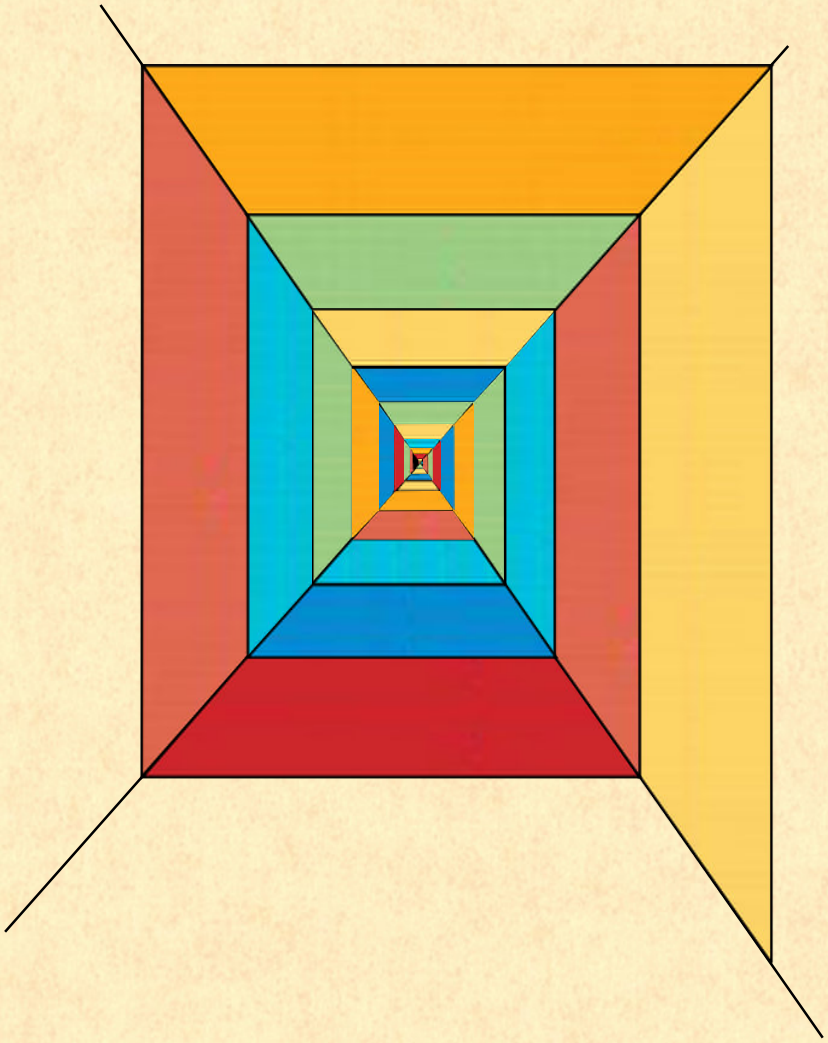
An Introduction to R



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

An Introduction to R

Nikhil Kaza

August 14, 2022

Contents

Preface

0.1 The aim of this book

This book is a derivative of "An Introduction to R" by Alex Douglas, Deon Roos, Francesca Mancini, Ana Couto & David Lusseau". I have adapated it in 2022 to make it suitable for students who are getting started with Urban Analytics course and want to have a refresher course in R. As such much of this book is verbatim from Douglas et. al(2022) with no claim to originality. The same license (CC-NC-SA) applies.

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. 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 get over the initial learning hurdle 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 is primarily intended for students who have some experience with programming and little experience with R. It is also intended for students, typically advanced undergraduates and graduate students, who are interested in urban analytics. I'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 How to use this book

For the best experience we recommend that you read the web version of this book which you can find here. 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.

0.4 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.

0.5 Some R pointers

- 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 elses, 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, don't lose sight of what's important - your analytical 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 payed back 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 on-line (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.

0.6 License

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 Twitter and GitHub).

Chapter 1

Getting started with R and RStudio

Although R is not new, it's popularity has increased rapidly over the last 10 years or so (see here for some interesting data). It was originally created and developed by Ross Ihaka and Robert 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 4.2.1 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-4.2.1.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):

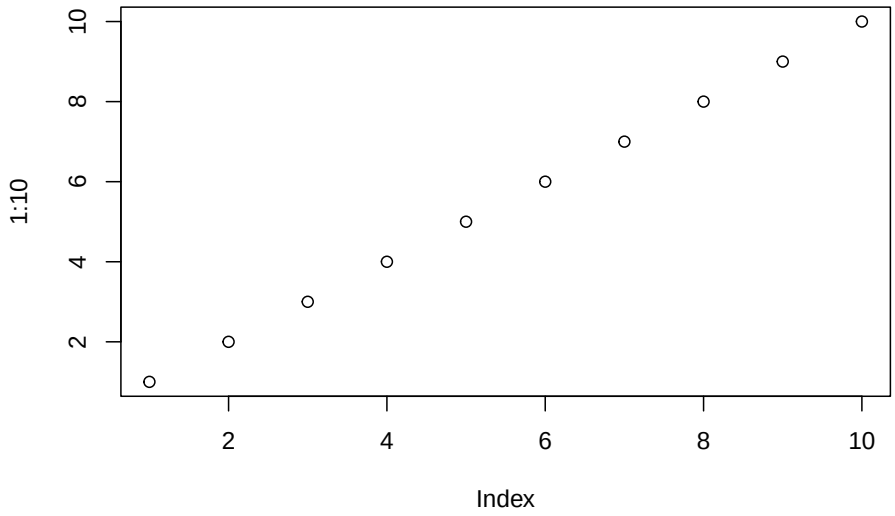
deb https://cloud.r-project.org/bin/linux/ubuntu disco-cran35/

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 of you don't understand this - we're just checking if R works)

```
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 Git Hub 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
```

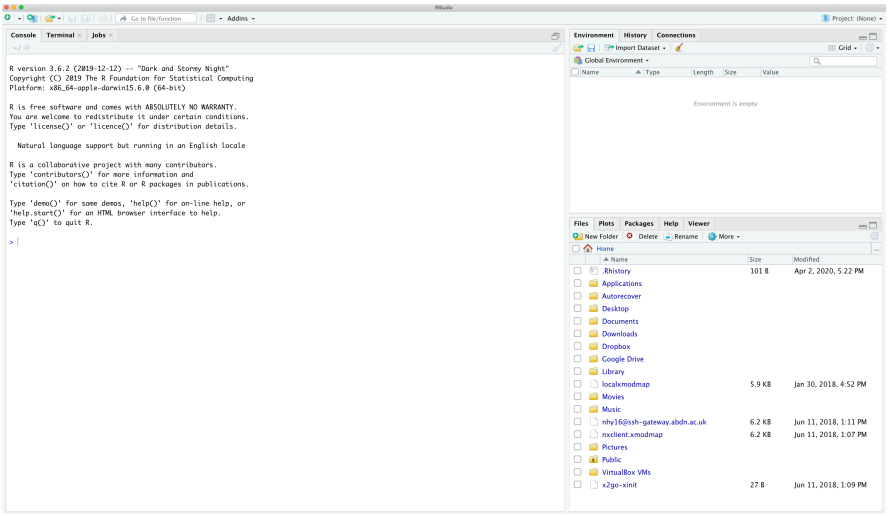
where '-1.2.5XXX' is the current version for Ubuntu (rstudio-xenial-1.2.5019-amd64.deb at the time of writing). You can then start RStudio from the Console by simply typing

```
rstudio
```

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

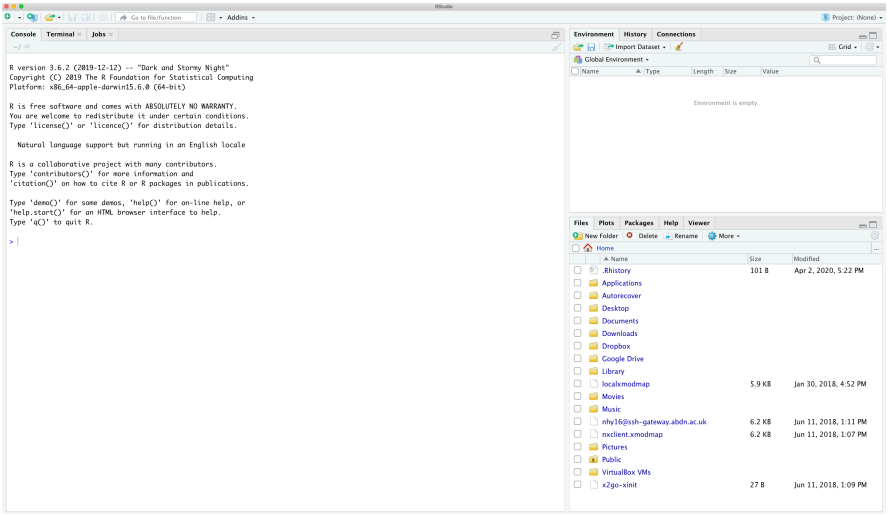
1.2.3 Testing RStudio

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).

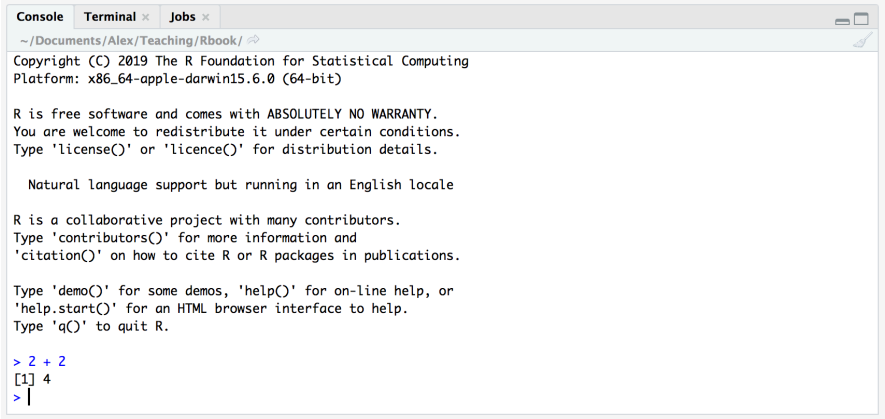


The large window (aka pane) on the left is the **Console** window. The window on the top right is the **Environment / History / Connections** pane and the bottom right window is the **Files / Plots / Packages / Help / Viewer** window. We will discuss each of these panes in turn below. You can customise 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.

