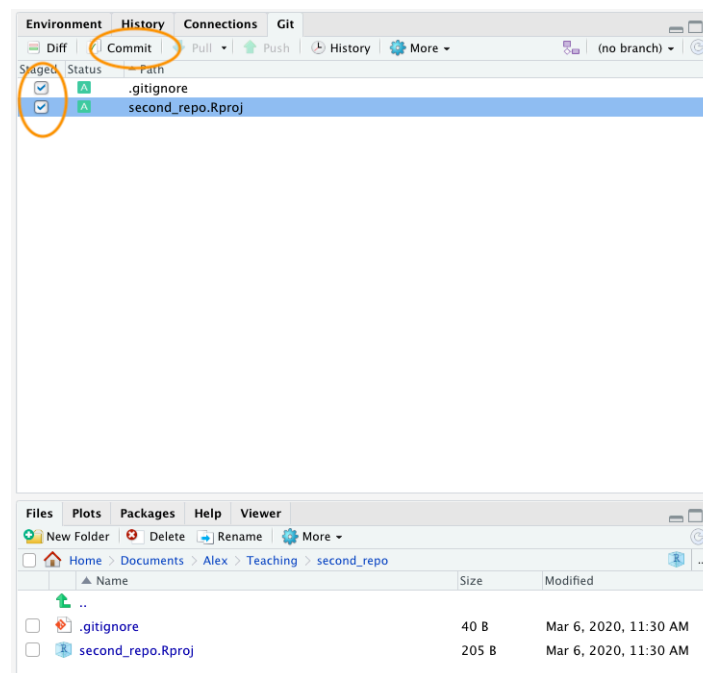
This will take you to the 'Review Changes' window. Type in the commit message 'First commit' in the 'Commit message' window and click on the 'Commit' button. A new window will appear with some messages which you can ignore for now. Click 'Close' to close this window and also close the 'Review Changes' window. The two files should now have disappeared from the Git pane in RStudio indicating a successful commit.
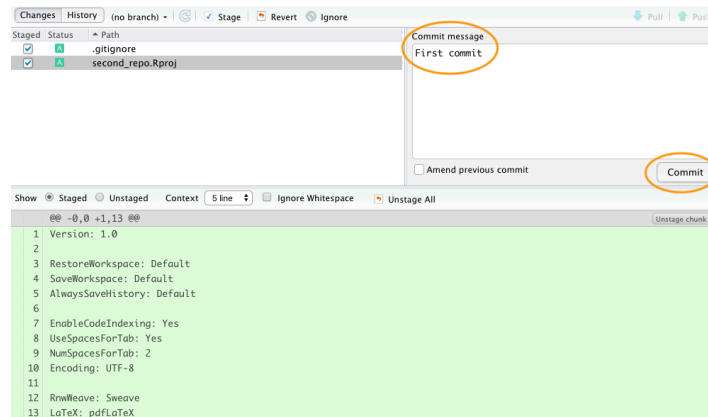
OK, that's those two files now under version control. Now we need to create a new repository on GitHub. In your browser go to your GitHub page and sign in if necessary. Click on the 'Repositories' tab and then click on the green 'New' button on the right. Give your new repo the name second_repo (the same as your version controlled directory name) and select 'Public'. This time **do not** tick the 'Initialize this repository with a README' (this is important) and then click on 'Create repository'.

This will take you to a Quick setup page which provides you with some code for various situations. The code we are interested in is the code under `...or push an existing repository from the command line` heading.

Highlight and copy the first line of code (note: yours will be slightly different as it will include your GitHub username not mine)

`git remote add origin https://github.com/alexd106/second_repo.git`

Switch to RStudio, click on the 'Terminal' tab and paste the command into the Terminal. Now go back to GitHub and copy the second line of code

`git push -u origin master`

and paste this into the Terminal in RStudio. You should see something like this

If you take a look at your repo back on GitHub (click on the `/second_repo` link at the top) you will see the `second_repo.Rproj` and `.gitignore` files have now been **pushed** to GitHub from your local repository.
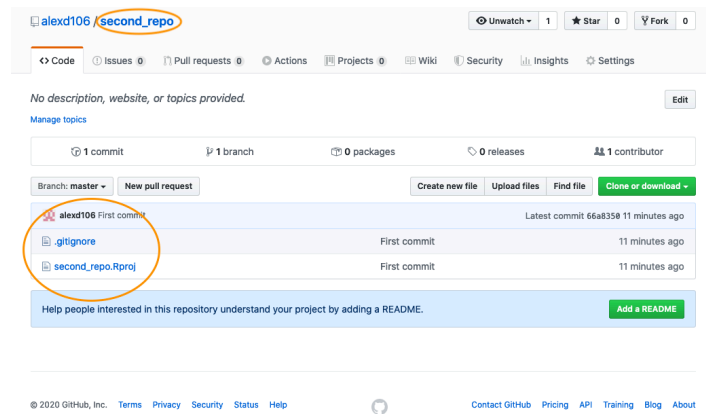
The last thing we need to do is create and add a **README** file to your repository. A README file describes your project and is written using the same Markdown language you learned in the R markdown Chapter. A good README file makes it easy for others (or the future you!) to use your code and reproduce your project. You can create a README file in RStudio or in GitHub. Let's use the second option.

In your repository on GitHub click on the green `Add a README` button.
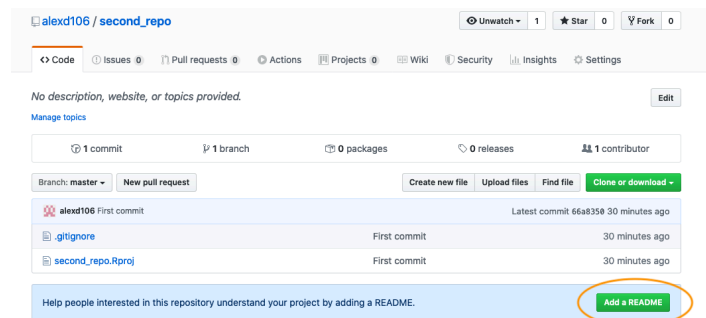
Now write a short description of your project in the `<> Edit new file` section and then click on the green `Commit new file` button.
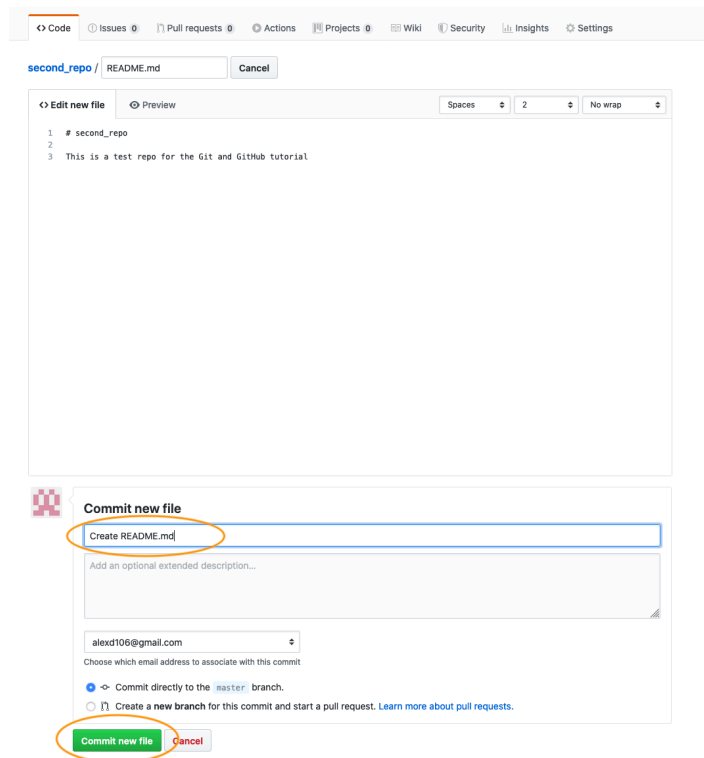
You should now see the `README.md` file listed in your repository. It won't actually exist on your computer yet as you will need to **pull** these changes back to your local repository, but more about that in the next section.

Whether you followed Option 1 or Option 2 (or both) you have now successfully setup a version controlled RStudio project (and associated directory) and linked this to a GitHub repository. Git will now monitor this directory for any changes you make to files and also if you add or delete files. If the steps above seem like a bit of an ordeal, just remember, you only need to do this once for each project and it gets much easier over time.

9.6   Using Git

Now that we have our project and repositories (both local and remote) set up, it's finally time to learn how to use Git in RStudio!

Typically, when using Git your workflow will go something like this:

1. You create/delete and edit files in your project directory on your computer as usual (saving these changes as you go)

2. Once you've reached a natural 'break point' in your progress (i.e. you'd be sad if you lost this progress) you **stage** these files

3. You then **commit** the changes you made to these staged files (along with a useful commit message) which creates a permanent snapshot of these changes

4. You keep on with this cycle until you get to a point when you would like to **push** these changes to GitHub

5. If you're working with other people on the same project you may also need to **pull** their changes to your local computer
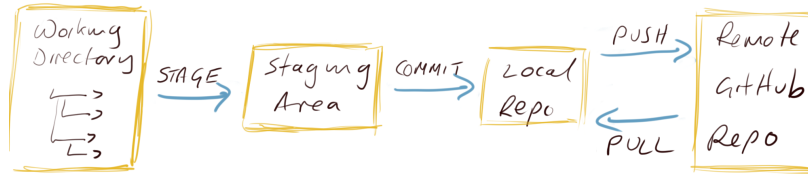
OK, let's go through an example to help clarify this workflow. In RStudio open up the `first_repo.Rproj` you created previously during Option 1. Either use the `File -> Open Project` menu or click on the top right project icon and select the appropriate project.
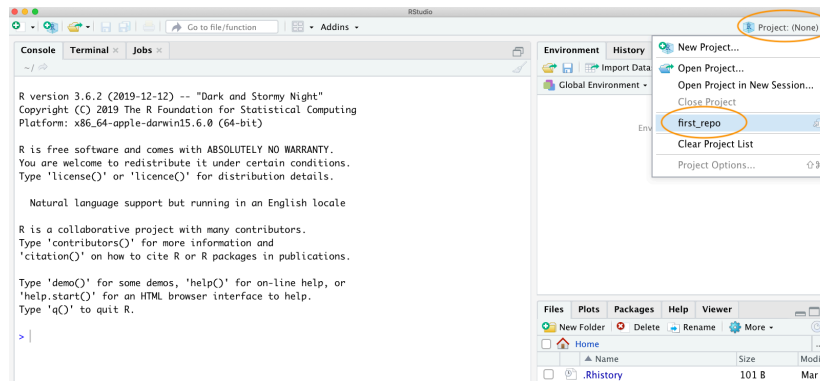
Create an R markdown document inside this project by clicking on the `File -> New File -> R markdown` menu (remember from the R markdown Chapter?).

Once created, we can delete all the example R markdown code (except the YAML header) as usual and write some interesting R markdown text and include a plot. We'll use the inbuilt `cars` dataset to do this. Save this file (cmd + s for Mac or ctrl + s in Windows). Your R markdown document should look something like the following (it doesn't matter if it's not exactly the same).

Take a look at the 'Git' tab which should list your new R markdown document (`first_doc.Rmd` in this example) along with `first_repo.Rproj`, and `.gitignore` (you created these files previously when following Option 1).
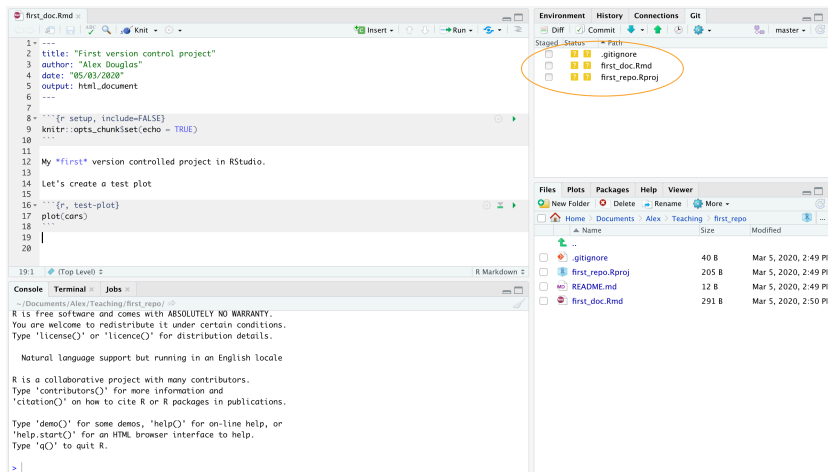
Following our workflow, we now need to **stage** these files. To do this tick the boxes under the 'Staged' column for all files. Notice that there is a status icon next to the box which gives you an indication of how the files were changed. In our case all of the files are to be added (capital A) as we have just created them.