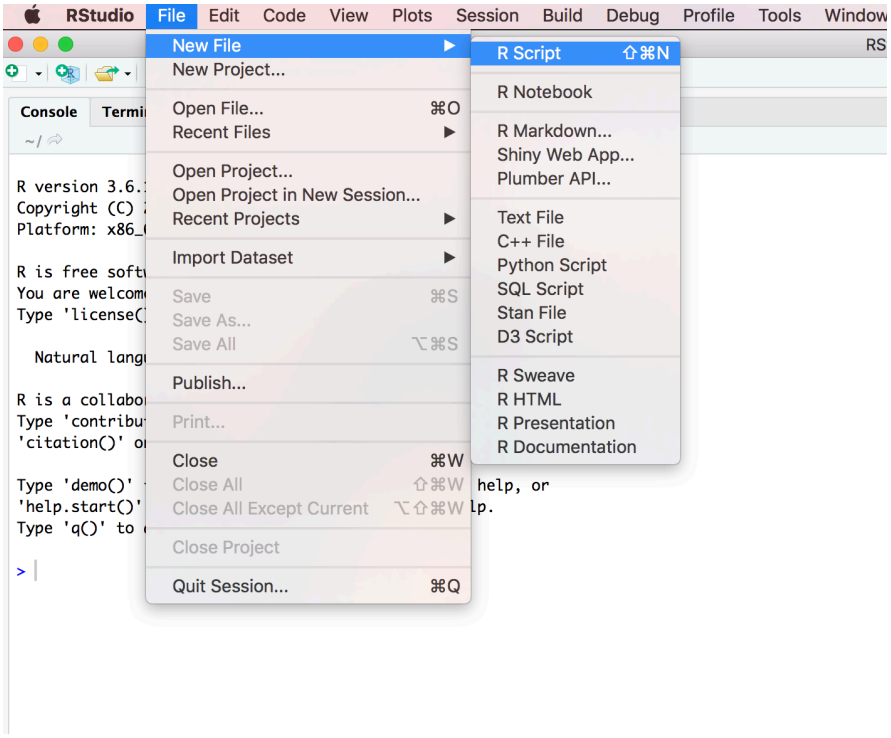
See this video for a quick introduction to RStudio

1.3.1 Console

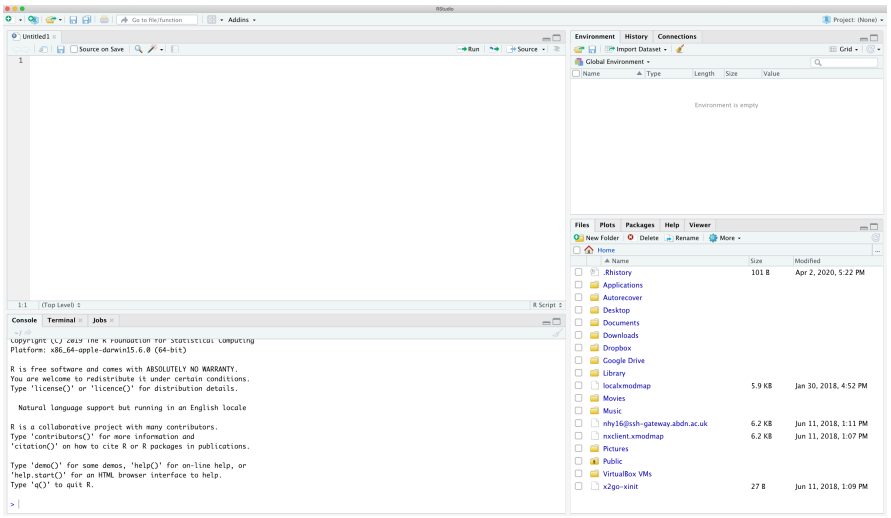
The Console is the workhorse of R. This is where R evaluates all the code you write. You can type R code directly into the Console at the command line prompt, `>`. For example, if you type `2 + 2` into the Console you should obtain the answer `4` (reassuringly). Don't worry about the `[1]` at the start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



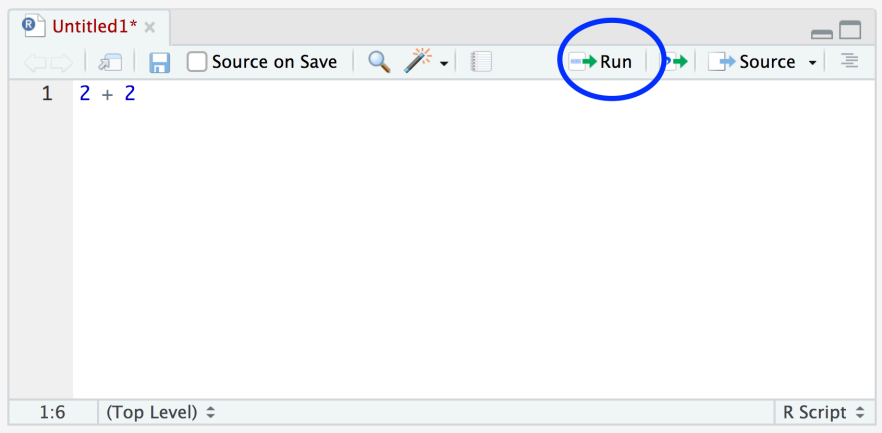
However, once you start writing more R code this becomes rather cumbersome. Instead of typing R code directly into the Console a better approach is to create an R script. An R script is just a plain text file with a `.R` file extension which contains your lines of R code. These lines of code are then 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 `.R` 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 `.R` 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 developement environements

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)

```
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)

```
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

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

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

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

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

```
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

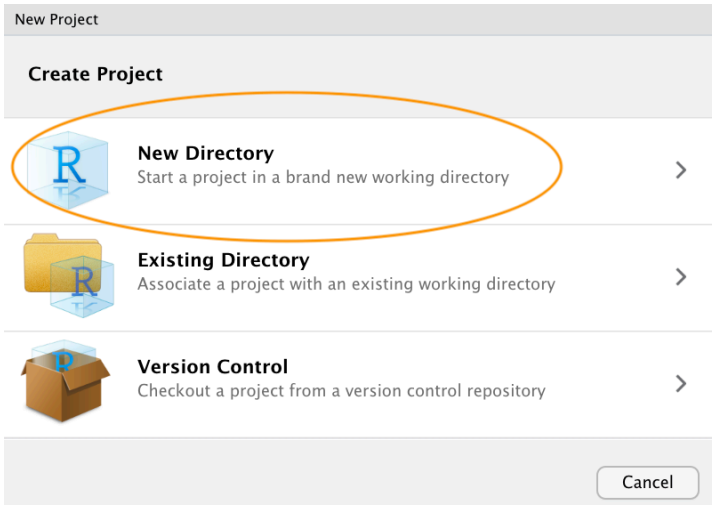
both you (especially the future you) and your collaborators to make sense of what you've done. There's nothing more frustrating than coming back to a project months (sometimes years) later and have to spend days (or weeks) figuring out where everything is, what you did and why you did it. A well documented 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.

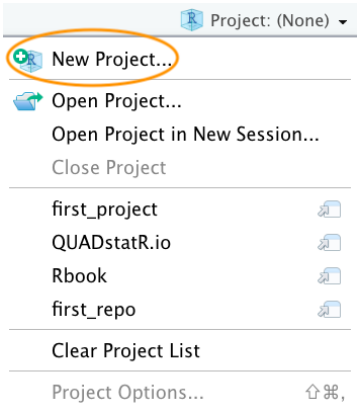
A great way of keeping things organised is to use RStudio Projects. An RStudio Project keeps all of your R scripts, R markdown documents, R functions and data together in one place. The nice thing about RStudio Projects is that each project has its own directory, workspace, history and source documents 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 switch very easily 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

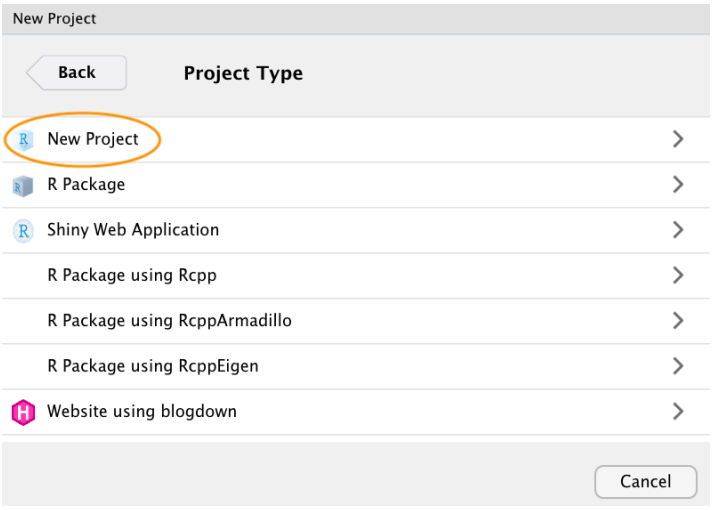
To create a project, open RStudio and select `File -> New Project...` from the menu. You can create either an entirely new project, a project from an existing directory or a version controlled project (see the GitHub Chapter for further details about this). In this Chapter we will create a project in a new directory.



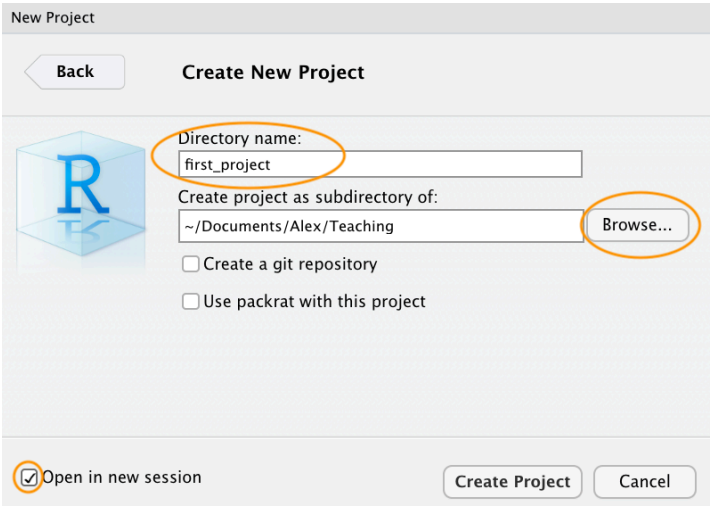
You can also create a new project by clicking on the 'Project' button in the top right of RStudio and selecting 'New Project...'



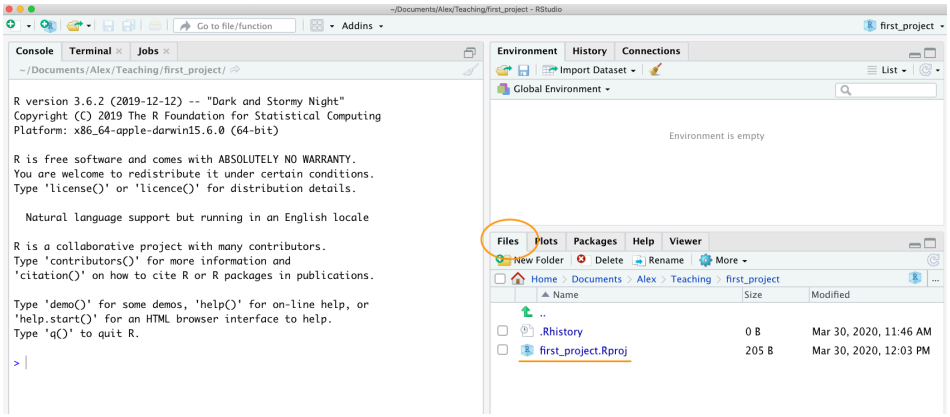
In the next window select 'New Project'.



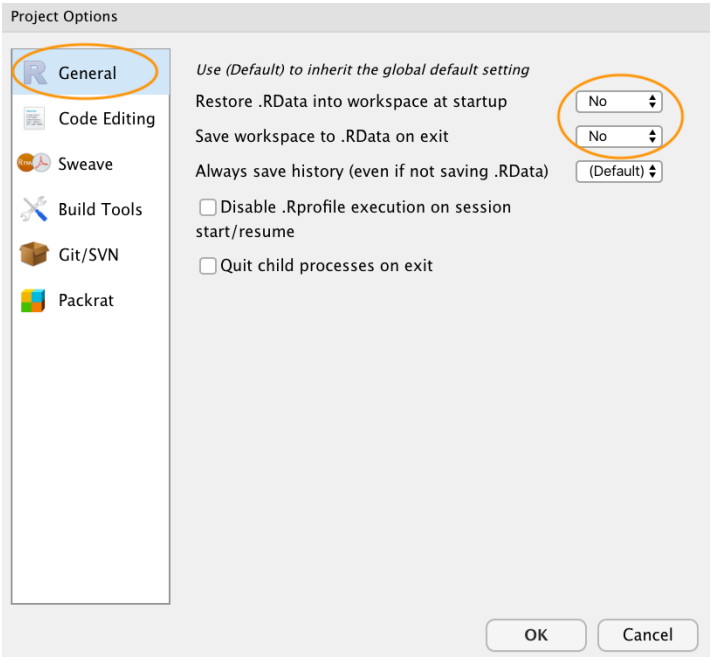
Now enter the name of the directory you want to create in the 'Directory name:' field (we'll call it `first_project` for this Chapter). If you want to change the location of the directory on your computer click the 'Browse...' button and navigate to where you would like to create the directory. We always 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).



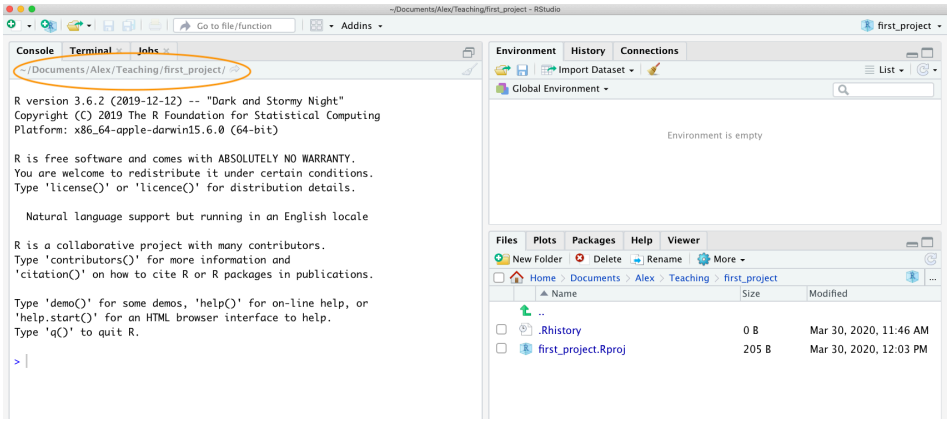
The last thing we suggest you do is select `Tools -> Project Options...` from the menu. Click on the 'General' tab on the left hand side and then change the values for 'Restore .RData into workspace at startup' and 'Save workspace to .RData on exit' from 'Default' to 'No'. This ensures that every time 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 you project.



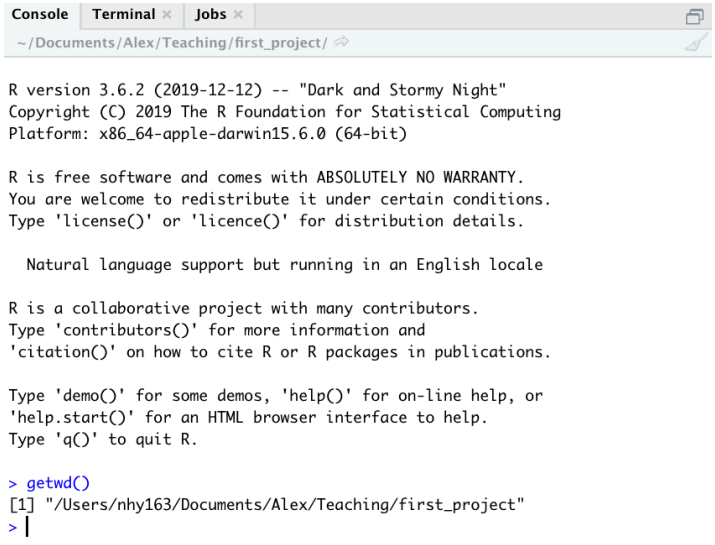
Now that you have an RStudio project set up you can start creating R scripts (or R markdown documents) or whatever you need to complete you project. All of the R scripts will now be contained within the RStudio project and saved in the project folder.

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.



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

```
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.

```
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/myda
    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

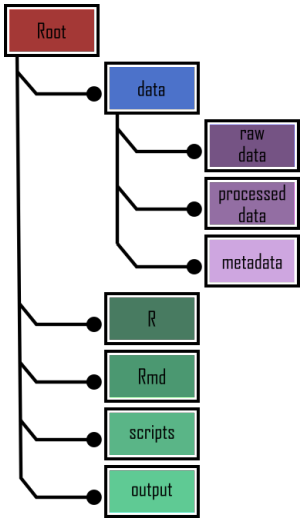
```
library(here)  # you may need to install the here package first
dataf <- read.table(here("raw_data", "mydata.txt"),
                    header = TRUE, sep = '\t',
                    stringsAsFactors = TRUE)

# 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:

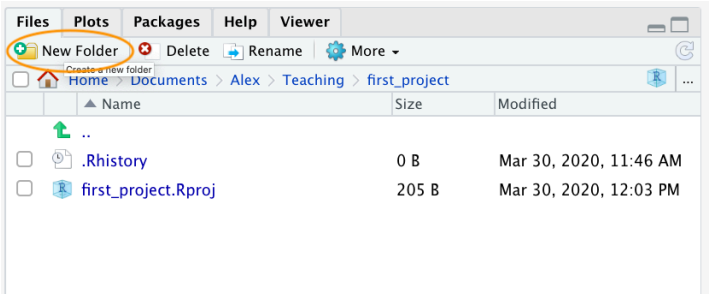
- **Root** - This is your project directory containing your `.Rproj` file.
- **data** - We store all our data in this directory. The subdirectory called `raw_data` contains raw data files and only raw data files. These files should be treated

as **read only** and should not be changed in any way. If you need to process/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 collection 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` subdirectory.

- **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 summaries 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

```
# 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!

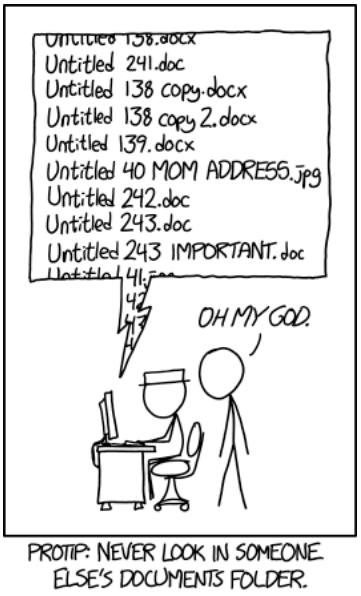


Figure 1.1: source:https://xkcd.com/1459/

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 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

```
# 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:'/Users/Another/snouter_analysis/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()
```

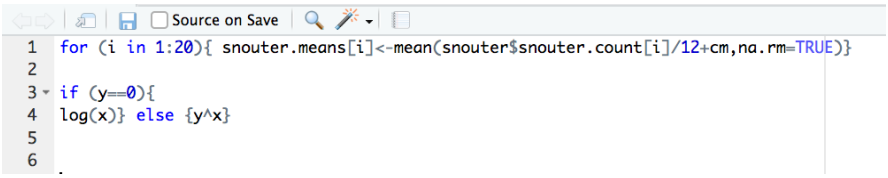
This is just one example and there are no hard and fast rules so feel free to develop a system that works for you. A really useful shortcut in RStudio is to automatically include a time and date stamp in your R script. To do this, write `ts` where you want to insert your time stamp in your R script and 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.

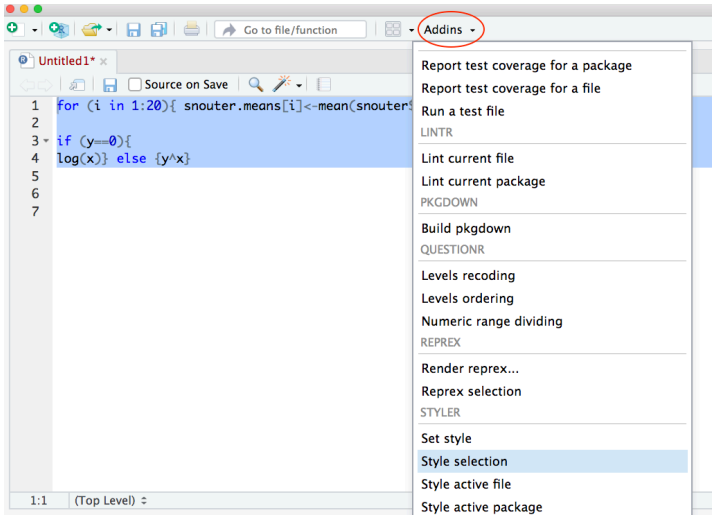
1.11 R style guide

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

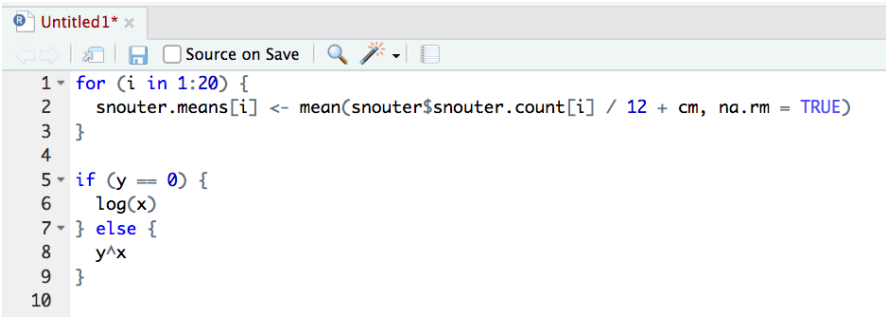
sounds like too much hard work you can install the `styler` package which includes an RStudio add-in to allow you to automatically restyle selected code (or entire files and projects) with the click of your mouse. You can find more information about the `styler` package including how to install here. 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



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 Docs 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 (2022). R: A language and environment for statistical
##   computing. R Foundation for Statistical Computing, Vienna, Austria.
##   URL 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 = {2022},
##     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 here. If you want to see our solutions for this exercise you can find them here (don't peek at them too soon though!).

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 `-` .

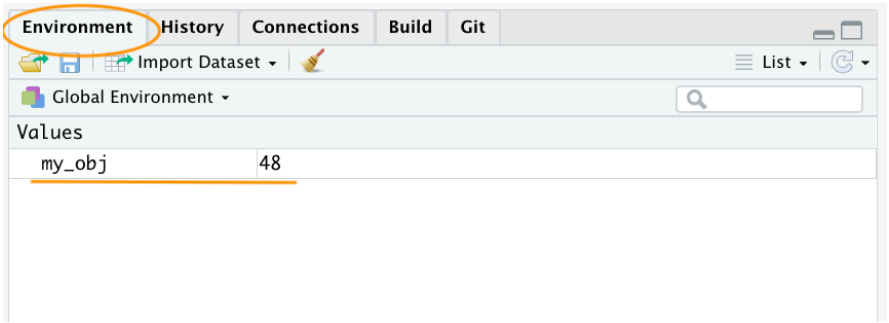
```
my_obj <- 48
```

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.

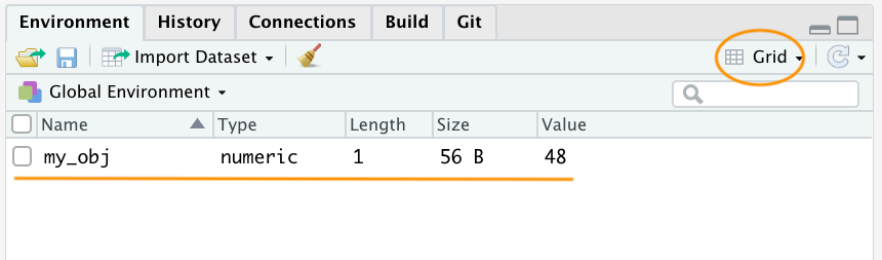
To view the value of the object you simply type the name of the object