After you have staged the files the next step is to **commit** the files. This is done by clicking on the 'Commit' button.
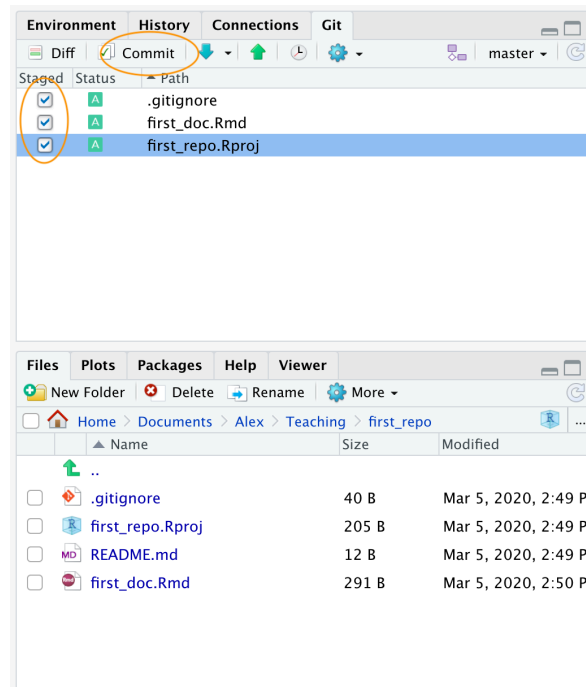
After clicking on the 'Commit' button you will be taken to the 'Review Changes' window. You should see the three files you staged from the previous step in the left pane. If you click on the file name `first_doc.Rmd` you will see the changes you have made to this file highlighted in the bottom pane. Any content that you have added is highlighted in green and deleted content is highlighted in red. As you have only just created this file, all the content is highlighted in green. To commit these files (take a snapshot) first enter a mandatory commit message in the 'Commit message' box. This message should be relatively short and informative (to you and your collaborators) and indicate why you made the changes, not what you changed. This makes sense as Git keeps track of *what* has changed and so it is best not to use commit messages for this purpose. It's traditional to enter the message 'First commit' (or 'Initial commit') when you commit files for the first time. Now click on the 'Commit' button to commit these changes.
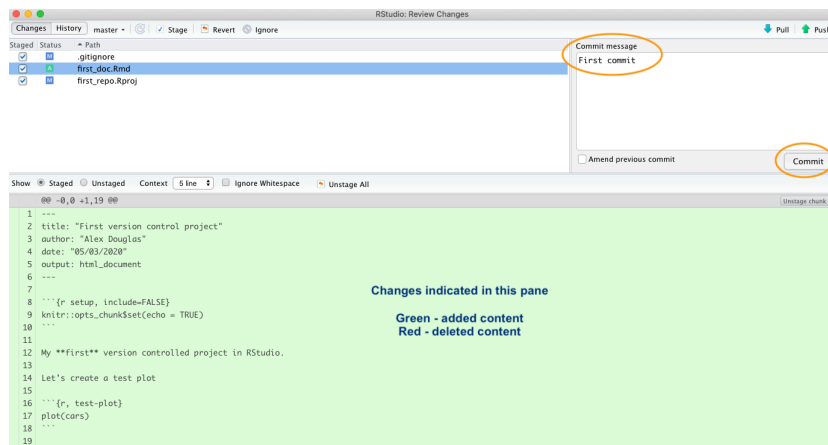
A summary of the commit you just performed will be shown. Now click on the 'Close' button to return to the 'Review Changes' window. Note that the staged files have now been removed.

Now that you have committed your changes the next step is to **push** these changes to GitHub. Before you push your changes it's good practice to first **pull** any changes from GitHub. This is especially important if both you and your collaborators are working on the same files as it keeps you local copy up to date and avoids any potential conflicts. In this case your repository will already be up to date but it's a good habit to get into. To do this, click on the 'Pull' button on the top right of the 'Review Changes' window. Once you have pulled any changes click on the green 'Push' button to push your changes. You will see a summary of the push you just performed. Hit the 'Close' button and then close the 'Review Changes' window.
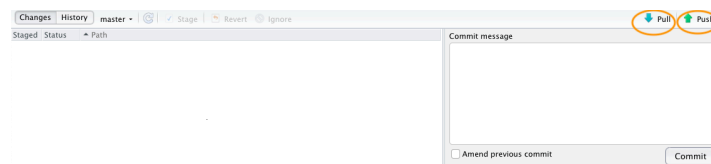
To confirm the changes you made to the project have been pushed to GitHub, open your GitHub page, click on the Repositories link and then click on the `first_repo` repository. You should see four files listed including the `first_doc.Rmd` you just pushed. Along side the file name you will see your last commit message ('First commit' in this case) and when you made the last commit.
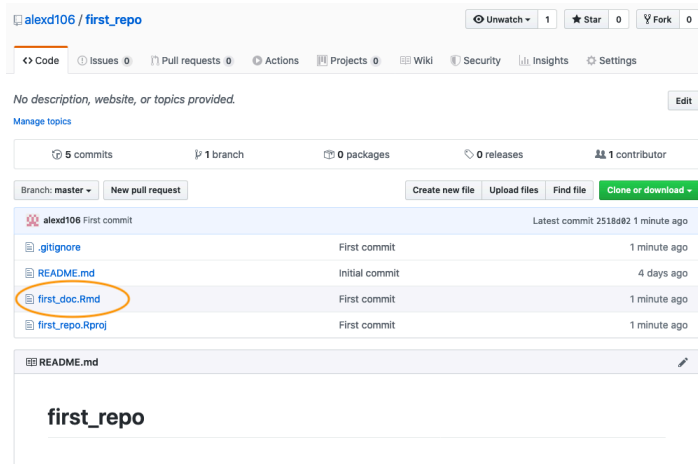
To see the contents of the file click on the `first_doc.Rmd` file name.
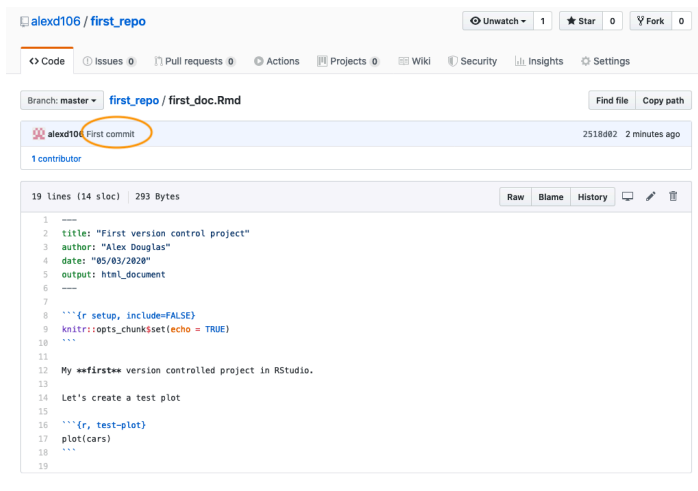
9.6.1 Tracking changes

After following the steps outlined above, you will have successfully modified an RStudio project by creating a new R markdown document, staged and then committed these

changes and finally pushed the changes to your GitHub repository. Now let's make some further changes to your R markdown file and follow the workflow once again but this time we'll take a look at how to identify changes made to files, examine the commit history and how to restore to a previous version of the document.