```
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).



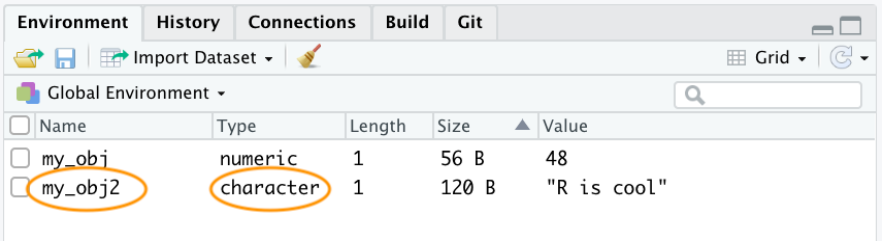
There are many different types of values that you can assign to an object. For example

```
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"
```

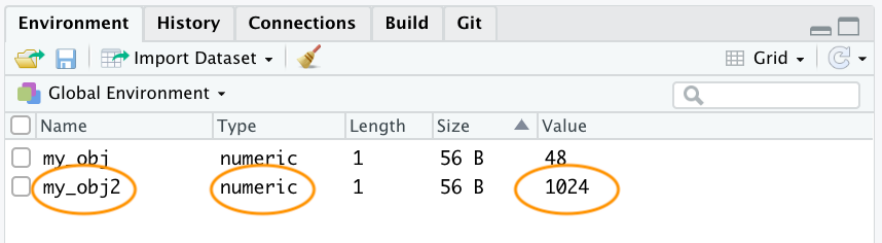
Our workspace now contains both objects we've created so far with `my_obj2` listed as type character.



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
```

2.3.1 Pipes. A better way!!!!!

A better way to nest functions is to use what are called pipes. Pipes are a way of passing the output of one function to another function without storing the values in an object in between. You can use two different pipes, one in native R (`|>`) and another from `magittr` (`%>%`) packages.

```
my_seq8 <- c(3, 1, 10, 7) |>  # Native R version
                            rep(each = 3)

my_seq8
##  [1]  3  3  3  1  1  1 10 10 10  7  7  7

library(magrittr)
my_seq9 <- c(3, 1, 10, 7) %>%  # magittr version
                            rep(each = 3)
```

```
all.equal(my_seq8, my_seq9)
## [1] TRUE
```

You can build long series of these nested functions as you will do in the urban analytics course.

Notice that in both cases the the first argument of the `rep` function is not specified. That is because the output of the previous step before the pipe is automatically the first argument. This is how you will use it in many situations, however, in more advanced settings you can change the output to be a different agument of a later functions. But that is for a different day!

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 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 1^{st}, 5^{th}, 6^{th} and 8^{th} 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 3^{rd} to the 8^{th} 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 4^{th} and 8^{th} 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 4^{th} 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.

```
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)

```
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`

```
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

```
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

```
# 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

```
# 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).

```
# 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

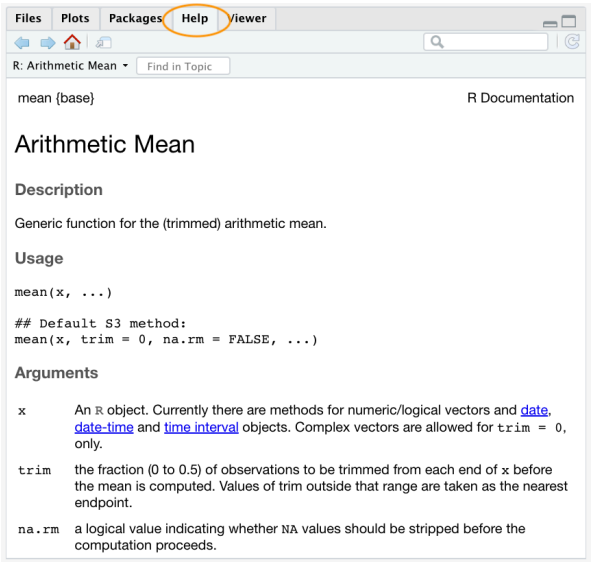
To access R's built-in help facility to get information on any function simply use the `help()` function. For example, to open the help page for our friend the `mean()` function.

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

or you can use the equivalent shortcut

```
?mean
```

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")`)

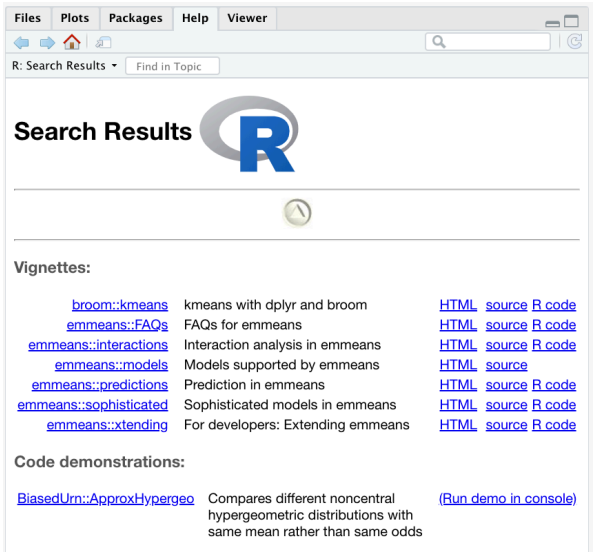
The `help()` function is useful if you know the name of the function. If you're not sure of the name, but can remember a key word then you can search R's help system using the `help.search()` function.

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

or you can use the equivalent shortcut

```
??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

```
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 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)

```
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

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

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

```
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 here. If you want to see our solutions for this exercise you can find them here (don't peek at them too soon though!).

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 Basic 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.

```
int <- 2L # Need to include L to specify integer
class(int)
## [1] "integer"

num <- 2
class(num)
## [1] "numeric"

identical(int, num) # Check to see if int and num are identical
## [1] FALSE

all.equal(int, num) # On most machines this will be true
## [1] TRUE

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).

```
# coerce numeric to character
class(num)
## [1] "numeric"
num_char <-  as.character(num)
num_char
## [1] "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 Complicated Data Types

There are lots of complicated data types in R. But the two we will be faced with frequently are time/date and factors.

3.2.1 Dealing with Date and Time

Often dates and time pose significant problems to students, because of the complicated background that undergirds them to make them legible. For example,

```
date_of_collection <- 2022-08-31

date_of_collection
## [1] 1983

class(date_of_collection)
## [1] "numeric"
```

Because - is typically interpreted as subtraction rather than as an en dash or a hyphen, R treats is as a calculator. You can get around it by enclosing in quotes and converting to character, but that looses the ability to do arithmetic

```
date_of_collection <- "2022-08-31"

class(date_of_collection)
# > [1] "Character"

date_of_collection + 2
# > Error in date_of_collection + 2 : non-numeric argument to binary operator
```

You will have to explicitly specify that you want to store the date as a type. I strongly recommend using the `lubridate` package

```
library(lubridate)
##
## Attaching package: 'lubridate'
## The following objects are masked from 'package:base':
##
##     date, intersect, setdiff, union

date_of_collection <- as_date("2022-08-31")

class(date_of_collection)
## [1] "Date"

Sys.Date() - date_of_collection
## Time difference of -17 days
```

In R, dates are represented as the number of days since 1970-01-01.

```
unclass(date_of_collection)
## [1] 19235
```

Time is even more complicated than dates, as times need to have time zones associated

with them as well as potential offsets such as daylight savings. In particular, I recommend, the `ymd_hms` type functions in lubridate to convert the character into a time object. More on this later.

```
time_of_collection <- ymd_hms("2010-12-13 15:30:30")
time_of_collection
## [1] "2010-12-13 15:30:30 UTC"


# Changes printing
with_tz(time_of_collection, "America/Los_Angeles")
## [1] "2010-12-13 07:30:30 PST"


# Changes time
force_tz(time_of_collection, "America/Chicago")
## [1] "2010-12-13 15:30:30 CST"
```

Time Zones are complicated. Over the last 100 years places have changed their affiliation between major time zones, have opted out of (or in to) daylight savings time (DST) in various years or adopted DST rule changes late or not at all. (The UK experimented with DST throughout 1971, only.) In a few countries (one is the Irish Republic) it is the summer time which is the 'standard' time and a different name is used in winter. And there can be multiple changes during a year, for example for Ramadan. These should be documented as part of the data collection and metadata processes.

3.2.2 Factors

R uses factors to handle categorical variables, variables that have a fixed and known set of possible values. Factors are also helpful for reordering character vectors to improve display. Factors are stored as integers rather than characters. For example, you can have day of the week as a factor variable.

```
day_of_week <- c('M', 'M', 'T', 'TH', 'W', 'SA', 'SU', 'TH')

(day_of_week <- factor(day_of_week))
## [1] M  M  T  TH W  SA SU TH
## Levels: M SA SU T TH W
```

Components of a factor can be modified using simple assignments. However values outside of its predefined levels are not permitted. Instead need to modify the levels first.

```
day_of_week[3] <- 'W'
```

```
day_of_week
## [1] M  M  W  TH W  SA SU TH
## Levels: M SA SU T TH W

day_of_week[4] <- 'F'
## Warning in `[<-.factor`(`*tmp*`, 4, value = "F"): invalid factor level, NA
## generated

levels(day_of_week) <- c(levels(day_of_week), "F")    # add new level
day_of_week[4] <- 'F'

str(day_of_week)
##  Factor w/ 7 levels "M","SA","SU",..: 1 1 6 7 6 2 3 5
```

Reordering the levels is often useful, especially for plotting purposes.

```
day_of_week <- c('M', 'M', 'T', 'TH', 'W', 'SA', 'SU', 'TH') %>% factor

str(day_of_week)
##  Factor w/ 6 levels "M","SA","SU",..: 1 1 4 5 6 2 3 5

day_of_week <- factor(day_of_week, levels=c("SU", "M", "T", "W", "TH", "F", "SA"))

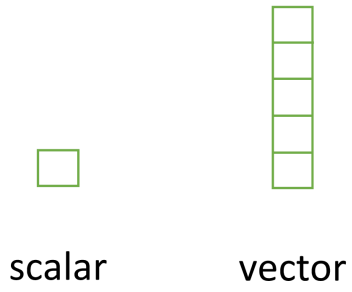
str(day_of_week)
##  Factor w/ 7 levels "SU","M","T","W",..: 2 2 3 5 4 7 1 5
```

3.3 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.

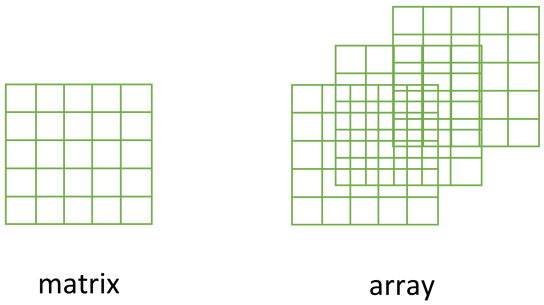
3.3.1 Scalars and vectors

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.



3.3.2 Matrices and arrays

Another useful data structure used in many disciplines 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.



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
```

3.3.3 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.

```
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

```
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

```
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
```

3.3.4 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.

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
```

3.4 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

Permit.Number	Address	Permit.Type	Residentia
	1717 Robert C Blakes, SR Dr	Short Term Rental Commercial Owner	N/A
	1006 Race St	Short Term Rental Commercial Owner	N/A
	2634 Louisiana Ave	Short Term Rental Commercial Owner	N/A
	3323 Rosalie Aly	Short Term Rental Residential Owner	Residentia
	1525 Melpomene St	Short Term Rental Residential Owner	Residentia
22-RSTR-15568	3112 Octavia St	Short Term Rental Residential Owner	Residentia
22-RSTR-14732	113 S Salcedo St	Short Term Rental Residential Owner	Residentia
22-RSTR-14693	1212 Mazant St A	Short Term Rental Residential Owner	Residentia
	2421 Chartres St	Short Term Rental Residential Owner	Residentia
22-RSTR-14696	1214 Mazant St A	Short Term Rental Residential Owner	Residentia
	814 Baronne St	Short Term Rental Commercial Owner	N/A
22-RSTR-14667	823 Mandeville St	Short Term Rental Residential Owner	Residentia

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.

As an example, the data frame below is a subset of Short Term Rental Applications and their status in Orleans Parish (City of New Orleans). It has Owner Name, Operator Name, Permit Type, Status, Expiration Date etc. In addition, it also has the number of bedrooms and occupancy limits.

In the base R, we typically use `read.csv` or `read.table` to read in an external file. Please pay attention to the file path and modify accordingly.

```
## 'data.frame':    89 obs. of  12 variables:
##  $ Permit.Number        : chr  "" "" "" "" ...
##  $ Address              : chr  "1717 Robert C Blakes, SR Dr" "1006 Race St" "2634 L
##  $ Permit.Type          : chr  "Short Term Rental Commercial Owner" "Short Term Ren
##  $ Residential.Subtype  : chr  "N/A" "N/A" "N/A" "Residential Partial Unit" ...
##  $ Current.Status       : chr  "Pending" "Pending" "Pending" "Pending" ...
##  $ Expiration.Date      : chr  "" "" "" "" ...
##  $ Bedroom.Limit        : int  5 5 3 1 3 1 2 2 1 2 ...
##  $ Guest.Occupancy.Limit: int  10 10 6 2 6 2 4 4 2 4 ...
##  $ Operator.Name        : chr  "Melissa Taranto" "Michael Heyne" "Michael Heyne" "C
##  $ License.Holder.Name  : chr  "Scott Taranto" "Boutique Hospitality" "Resonance Ho
##  $ Application.Date     : chr  "8/9/22" "8/9/22" "8/9/22" "8/9/22" ...
##  $ Issue_Date           : chr  "" "" "" "" ...
```

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.
- R tries to guess the data type based on sampling a few rows. More often than not, it gets it wrong. This is why I explicitly specified stringsAsFactors = FALSE to prevent reading them as factors. Ideally you want to store variables such as Current.Status and Permit.Type as factors and Address and Operator.Name as character.
- Check to see how the date variables such as Issue_Date and Application.Date are read and stored. `str()` is often your friend.

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

3.4.1 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). I

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 `opertator name` with `operator_name` or `operator.name`)
- Avoid using special characters (i.e. `floor area (ft^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.

3.4.2 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.csv()` function (we discuss some alternatives later in the chapter). The `read.csv()` function is a very flexible function with a shed load of arguments (see `?read.csv`) but it's quite simple to use.

Let's import a tab delimited file called `NOLA_str.txt` which contains the data we saw previously in this Chapter and assign it to an object called `str`. 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.csv()` function to import this file

```
nola_str <- read.csv(file = 'data/nola_STR.csv', header = TRUE,
                     stringsAsFactors = FALSE)
```

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/nola_STR.csv` bit) as the `read.csv()` 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.

```
nola_str <- read.csv(file = 'C:\\Documents\\Prog1\\data\\nola_STR.csv',
                     header = TRUE, stringsAsFactors = FALSE)
```

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(nola_str)
## 'data.frame':    89 obs. of  12 variables:
##  $ Permit.Number        : chr  "" "" "" "" ...
##  $ Address              : chr  "1717 Robert C Blakes, SR Dr" "1006 Race St" "2634
##  $ Permit.Type          : chr  "Short Term Rental Commercial Owner" "Short Term R
##  $ Residential.Subtype  : chr  "N/A" "N/A" "N/A" "Residential Partial Unit" ...
##  $ Current.Status       : chr  "Pending" "Pending" "Pending" "Pending" ...
##  $ Expiration.Date      : chr  "" "" "" "" ...
##  $ Bedroom.Limit        : int  5 5 3 1 3 1 2 2 1 2 ...
##  $ Guest.Occupancy.Limit: int  10 10 6 2 6 2 4 4 2 4 ...
##  $ Operator.Name        : chr  "Melissa Taranto" "Michael Heyne" "Michael Heyne"
##  $ License.Holder.Name  : chr  "Scott Taranto" "Boutique Hospitality" "Resonance
##  $ Application.Date     : chr  "8/9/22" "8/9/22" "8/9/22" "8/9/22" ...
##  $ Issue_Date           : chr  "" "" "" "" ...
```

Here we see that `nola_str` is a 'data.frame' object which contains 89 rows and 13 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.

From R version 4.0.0 you can just leave out this argument as `stringsAsFactors = FALSE` is the default.

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(str)
## NULL
```

`read.csv` is basically a thin wrapper around the workhorse function, `read.table`. 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. Similar to `read.csv()`, you can also use `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.

```
# import .csv file
nola_str <- read.csv(file = 'data/nola_STR.csv')

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

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

You can even import spreadsheet files from MS Excel or other statistics software, using packages, 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.

3.4.3 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

```
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
  cannot open file 'nola_STR.csv': 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 along way to avoiding these types of errors.

3.4.4 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 `read.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 of the things you will need to consider though is that the `fread()` function will return a `data.table` object not a `data.frame` as would be the case with the `read.table()` function. This is usually not a problem as you can pass a `data.table` object to any function that only accepts `data.frame` objects. To learn more about the differences between `data.table` and `data.frame` objects see here.

```
library(read.table)
all_data <- fread(file = 'data/nola_str.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).

In PLAN 672, we almost exclusively use functions from `readr` packages. I strongly recommend it.

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

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

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

# or use
all_data <- read_tsv(file = 'data/nola_STR.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.

3.5 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 `str` data frame we imported in the previous section.

```
nola_str <- read.table(file = 'data/nola_STR.txt', header = TRUE, sep = "\t")
## Warning in scan(file = file, what = what, sep = sep, quote = quote, dec = dec, :
## EOF within quoted string
## Warning in scan(file = file, what = what, sep = sep, quote = quote, dec = dec, :
## number of items read is not a multiple of the number of columns
str(nola_str)
## 'data.frame':    23 obs. of  12 variables:
##  $ Permit.Number        : chr  "" "" "" "" ...
##  $ Address              : chr  "1717 Robert C Blakes, SR Dr" "1006 Race St" "2634
##  $ Permit.Type          : chr  "Short Term Rental Commercial Owner" "Short Term R
##  $ Residential.Subtype  : chr  "N/A" "N/A" "N/A" "Residential Partial Unit" ...
##  $ Current.Status       : chr  "Pending" "Pending" "Pending" "Pending" ...
##  $ Expiration.Date      : chr  "" "" "" "" ...
##  $ Bedroom.Limit        : int  5 5 3 1 3 1 2 2 1 2 ...
##  $ Guest.Occupancy.Limit: int  10 10 6 2 6 2 4 4 2 4 ...
##  $ Operator.Name        : chr  "Melissa Taranto" "Michael Heyne" "Michael Heyne"
##  $ License.Holder.Name  : chr  "Scott Taranto" "Boutique Hospitality" "Resonance
##  $ Application.Date     : chr  "8/9/22" "8/9/22" "8/9/22" "8/9/22" ...
##  $ Issue_Date           : chr  "" "" "" "" ...
```

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

```
nola_str$Bedroom.Limit
##  [1]  5  5  3  1  3  1  2  2  1  2  4  2  2  3  2  1  2  1  3  1  2  1 NA
```

This will return a vector of the `Bedroom.Limit` 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_bedroom_limit <- nola_str$Bedroom.Limit
mean(f_bedroom_limit)
## [1] NA
summary(f_bedroom_limit)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   1.000   1.000   2.000   2.227   3.000   5.000       1
```

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(nola_str$Bedroom.Limit)
## [1] NA
```

```
summary(nola_str$Bedroom.Limit)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   1.000   1.000   2.000   2.227   3.000   5.000       1
```

Alternately, you can also use pipes.