In RStudio open up the `first_repo.Rproj` file you created previously (if not already open) then open the `first_doc.Rmd` file (click on the file name in the `Files` tab in RStudio).

Let's make some changes to this document. Delete the line beginning with 'My first version controlled ...' and replace it with something more informative (see figure below). We will also change the plotted symbols to red and give the plot axes labels. Lastly, let's add a summary table of the dataframe using the `kable()` and `summary()` functions (you may need to install the `knitr` package if you haven't done so previously to use the `kable()` function) and finally render this document to pdf by changing the YAML option to `output: pdf_document`.

Now save these changes and then click the `knit` button to render to pdf. A new pdf file named `first_doc.pdf` will be created which you can view by clicking on the file name in the `Files` tab in RStudio.

Notice that these two files have been added to the `Git` tab in RStudio. The status icons indicate that the `first_doc.Rmd` file has been modified (capital M) and the `first_doc.pdf` file is currently untracked (question mark).
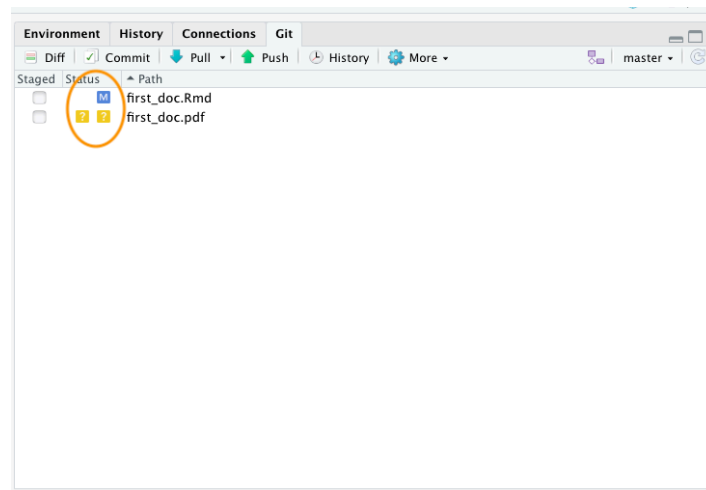
To stage these files tick the 'Staged' box for each file and click on the 'Commit' button to take you to the 'Review Changes' window.
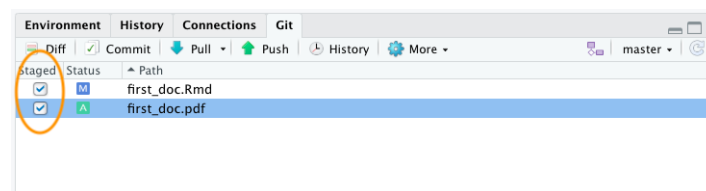
Before you commit your changes notice the status of `first_doc.pdf` has changed from untracked to added (A). You can view the changes you have made to the `first_doc.Rmd` by clicking on the file name in the top left pane which will provide you with a useful summary of the changes in the bottom pane (technically called diffs). Lines that have been deleted are highlighted in red and lines that have been added are highlighted in green (note that from Git's point of view, a modification to a line is actually two operations: the removal of the original line followed by the creation of a new line). Once you're happy, commit these changes by writing a suitable commit message and click on the 'Commit' button.
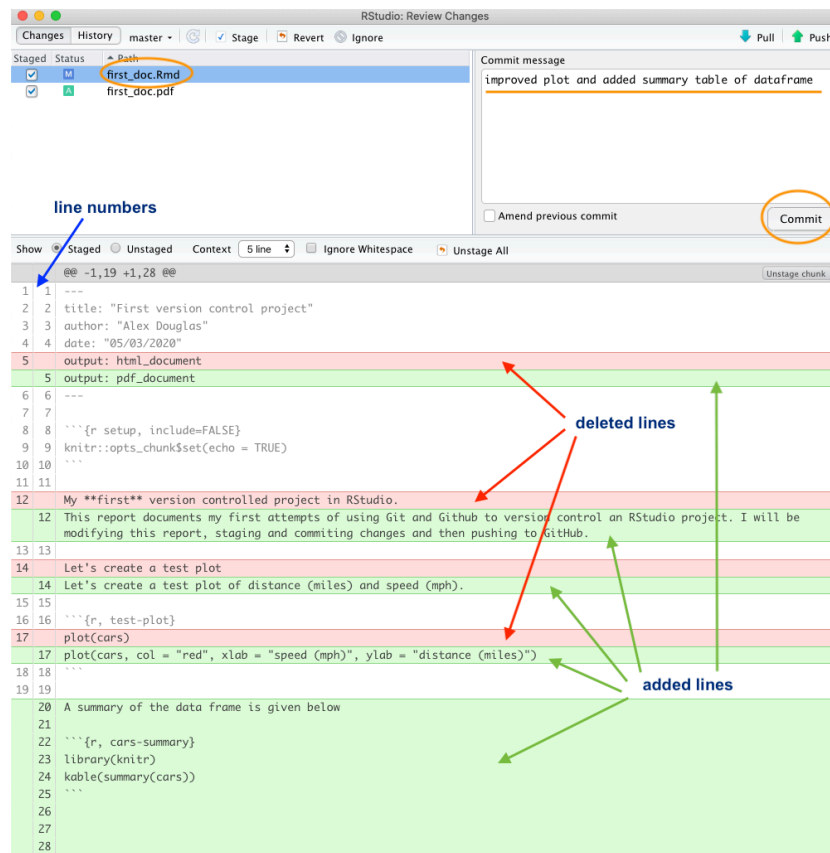
To push the changes to GitHub, click on the 'Pull' button first (remember this is good practice even though you are only collaborating with yourself at the moment) and then click on the 'Push' button. Go to your online GitHub repository and you will see your new commits, including the `first_doc.pdf` file you created when you rendered your R markdown document.

To view the changes in `first_doc.Rmd` click on the file name for this file.

9.6.2 Commit history

One of the great things about Git and GitHub is that you can view the history of all the commits you have made along with the associated commit messages. You can do this locally using RStudio (or the Git command line) or if you have pushed your commits to GitHub you can check them out on the GitHub website.

To view your commit history in RStudio click on the 'History' button (the one that looks like a clock) in the Git pane to bring up the history view in the 'Review Changes' window. You can also click on the 'Commit' or 'Diff' buttons which takes you to the same window (you just need to additionally click on the 'History' button in the 'Review Changes' window).
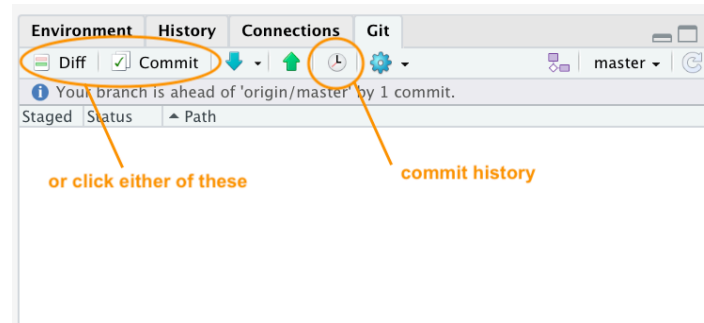
The history window is split into two parts. The top pane lists every commit you have made in this repository (with associated commit messages) starting with the most recent one at the top and oldest at the bottom. You can click on each of these commits and the bottom pane shows you the changes you have made along with a summary of the **Date** the commit was made, **Author** of the commit and the commit message (**Subject**). There is also a unique identifier for the commit (**SHA** - Secure Hash Algorithm) and a **Parent** SHA which identifies the previous commit. These SHA identifiers are really important as you can use them to view and revert to previous versions of files (details below). You can also view the contents of each file by clicking on the 'View file @ SHA key' link (in our case 'View file @ 2b4693d1').
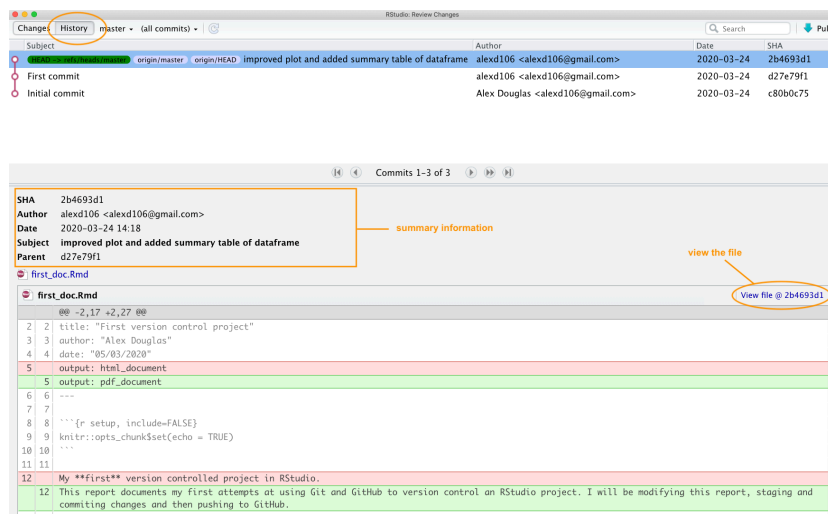
You can also view your commit history on GitHub website but this will be limited to only those commits you have already pushed to GitHub. To view the commit history navigate to the repository and click on the 'commits' link (in our case the link will be labelled '3 commits' as we have made 3 commits).
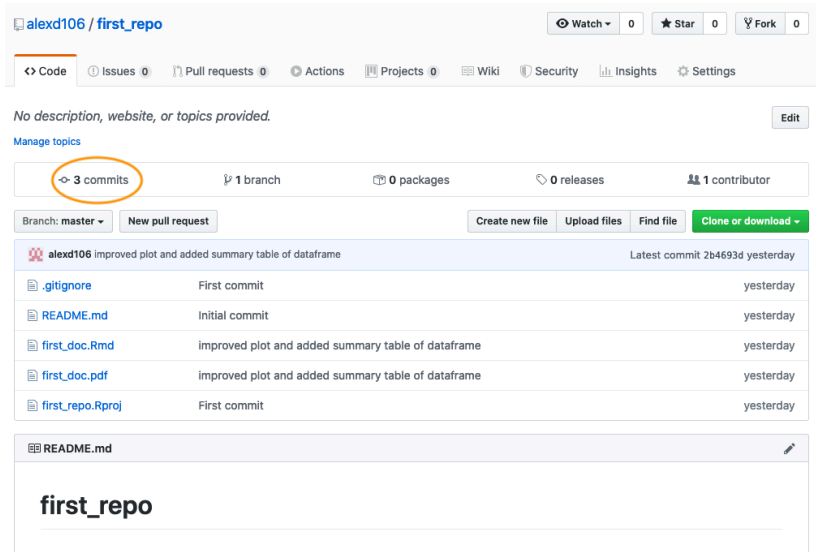
You will see a list of all the commits you have made, along with commit messages, date of commit and the SHA identifier (these are the same SHA identifiers you saw in the RStudio history). You can even browse the repository at a particular point in time by clicking on the <> link. To view the changes in files associated with the commit simply click on the relevant commit link in the list.
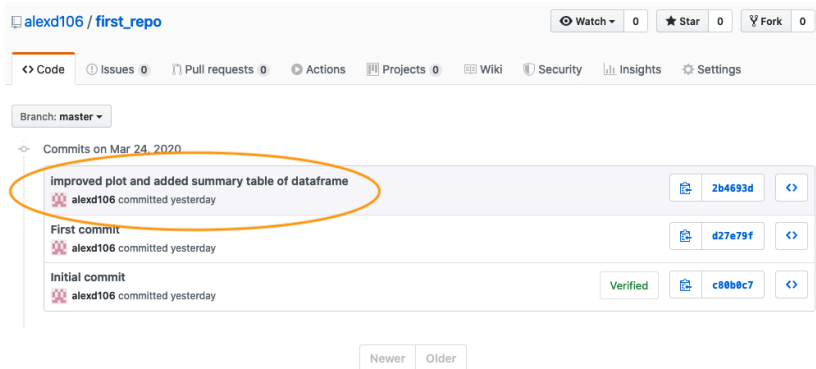
Which will display changes using the usual format of green for additions and red for deletions.
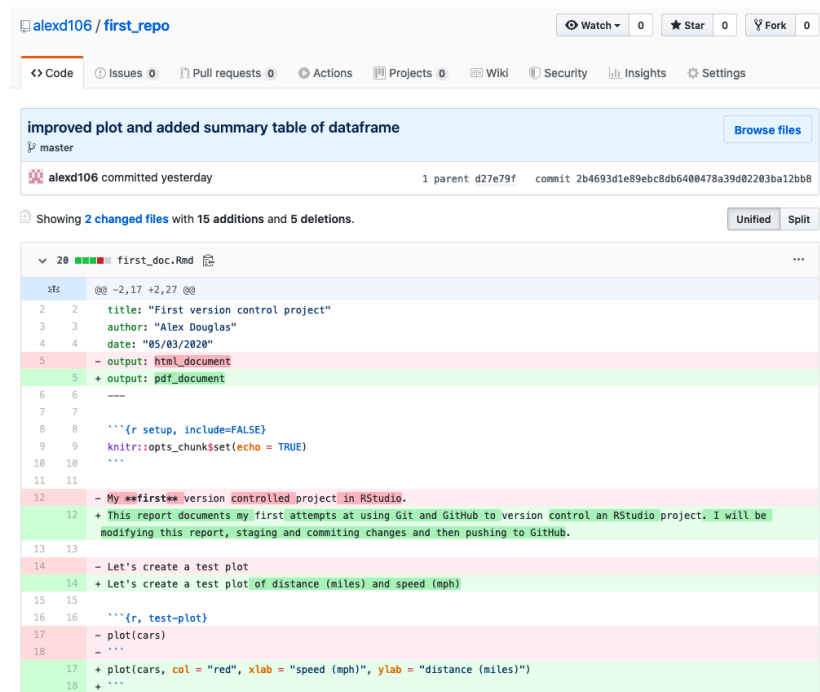
9.6.3   Reverting changes