```
nola_str$Bedroom.Limit |> mean()
## [1] NA
nola_str$Bedroom.Limit |> summary()
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   1.000   1.000   2.000   2.227   3.000   5.000       1
```

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.

3.5.1 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 (1^{st} row) of the `height` variable (4^{th} column)

```
nola_str[1, 4]
## [1] "N/A"

# this would give you the same
nola_str$Bedroom.Limit[1]
## [1] 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`).

```
nola_str[1:10, 1:4]
##    Permit.Number                      Address
## 1                1717 Robert C Blakes, SR Dr
## 2                             1006 Race St
## 3                        2634 Louisiana Ave
## 4                          3323 Rosalie Aly
## 5                         1525 Melpomene St
```

```
## 6  22-RSTR-15568             3112 Octavia St
## 7                           3149 Chartres St
## 8                           1952 Treasure St
## 9                         4616 S Robertson St
## 10                          3043 St Philip St
##                            Permit.Type      Residential.Subtype
## 1   Short Term Rental Commercial Owner                      N/A
## 2   Short Term Rental Commercial Owner                      N/A
## 3   Short Term Rental Commercial Owner                      N/A
## 4  Short Term Rental Residential Owner Residential Partial Unit
## 5  Short Term Rental Residential Owner   Residential Small Unit
## 6  Short Term Rental Residential Owner Residential Partial Unit
## 7                        Commercial STR                      N/A
## 8  Short Term Rental Residential Owner   Residential Small Unit
## 9  Short Term Rental Residential Owner Residential Partial Unit
## 10 Short Term Rental Residential Owner Residential Partial Unit
```

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

```
nola_str[c(1, 5, 12, 30), c(1, 3, 6, 8)]
##    Permit.Number                          Permit.Type Expiration.Date
## 1                 Short Term Rental Commercial Owner
## 5                Short Term Rental Residential Owner
## 12 22-RSTR-15454 Short Term Rental Residential Owner          8/8/23
## NA          <NA>                                <NA>            <NA>
##    Guest.Occupancy.Limit
## 1                     10
## 5                      6
## 12                     4
## NA                    NA
```

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

```
nola_str[1:8, ]
##   Permit.Number                    Address                        Permit.Type
## 1               1717 Robert C Blakes, SR Dr  Short Term Rental Commercial Owner
## 2                           1006 Race St  Short Term Rental Commercial Owner
## 3                      2634 Louisiana Ave  Short Term Rental Commercial Owner
## 4                        3323 Rosalie Aly Short Term Rental Residential Owner
## 5                       1525 Melpomene St Short Term Rental Residential Owner
## 6 22-RSTR-15568           3112 Octavia St Short Term Rental Residential Owner
## 7                        3149 Chartres St                      Commercial STR
## 8                        1952 Treasure St Short Term Rental Residential Owner
##       Residential.Subtype Current.Status Expiration.Date Bedroom.Limit
## 1                     N/A        Pending                             5
## 2                     N/A        Pending                             5
## 3                     N/A        Pending                             3
## 4 Residential Partial Unit        Pending                             1
## 5   Residential Small Unit        Pending                             3
## 6 Residential Partial Unit         Issued          8/4/23             1
## 7                     N/A        Pending                             2
## 8   Residential Small Unit        Pending                             2
##   Guest.Occupancy.Limit         Operator.Name    License.Holder.Name
## 1                    10       Melissa Taranto          Scott Taranto
## 2                    10         Michael Heyne   Boutique Hospitality
## 3                     6         Michael Heyne     Resonance Home LLC
## 4                     2         Caroline Stas          Caroline Stas
## 5                     6        Craig Redgrave       Craig R Redgrave
## 6                     2       Philip Wheeler Philip Barrett Wheeler
## 7                     4 Bruce Michael Ferweda          Bruce Ferweda
## 8                     4         Zedrick Price        Zedrick L Price
##   Application.Date Issue_Date
## 1           8/9/22
## 2           8/9/22
## 3           8/9/22
## 4           8/9/22
## 5           8/8/22
## 6           8/5/22     8/5/22
## 7           8/5/22
## 8           8/5/22
```

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.

```
nola_str[, 1:3]
```

```
##    Permit.Number
```

```
## 1
## 2
## 3
## 4
## 5
## 6  22-RSTR-15568
## 7
## 8
## 9
## 10
## 14
## 15
## 16
## 17 22-CSTR-10184
## 18
## 19
## 20
## 21
## 22
## 23 22-RSTR-15236
##
## 1
## 2
## 3
## 4
## 5
## 6
## 7
## 8
## 9
## 10
## 14
## 15
## 16
## 17
## 18
## 19
## 20
## 21
## 22
## 23 4320 Dhemecourt St\tShort Term Rental Residential Owner\tResidential Partial Uni
##                            Permit.Type
## 1   Short Term Rental Commercial Owner
## 2   Short Term Rental Commercial Owner
## 3   Short Term Rental Commercial Owner
## 4  Short Term Rental Residential Owner
## 5  Short Term Rental Residential Owner
```

```
## 6  Short Term Rental Residential Owner
## 7                       Commercial STR
## 8  Short Term Rental Residential Owner
## 9  Short Term Rental Residential Owner
## 10 Short Term Rental Residential Owner
## 14  Short Term Rental Commercial Owner
## 15  Short Term Rental Commercial Owner
## 16 Short Term Rental Residential Owner
## 17  Short Term Rental Commercial Owner
## 18 Short Term Rental Residential Owner
## 19 Short Term Rental Residential Owner
## 20  Short Term Rental Commercial Owner
## 21  Short Term Rental Commercial Owner
## 22  Short Term Rental Commercial Owner
## 23
```

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 4^{th}, 7^{th} and 8^{th} 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`).

```
nola_str[-(1:85), -c(4, 7, 8)]
## [1] Permit.Number       Address             Permit.Type
## [4] Current.Status      Expiration.Date     Operator.Name
## [7] License.Holder.Name Application.Date    Issue_Date
## <0 rows> (or 0-length row.names)
```

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 `str[1:5, c(1, 2, 6)]` we can instead use

```
nola_str[1:5, c("Operator.Name", "License.Holder.Name", "Application.Date")]
##     Operator.Name  License.Holder.Name Application.Date
## 1 Melissa Taranto        Scott Taranto           8/9/22
## 2   Michael Heyne Boutique Hospitality           8/9/22
## 3   Michael Heyne   Resonance Home LLC           8/9/22
## 4   Caroline Stas        Caroline Stas           8/9/22
## 5  Craig Redgrave     Craig R Redgrave           8/8/22
```

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.

3.5.2 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 `Bedroom.Limit` is greater than 3 and extract all columns by default (remember, if you don't include a column index after the comma it means all columns).

```
big_str <- nola_str[nola_str$Bedroom.Limit > 3, ]
big_str
##    Permit.Number                      Address                        Permit.Type
## 1                1717 Robert C Blakes, SR Dr Short Term Rental Commercial Owner
## 2                             1006 Race St Short Term Rental Commercial Owner
## 11              1727 Henriette Delille St Short Term Rental Commercial Owner
## NA          <NA>                        <NA>                              <NA>
##    Residential.Subtype Current.Status Expiration.Date Bedroom.Limit
## 1                  N/A        Pending                            5
## 2                  N/A        Pending                            5
## 11                 N/A        Pending                            4
## NA                <NA>           <NA>            <NA>           NA
##    Guest.Occupancy.Limit   Operator.Name  License.Holder.Name Application.Date
## 1                     10 Melissa Taranto        Scott Taranto           8/9/22
## 2                     10   Michael Heyne Boutique Hospitality           8/9/22
## 11                     8   Jane Chaisson   David Trocquet Jr.           8/4/22
## NA                    NA            <NA>                 <NA>             <NA>
##    Issue_Date
## 1
## 2
## 11
## NA       <NA>
```

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

```
big_str <- nola_str[Bedroom.Limit > 3, ]
Error in `[.data.frame`(nola_str, Bedroom.Limit > 3, ) :
  object 'Bedroom.Limit' not found
```

So how does this work? The logical test is `nola_str$Bedroom.Limit > 3` 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 `Bedroom.Limit`

is greater than 3 and `FALSE` if `Bedroom.Limit` is not greater than 3.

```
nola_str$Bedroom.Limit > 3
##  [1]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE    NA
```

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

```
nola_str[nola_str$Bedroom.Limit >= 3, ]       # values greater or equal to 3

nola_str[nola_str$Bedroom.Limit <= 3, ]        # values less than or equal to 3

nola_str[nola_str$Bedroom.Limit <= 4, ]       # values  equal to 4

nola_str[nola_str$Bedroom.Limit != 4, ]        # values  not equal to 4
```

We can also extract rows based on the value of a character string or factor level. Let's extract all rows where the `Operator.Name` is equal to `Michael Heyne` (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. `"Michael Heyne"`).

```
k <- nola_str[nola_str$Operator.Name  == "Michael Heyne", ]
k
```

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

```
nola_str_notPending <- nola_str[nola_str$Current.Status  != "Pending", 1:4]
nola_str_notPending
```

```
##     Permit.Number
## 6   22-RSTR-15568
## 12  22-RSTR-15454
## 17  22-CSTR-10184
## 23  22-RSTR-15236
## 61  22-RSTR-15568
## 121 22-RSTR-15454
## 171 22-CSTR-10184
## 231 22-RSTR-15236
##
## 6
```

```
## 12
## 17
## 23  4320 Dhemecourt St\tShort Term Rental Residential Owner\tResidential Partial Un
## 61
## 121
## 171
## 231 4320 Dhemecourt St\tShort Term Rental Residential Owner\tResidential Partial Un
##                               Permit.Type      Residential.Subtype
## 6   Short Term Rental Residential Owner Residential Partial Unit
## 12  Short Term Rental Residential Owner   Residential Large Unit
## 17   Short Term Rental Commercial Owner                     N/A
## 23
## 61  Short Term Rental Residential Owner Residential Partial Unit
## 121 Short Term Rental Residential Owner   Residential Large Unit
## 171  Short Term Rental Commercial Owner                     N/A
## 231
```

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 `Bedroom.Limit` is greater or equal to `3` AND `Current.Status` is equal to `Pending` AND `Permit.Type` is equal to `Short Term Rental Residential Owner"` we combine a series of logical expressions with the `&` symbol.

```
k2 <- nola_str[nola_str$Bedroom.Limit >= 3 &
                               nola_str$Current.Status == "Pending" &
                              nola_str$Permit.Type == "Short Term Rental Residential Ow
                              ]
k2[,1:5]
##    Permit.Number           Address                          Permit.Type
## 5                1525 Melpomene St Short Term Rental Residential Owner
## 19                2000 Barracks St Short Term Rental Residential Owner
##         Residential.Subtype Current.Status
## 5    Residential Small Unit        Pending
## 19 Residential Partial Unit        Pending
```

To extract rows based on an 'OR' Boolean expression we can use the `|` symbol. Don't forget the `,` to specify and extract the column indices.

```
k3 <- nola_str[nola_str$Bedroom.Limit >= 3 |
                               nola_str$Current.Status != "Pending",
                              1:3]

k3
##    Permit.Number
## 1
## 2
```

```
## 3
## 5
## 6  22-RSTR-15568
## 11
## 12 22-RSTR-15454
## 14
## 17 22-CSTR-10184
## 19
## 23 22-RSTR-15236
##
## 1
## 2
## 3
## 5
## 6
## 11
## 12
## 14
## 17
## 19
## 23 4320 Dhemecourt St\tShort Term Rental Residential Owner\tResidential Partial U
##                              Permit.Type
## 1   Short Term Rental Commercial Owner
## 2   Short Term Rental Commercial Owner
## 3   Short Term Rental Commercial Owner
## 5  Short Term Rental Residential Owner
## 6  Short Term Rental Residential Owner
## 11  Short Term Rental Commercial Owner
## 12 Short Term Rental Residential Owner
## 14  Short Term Rental Commercial Owner
## 17  Short Term Rental Commercial Owner
## 19 Short Term Rental Residential Owner
## 23
```

3.5.3 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
```

3.6 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 `str` data frame using the `summary()` function.