One the great things about using Git is that you are able to revert to previous versions of files if you've made a mistake, broke something or just prefer and earlier approach. How you do this will depend on whether the changes you want to discard have been staged, committed or pushed to GitHub. We'll go through some common scenarios below mostly using RStudio but occasionally we will need to resort to using the Terminal (still in RStudio though).

Changes saved but not staged, committed or pushed

If you have saved changes to your file(s) but not staged, committed or pushed these files to GitHub you can right click on the offending file in the Git pane and select 'Revert …'. This will roll back all of the changes you have made to the same state as your last commit. Just be aware that you cannot undo this operation so use with caution.
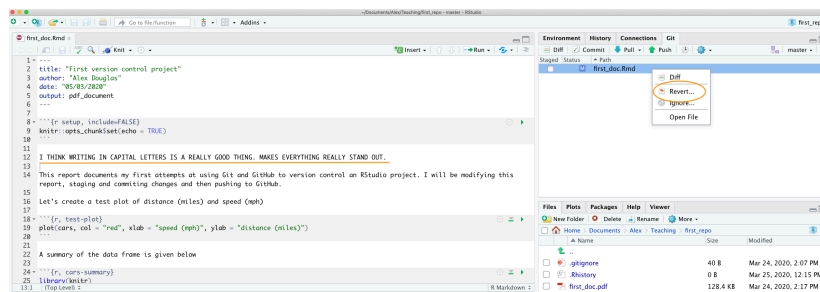
You can also undo changes to just part of a file by opening up the 'Diff' window (click on the 'Diff' button in the Git pane). Select the line you wish to discard by double clicking on the line and then click on the 'Discard line' button. In a similar fashion you can discard chunks of code by clicking on the 'Discard chunk' button.
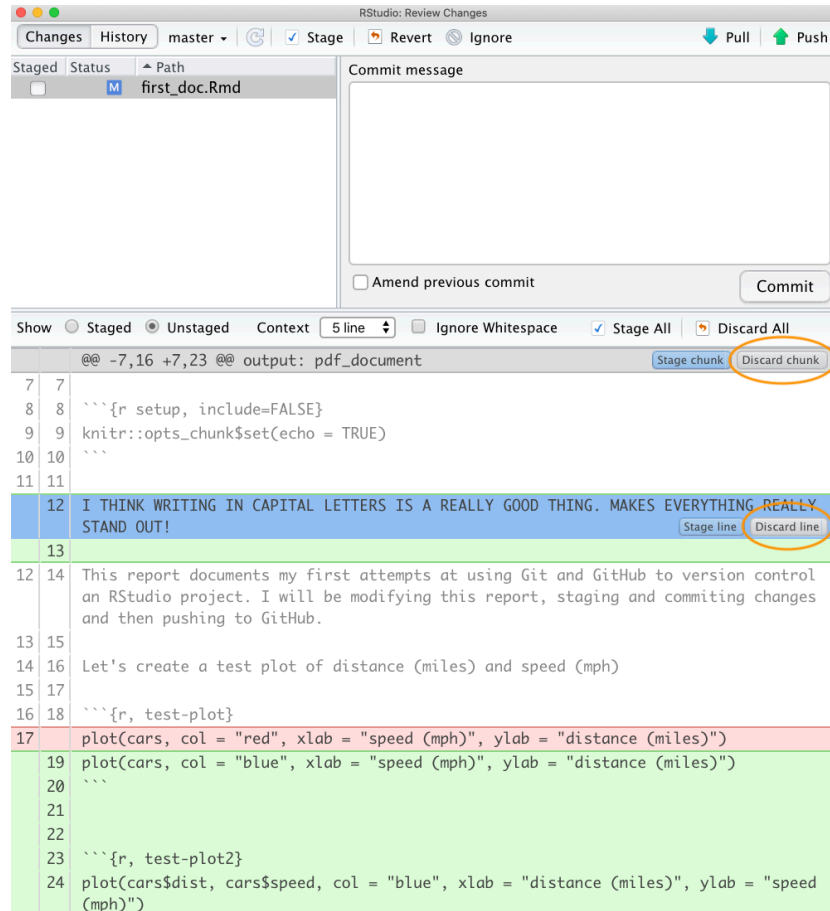
Staged but not committed and not pushed

If you have staged your files, but not committed them then simply unstage them by clicking on the 'Staged' check box in the Git pane (or in the 'Review Changes' window) to remove the tick. You can then revert all or parts of the file as described in the section above.

Staged and committed but not pushed

If you have made a mistake or have forgotten to include a file in your last commit which you have not yet pushed to GitHub, you can just fix your mistake, save your changes, and then amend your previous commit. You can do this by staging your file and then tick the 'Amend previous commit' box in the 'Review Changes' window before committing.
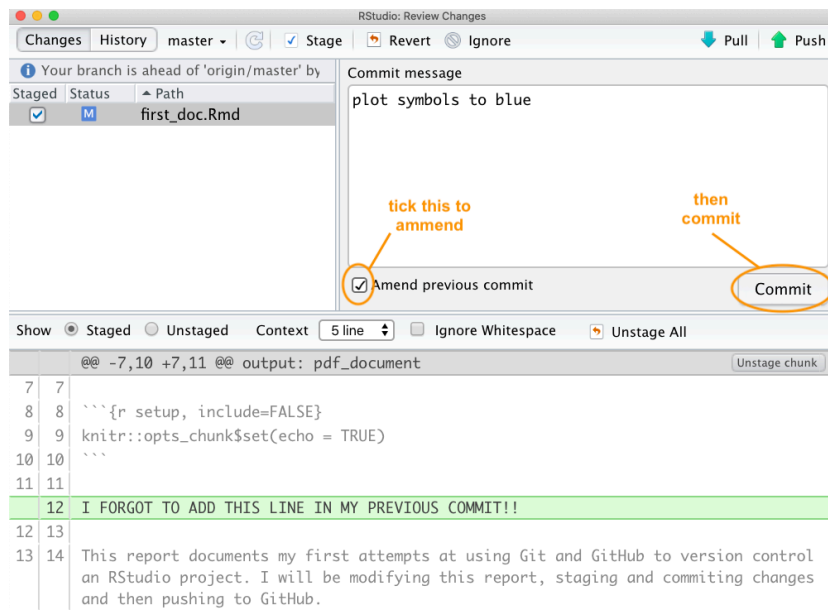
If we check out our commit history you can see that our latest commit contains both changes to the file rather than having two separate commits. We use the amend commit approach alot but it's important to understand that you should **not** do this if you have already pushed your last commit to GitHub as you are effectively rewriting history and all sorts bad things may happen!

If you spot a mistake that has happened multiple commits back or you just want to revert to a previous version of a document you have a number of options.

Option 1 - (probably the easiest but very unGit - but like, whatever!) is to look in your commit history in RStudio, find the commit that you would like to go back to and click on the 'View file @' button to show the file contents.

You can then copy the contents of the file to the clipboard and paste it into your current file to replace your duff code or text. Alternatively, you can click on the 'Save As' button and save the file with a different file name. Once you have saved your new file you can delete your current unwanted file and then carry on working on your new file. Don't forget to stage and commit this new file.

Option 2 - (Git like) Go to your Git history, find the commit you would like to roll back to and write down (or copy) its SHA identifier.

Now go to the Terminal in RStudio and type `git checkout <SHA> <filename>`. In our case the SHA key is `2b4693d1` and the filename is `first_doc.Rmd` so our command would look like this:

```
git checkout 2b4693d1 first_doc.Rmd
```

The command above will copy the selected file version from the past and place it into the present. RStudio may ask you whether you want to reload the file as it now changed - select yes. You will also need to stage and commit the file as usual.

If you want to revert all your files to the same state as a previous commit rather than just one file you can use (the single 'dot' . is important otherwise your HEAD will detach!):

```
git rm -r .
git checkout 2b4693d1 .
```

Note that this will delete all files that you have created since you made this commit so be careful!

Staged, committed and pushed

If you have already pushed your commits to GitHub you can use the `git checkout` strategy described above and then commit and push to update GitHub (although this is not really considered 'best' practice). Another approach would be to use `git revert` (Note: as far as we can tell `git revert` is not the same as the 'Revert' option in RStudio). The `revert` command in Git essentially creates a new commit based on a previous commit and therefore preserves all of your commit history. To rollback to a previous state (commit) you first need to identify the SHA for the commit you wish to go back to (as we did above) and then use the `revert` command in the Terminal. Let's say we want to revert back to our 'First commit' which has a SHA identifier `d27e79f1`.
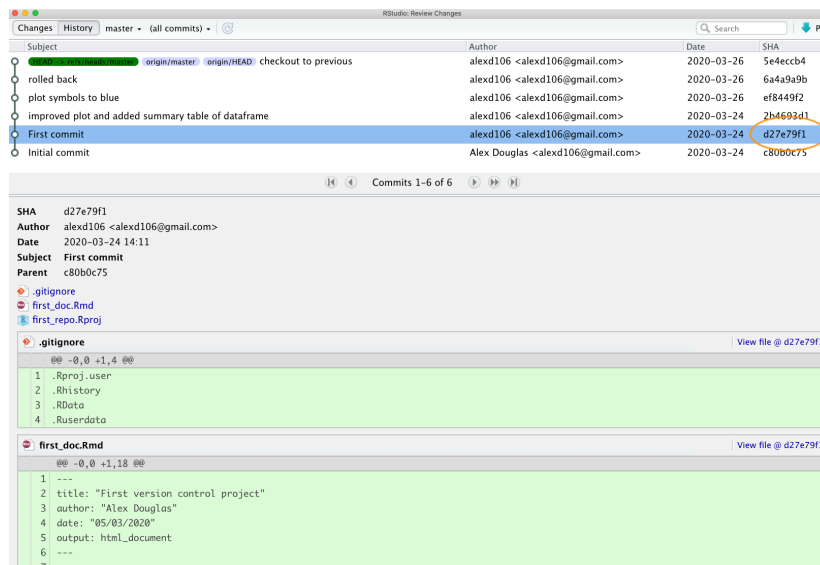
We can use the `revert` command as shown below in the Terminal. The `--no-commit` option is used to prevent us from having to deal with each intermediate commit.

```
git revert --no-commit d27e79f1..HEAD
```

Your `first_doc.Rmd` file will now revert back to the same state as it was when you did your 'First commit'. Notice also that the `first_doc.pdf` file has been deleted as this wasn't present when we made our first commit. You can now stage and commit these files with a new commit message and finally push them to GitHub. Notice that if we look at our commit history all of the commits we have made are still present.

and our repo on GitHub also reflects these changes

9.6.4   Collaborate with Git

GitHub is a great tool for collaboration, it can seem scary and complicated at first, but it is worth investing some time to learn how it works. What makes GitHub so good for collaboration is that it is a *distributed system*, which means that every collaborator works on their own copy of the project and changes are then merged together in the remote repository. There are two main ways you can set up a collaborative project on GitHub. One is the workflow we went through above, where everybody connects their local repository to the same remote one; this system works well with small projects where different people mainly work on different aspects of the project but can quickly become unwieldy if many people are collaborating and are working on the same files (merge misery!). The second approach consists of every collaborator creating a copy (or **fork**) of the main repository, which becomes their remote repository. Every collaborator then needs to send a request (a **pull request**) to the owner of the main repository to incorporate any changes into the main repository and this includes a review process before the changes are integrated. More detail of these topics can be found in the Further resources section.