```
nola_str$Permit.Type <- factor(nola_str$Permit.Type)
nola_str$Current.Status <- factor(nola_str$Current.Status)
summary(nola_str)
##  Permit.Number        Address                                        Permit.Type
##  Length:23          Length:23                                             : 1
##  Class :character   Class :character   Commercial STR                      : 1
##  Mode  :character   Mode  :character   Short Term Rental Commercial Owner :10
##                                        Short Term Rental Residential Owner:11
##
##
##
##  Residential.Subtype Current.Status Expiration.Date    Bedroom.Limit
##  Length:23                  : 1     Length:23          Min.   :1.000
##  Class :character    Issued : 3     Class :character   1st Qu.:1.000
##  Mode  :character    Pending:19     Mode  :character   Median :2.000
##                                                        Mean   :2.227
##                                                        3rd Qu.:3.000
##                                                        Max.   :5.000
##                                                        NA's   :1
##  Guest.Occupancy.Limit Operator.Name      License.Holder.Name
##  Min.   : 2.000        Length:23          Length:23
##  1st Qu.: 2.000        Class :character   Class :character
##  Median : 4.000        Mode  :character   Mode  :character
##  Mean   : 4.455
```

```
##  3rd Qu.: 6.000
##  Max.   :10.000
##  NA's   :1
##  Application.Date    Issue_Date
##  Length:23          Length:23
##  Class :character   Class :character
##  Mode  :character   Mode  :character
##
##
##
##
```

For numeric variables the mean, minimum, maximum, median, first (lower) quartile and third (upper) quartile are presented. For factor variables (i.e. `Permit.Type` and `current.status`) 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. For character variables, only length of the vector is 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. Notice we include all rows by not specifying a row index.

```
summary(nola_str[, 4:7])
##  Residential.Subtype Current.Status Expiration.Date    Bedroom.Limit
##  Length:23                   : 1    Length:23          Min.   :1.000
##  Class :character    Issued : 3    Class :character   1st Qu.:1.000
##  Mode  :character    Pending:19    Mode  :character   Median :2.000
##                                                       Mean   :2.227
##                                                       3rd Qu.:3.000
##                                                       Max.   :5.000
##                                                       NA's   :1
```

And to summarise a single variable.

```
summary(nola_str$Permit.Type)
##                                                    Commercial STR
##                                 1                                1
##  Short Term Rental Commercial Owner Short Term Rental Residential Owner
##                                10                               11
```

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 `Permit.Type`

```
table(nola_str$Permit.Type)
##
##                                                    Commercial STR
##                                        1                         1
##  Short Term Rental Commercial Owner Short Term Rental Residential Owner
##                                       10                        11
```

We can extend this further by producing a table of counts for each combination of `Permit.Type` and `Current.Status` factor levels.

```
table(nola_str$Permit.Type, nola_str$Current.Status)
##
##                                      Issued Pending
##                                   1      0       0
##   Commercial STR                  0      0       1
##   Short Term Rental Commercial Owner  0      1       9
##   Short Term Rental Residential Owner 0      2       9
```

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(~ Permit.Type + Current.Status, data = nola_str)
##                                    Current.Status
## Permit.Type                           Issued Pending
##                                   1      0       0
##   Commercial STR                  0      0       1
##   Short Term Rental Commercial Owner  0      1       9
##   Short Term Rental Residential Owner 0      2       9
```

3.7 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.

3.8 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 here. If you want to see our solutions for this exercise you can find them here (don't peek at them too soon though!).

Chapter 4

Tidyverse

The tidyverse is a collection of related R packages developed to streamline data management, munging and analysis in R. The core tidyverse packages include `ggplot2`, `dplyr`, `tidyr`, `lubridate`, `readr`, `purrr`, `tibble`, `stringr`, and `forcats`. In this chapter, we will give an introduction to the philosophy of tidyverse packages to get you started. We will use these packages extensively though out the urban analytics course.

4.1 Data Frames & Tibbles

We already saw how rectangular data is stored in data frames in the previous Chapter. In contrast, Tidyverse uses tibbles.

> A tibble, or tbl_df, is a modern reimagining of the data.frame, keeping what time has proven to be effective, and throwing out what is not. Tibbles are data.frames that are lazy and surly: they do less (i.e. they don't change variable names or types, and don't do partial matching) and complain more (e.g. when a variable does not exist). This forces you to confront problems earlier, typically leading to cleaner, more expressive code. — http://tibble.tidyverse.org

```
library(tidyverse)
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
## v ggplot2 3.3.6     v purrr   0.3.4
## v tibble  3.1.7     v dplyr   1.0.9
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x lubridate::as.difftime() masks base::as.difftime()
## x lubridate::date()        masks base::date()
## x tidyr::extract()         masks magrittr::extract()
## x dplyr::filter()          masks stats::filter()
## x dplyr::group_rows()      masks kableExtra::group_rows()
```

```
## x lubridate::intersect()   masks base::intersect()
## x dplyr::lag()             masks stats::lag()
## x purrr::set_names()       masks magrittr::set_names()
## x lubridate::setdiff()     masks base::setdiff()
## x lubridate::union()       masks base::union()

data_tib <- tibble(`alphabet soup` = letters,
       `nums ints` = 1:26,
        `sample ints`  = sample(100, 26))

data_df <- data.frame(`alphabet soup` = letters,
       `nums ints` = 1:26,
        `sample ints`  = sample(100, 26))

glimpse(data_tib)
## Rows: 26
## Columns: 3
## $ `alphabet soup` <chr> "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k",~
## $ `nums ints`     <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,~
## $ `sample ints`   <int> 95, 99, 18, 72, 93, 79, 53, 91, 41, 90, 58, 28, 46, 52~
glimpse(data_df)
## Rows: 26
## Columns: 3
## $ alphabet.soup <chr> "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "~
## $ nums.ints     <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1~
## $ sample.ints   <int> 88, 82, 81, 47, 75, 95, 46, 5, 86, 92, 27, 79, 83, 100, ~

# Notice the use of glimpse instead of str. They are very similar functions.
```

Notice how data.frame changes the names of the variables because it does not like spaces in the column names. One advantage of tibbles is that columns need not be valid R variable names as long as they are enclosed in ticks.

You can use base R functions to work with tibbles, because tibble is indeed a data frame. However, functions based on tibbles may not work with data frames.

```
data_tib[,3]
## # A tibble: 26 x 1
##    `sample ints`
##            <int>
##  1            95
##  2            99
##  3            18
##  4            72
```

```
##  5            93
##  6            79
##  7            53
##  8            91
##  9            41
## 10            90
## # ... with 16 more rows

data_tib[2:4,1:3]
## # A tibble: 3 x 3
##   `alphabet soup` `nums ints` `sample ints`
##   <chr>                 <int>         <int>
## 1 b                         2            99
## 2 c                         3            18
## 3 d                         4            72
```

4.2 Reading external data

`readr` package, part of tidyverse reads flat files. Most of readr's functions are concerned with turning flat files into tibbles. In particular, * `read_csv` reads comma delimited files * `read_csv2` reads semicolon separated files (common in countries where , is used as the decimal place) * `read_tsv` reads tab delimited files * `read_delim` reads in files with any delimiter.

Notice the _ in the functions compared to . in base R.

```
library(tidyverse)

nola_str_tib <- read_csv('data/NOLA_STR.csv')
## Rows: 89 Columns: 12
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (10): Permit Number, Address, Permit Type, Residential Subtype, Current ...
## dbl  (2): Bedroom Limit, Guest Occupancy Limit
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

nola_str_tib
## # A tibble: 89 x 12
##    `Permit Number` Address       `Permit Type` `Residential S~` `Current Status`
##    <chr>           <chr>         <chr>         <chr>            <chr>
##  1 <NA>            1717 Robert ~ Short Term R~ N/A              Pending
##  2 <NA>            1006 Race St  Short Term R~ N/A              Pending
```

```
##  3 <NA>            2634 Louisia~ Short Term R~ N/A              Pending
##  4 <NA>            3323 Rosalie~ Short Term R~ Residential Par~ Pending
##  5 <NA>            1525 Melpome~ Short Term R~ Residential Sma~ Pending
##  6 22-RSTR-15568   3112 Octavia~ Short Term R~ Residential Par~ Issued
##  7 <NA>            3149 Chartre~ Commercial S~ N/A              Pending
##  8 <NA>            1952 Treasur~ Short Term R~ Residential Sma~ Pending
##  9 <NA>            4616 S Rober~ Short Term R~ Residential Par~ Pending
## 10 <NA>            3043 St Phil~ Short Term R~ Residential Par~ Pending
## # ... with 79 more rows, and 7 more variables: `Expiration Date` <chr>,
## #   `Bedroom Limit` <dbl>, `Guest Occupancy Limit` <dbl>,
## #   `Operator Name` <chr>, `License Holder Name` <chr>,
## #   `Application Date` <chr>, Issue_Date <chr>
```

Sometimes there are a few lines of metadata at the top of the file. You can use `skip = n` to skip the first n lines; or use `comment = "#"` to drop all lines that start with (e.g.) #.

By default, `read_csv` assumes that the first line is a header. If this is not the case, use `col_names = FALSE` or pass a character vector to `col_names`

`readr` uses a heuristic to figure out the type of each column: it reads the first 1000 rows and uses some (moderately conservative) heuristics to figure out the type of each column. In large data files the first 1000 rows may not sufficiently general, so you might have to specify the `col_types`.

```
nola_str_tib <- read_csv('data/NOLA_STR.csv',
                                                    col_types = cols(
                                                       `Permit Number` = col_character(),
                                                       `Address` = col_character(),
                                                       `Permit Type` = col_factor(),
                                                       `Application Date` = col_date(form
                                                       `Issue_Date` = col_date(format = '
                                                    ))

str(nola_str_tib)
## spec_tbl_df [89 x 12] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ Permit Number        : chr [1:89] NA NA NA NA ...
##  $ Address              : chr [1:89] "1717 Robert C Blakes, SR Dr" "1006 Race St"
##  $ Permit Type          : Factor w/ 3 levels "Short Term Rental Commercial Owner"
##  $ Residential Subtype  : chr [1:89] "N/A" "N/A" "N/A" "Residential Partial Unit"
##  $ Current Status       : chr [1:89] "Pending" "Pending" "Pending" "Pending" ...
##  $ Expiration Date      : chr [1:89] NA NA NA NA ...
##  $ Bedroom Limit        : num [1:89] 5 5 3 1 3 1 2 2 1 2 ...
##  $ Guest Occupancy Limit: num [1:89] 10 10 6 2 6 2 4 4 2 4 ...
##  $ Operator Name        : chr [1:89] "Melissa Taranto" "Michael Heyne" "Michael H
```

```
##  $ License Holder Name  : chr [1:89] "Scott Taranto" "Boutique Hospitality" "Reso
##  $ Application Date     : Date[1:89], format: "2022-08-09" "2022-08-09" ...
##  $ Issue_Date           : Date[1:89], format: NA NA ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   `Permit Number` = col_character(),
##   ..   Address = col_character(),
##   ..   `Permit Type` = col_factor(levels = NULL, ordered = FALSE, include_na = FA
##   ..   `Residential Subtype` = col_character(),
##   ..   `Current Status` = col_character(),
##   ..   `Expiration Date` = col_character(),
##   ..   `Bedroom Limit` = col_double(),
##   ..   `Guest Occupancy Limit` = col_double(),
##   ..   `Operator Name` = col_character(),
##   ..   `License Holder Name` = col_character(),
##   ..   `Application Date` = col_date(format = "%m/%d/%y"),
##   ..   Issue_Date = col_date(format = "%m/%d/%y")
##   .. )
##  - attr(*, "problems")=<externalptr>
```

Notice how `Permit Type` is now read in as a factor instead of a character.