9.6.5   Git tips

Generally speaking you should commit often (including amended commits) but push much less often. This makes collaboration easier and also makes the process of reverting to previous versions of documents much more straight forward. We generally only push changes to GitHub when we're happy for our collaborators (or the rest of the world) to see our work. However, this is entirely up to you and depends on the project (and who you are working with) and what your priorities are when using Git.

If you don't want to track a file in your repository (maybe they are too large or transient files) you can get Git to ignore the file by right clicking on the filename in the Git pane and selecting 'Ignore...'
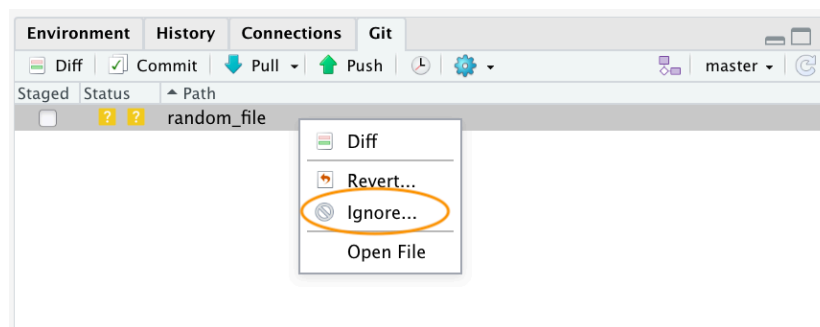
This will add the filename to the `.gitignore` file. If you want to ignore multiple files or a particular type of file you can also include wildcards in the `.gitignore` file. For example

to ignore all `png` files you can include the expression `*.png` in your `.gitignore` file and save.

If it all goes pear shaped and you end up completely trashing your Git repository don't despair (we've all been there!). As long as your GitHub repository is good, all you need to do is delete the offending project directory on your computer, create a new RStudio project and link this with your remote GitHub repository using Option 2. Once you have cloned the remote repository you should be good to go.

9.7   Further resources

There are many good online guides to learn more about git and GitHub and as with any open source software there is a huge community that can be a great resource:

- The British Ecological Society guide to Reproducible Code

- The GitHub guides

- The Mozilla Science Lab GitHub for Collaboration on Open Projects guide

- Jenny Bryan's Happy Git and GitHub. We borrowed the idea (but with different content) of RStudio first, RStudio second in the 'Setting up a version controlled Project in RStudio' section.

- Melanie Frazier's GitHub: A beginner's guide to going back in time (aka fixing mistakes). We followed this structure (with modifications and different content) in the 'Reverting changes' section.

- If you have done something terribly wrong and don't know how to fix it try Oh Shit, Git or if you're easily offended Dangit, Git

These are only a couple of examples, all you need to do is Google "version control with git and GitHub" to see how huge the community around these open source projects is and how many free resources are available for you to become a version control expert.

Appendix A

Installing R Markdown

Installing R markdown on your computer is pretty straight forward and should be painless. If you're getting started with R markdown we suggest that you use RStudio but of course RStudio is not required and there are other options available. You will also need to install some additional software if you want to render your R markdown documents to PDF format. The steps to do this are given in the following sections for both Windows and Mac computers.

A.1 MS Windows

This guide assumes you have already installed R and the RStudio IDE. RStudio is not required but recommended, because it makes it easier to work with R Markdown. If you don't have RStudio IDE installed, you will also have to install Pandoc. If you have RStudio installed there is no need to install Pandoc separately because it's bundled with RStudio. Next you can install the rmarkdown package in RStudio using the following code:

```r
# Install from CRAN
install.packages('rmarkdown', dep = TRUE)
```

The dep = TRUE argument will also install a bunch of additional R packages on which rmarkdown depends.

If you want to generate PDF output, you will need to install LaTeX. For R Markdown users who have not installed LaTeX before, we recommend that you install TinyTeX. You can install TinyTex from within RStudio using the following code:

```r
install.packages('tinytex')
tinytex::install_tinytex()  # install TinyTeX
```

TinyTeX is a lightweight, portable, cross-platform, and easy-to-maintain LaTeX distribution. The R companion package tinytex can help you automatically install missing LaTeX packages when compiling LaTeX or R Markdown documents to PDF. An alternative option would be to install MiKTeX instead. You can download the latest distribution of MiKTeX. Installing MiKTeX is pretty straight forward, but it can sometimes be a pain to get it to play nicely with RStudio. If at all possible we recommend that you use TinyTex.

With the rmarkdown package, RStudio/Pandoc, and LaTeX, you should be able to compile most R Markdown documents.

A.2   Mac OSX

This guide assumes you have already installed R and the RStudio IDE. RStudio is not required but recommended, because it makes it easier to work with R Markdown. If you do not have RStudio IDE installed, you will also have to install Pandoc. If you have RStudio installed there is no need to install Pandoc separately because its bundled with RStudio. Next you can install the rmarkdown package in RStudio using the following code:

```r
# Install from CRAN
install.packages('rmarkdown', dep = TRUE)
```

The `dep = TRUE` argument will also install a bunch of additional R packages on which rmarkdown depends.

If you want to generate PDF output, you will need to install LaTeX. For R Markdown users who have not installed LaTeX before, we recommend that you install TinyTeX. You can install TinyTex from within RStudio using the following code:

```r
install.packages('tinytex')
tinytex::install_tinytex()  # install TinyTeX
```

TinyTeX is a lightweight, portable, cross-platform, and easy-to-maintain LaTeX distribution. The R companion package tinytex can help you automatically install missing LaTeX packages when compiling LaTeX or R Markdown documents to PDF.

If for some reason TinyTeX does not work on your Mac computer then you can try to install MacTeX instead. You can download the latest version of MacTeX. Click on the *MacTeX.pkg* link to download. Please be aware that the file is quite large, approximately 3 GB, so it may take some time (also make sure you have enough available space on your computer hard disk). When your download is complete, run the downloaded installer. The installation procedure is quite straightforward. You are given a few options, for example you can choose not to install some components. I recommend

you stick with the default settings, so having accepted the license agreement, you can basically just keep clicking 'Continue' on each screen. On the final screen of the wizard, click Install. It may ask for an administrator password. Enter the password, click 'Install Software' and go make a cup of coffee while the installation completes. When the program is done installing, click 'Close' to complete the installation.

With the rmarkdown package, RStudio/Pandoc, and LaTeX, you should be able to compile most R Markdown documents.

Index