4.3 Wrangling Data

`dplyr` is the workhorse package for exploratory data analysis. In particular, `select`, `filter` and `mutate` are useful.

- `select` is used to select on columns.
- `filter` is used to select on rows.
- `mutate` changes/adds columns

```
nola_str_tib %>%
    select(c(`Permit Type`, `Residential Subtype`))
## # A tibble: 89 x 2
##    `Permit Type`                       `Residential Subtype`
##    <fct>                               <chr>
##  1 Short Term Rental Commercial Owner  N/A
##  2 Short Term Rental Commercial Owner  N/A
##  3 Short Term Rental Commercial Owner  N/A
##  4 Short Term Rental Residential Owner Residential Partial Unit
##  5 Short Term Rental Residential Owner Residential Small Unit
##  6 Short Term Rental Residential Owner Residential Partial Unit
```

```
##  7 Commercial STR                        N/A
##  8 Short Term Rental Residential Owner Residential Small Unit
##  9 Short Term Rental Residential Owner Residential Partial Unit
## 10 Short Term Rental Residential Owner Residential Partial Unit
## # ... with 79 more rows
```

```
nola_str_tib %>%
    filter(`Permit Type` == "Short Term Rental Residential Owner")
```

```
## # A tibble: 53 x 12
##    `Permit Number` Address       `Permit Type` `Residential S~` `Current Status`
##    <chr>           <chr>         <fct>         <chr>            <chr>
##  1 <NA>            3323 Rosalie~ Short Term R~ Residential Par~ Pending
##  2 <NA>            1525 Melpome~ Short Term R~ Residential Sma~ Pending
##  3 22-RSTR-15568   3112 Octavia~ Short Term R~ Residential Par~ Issued
##  4 <NA>            1952 Treasur~ Short Term R~ Residential Sma~ Pending
##  5 <NA>            4616 S Rober~ Short Term R~ Residential Par~ Pending
##  6 <NA>            3043 St Phil~ Short Term R~ Residential Par~ Pending
##  7 22-RSTR-15454   1731 Third S~ Short Term R~ Residential Lar~ Issued
##  8 <NA>            1731 Third S~ Short Term R~ Residential Lar~ Pending
##  9 <NA>            4632 Frankli~ Short Term R~ Residential Par~ Pending
## 10 <NA>            621 Desire St Short Term R~ Residential Par~ Pending
## # ... with 43 more rows, and 7 more variables: `Expiration Date` <chr>,
## #   `Bedroom Limit` <dbl>, `Guest Occupancy Limit` <dbl>,
## #   `Operator Name` <chr>, `License Holder Name` <chr>,
## #   `Application Date` <date>, Issue_Date <date>
```

```
library(lubridate)

nola_str_tib %>%
    mutate(Backlogged = if_else(
                                                        (today() - `Application Da
                                                            T, F)
                                                        ) %>%
    filter(Backlogged == T) %>%
    select(`Address`, `Operator Name`, `License Holder Name`, `Application Date`)
```

```
## # A tibble: 25 x 4
##    Address               `Operator Name`  `License Holder Name` `Application D~`
##    <chr>                 <chr>            <chr>                 <date>
##  1 11416 Prentiss Ave    April Jenkins    April Jenkins         2022-07-30
##  2 1029 Montegut St      Bettina Reutter  Bywaterbeauty guest ~ 2022-07-30
##  3 1743 N Rocheblave St  Danielle Wright  Danielle D Wright     2022-07-29
##  4 862 Tchoupitoulas St  Carter Kronlage  Tchoup N Block LLC    2022-07-29
##  5 813 Jackson Ave       Michael Springer 813 Jackson Ave       2022-07-29
```

```
##  6 2312 St Louis St      Michael Springer P Rubenstein William  2022-07-28
##  7 734 Union St Unit 401 Devrim Hayes     Unique Union Condos ~ 2022-07-27
##  8 1028 Octavia St       Claire Stockton  Claire Stockton       2022-07-27
##  9 7810 Spruce St        Nancy Caddigan   Nancy Caddigan        2022-07-27
## 10 7645 Forum Blvd       AUSTIN LEVY      Austin Levy           2022-07-26
## # ... with 15 more rows
```

Few things to notice here.

- There is no need to specify the tibble using `$` to access the column names. `dplyr` assumes that you are working with the dataset that you specified earlier in the pipe. This makes the code cleaner.
- When there are no spaces in the column names you can omit the ticks.
- You can chain a number of functions as in `mutate`, `filter` and `select` to do complicated analyses in a simple way using pipe operator `%>%`.
- We also used `if_else` to create a logical variable called `Backlogged`. Notice that mutate automatically creates and populates this new column. Also notice the date arithmetic as well as logical operator `&`. Be careful to make sure that the new column is the same length as the rest of the dataset.
- Notice the use of `(today()...)`. They are used to improve readability of code but also to specify the order of operations.

4.4 Relational Data

It is rarely the case that we will be working with one table. Often we need to join multiple tables together because information is scattered in different databases. Typically the relationships are defined using at set of columns.

Suppose there is a table that includes the license number of the operator and we want to include it into the tibble `nola_str_tib`.

```
nola_operators <- read_csv("data/NOLA_STR_operators.csv")
## Rows: 59 Columns: 2
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): Operator Name, Operator Permit Number
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

First notice that `Operator Name` is the same between the tables. Presumably this is what we can use to join the tables.

Notice that there are only 59 operators in the table compared to 89 rows in `nola_str_tib`. Either there are duplicates in `nola_str_tib` or there are missing operators in `nola_operators`. It could also very well be the case that they are in different order than `nola_str_tib`. In any case, we cannot simply take the column from `nola_operators` and column bind it. We have to use `*_join()` type of functions to join the datasets together.

```
nola_str_tib_join <- left_join(nola_str_tib, nola_operators,
                                                                by=c("Operator Name" = "O
                                                                )

str(nola_str_tib_join)
## spec_tbl_df [89 x 13] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ Permit Number         : chr [1:89] NA NA NA NA ...
##  $ Address               : chr [1:89] "1717 Robert C Blakes, SR Dr" "1006 Race St
##  $ Permit Type           : Factor w/ 3 levels "Short Term Rental Commercial Owner
##  $ Residential Subtype   : chr [1:89] "N/A" "N/A" "N/A" "Residential Partial Unit
##  $ Current Status        : chr [1:89] "Pending" "Pending" "Pending" "Pending" ...
##  $ Expiration Date       : chr [1:89] NA NA NA NA ...
##  $ Bedroom Limit         : num [1:89] 5 5 3 1 3 1 2 2 1 2 ...
##  $ Guest Occupancy Limit : num [1:89] 10 10 6 2 6 2 4 4 2 4 ...
##  $ Operator Name         : chr [1:89] "Melissa Taranto" "Michael Heyne" "Michael
##  $ License Holder Name   : chr [1:89] "Scott Taranto" "Boutique Hospitality" "Res
##  $ Application Date      : Date[1:89], format: "2022-08-09" "2022-08-09" ...
##  $ Issue_Date            : Date[1:89], format: NA NA ...
##  $ Operator Permit Number: chr [1:89] "20-ostr-01377" "20-OSTR-30075" "20-OSTR-30
##  - attr(*, "spec")=
##   .. cols(
##   ..   `Permit Number` = col_character(),
##   ..   Address = col_character(),
##   ..   `Permit Type` = col_factor(levels = NULL, ordered = FALSE, include_na = FA
##   ..   `Residential Subtype` = col_character(),
##   ..   `Current Status` = col_character(),
##   ..   `Expiration Date` = col_character(),
##   ..   `Bedroom Limit` = col_double(),
##   ..   `Guest Occupancy Limit` = col_double(),
##   ..   `Operator Name` = col_character(),
##   ..   `License Holder Name` = col_character(),
##   ..   `Application Date` = col_date(format = "%m/%d/%y"),
##   ..   Issue_Date = col_date(format = "%m/%d/%y")
##   .. )
##  - attr(*, "problems")=<externalptr>
```

Couple of things to notice here

- `left_join` keeps all the rows in the first table. So `Operator Permit Number`

should have the same length as the rest of the table even if some are NA.

- Notice the use “ ” in the `by` rather than “. This is because the matching is based on strings rather than names. This is a subtle difference and would occasionally trip students up.

4.5 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 here. If you want to see our solutions for this exercise you can find them here (don't peek at them too soon though!).

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.

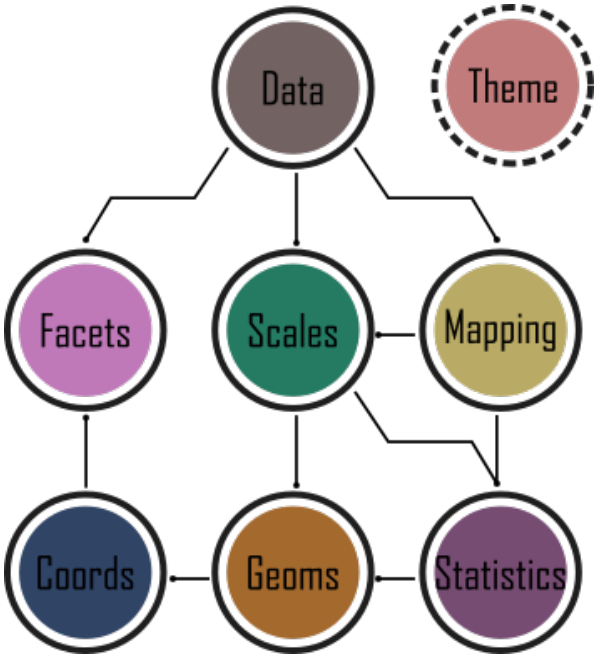


Figure 5.1: framework behind ggplot2

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.

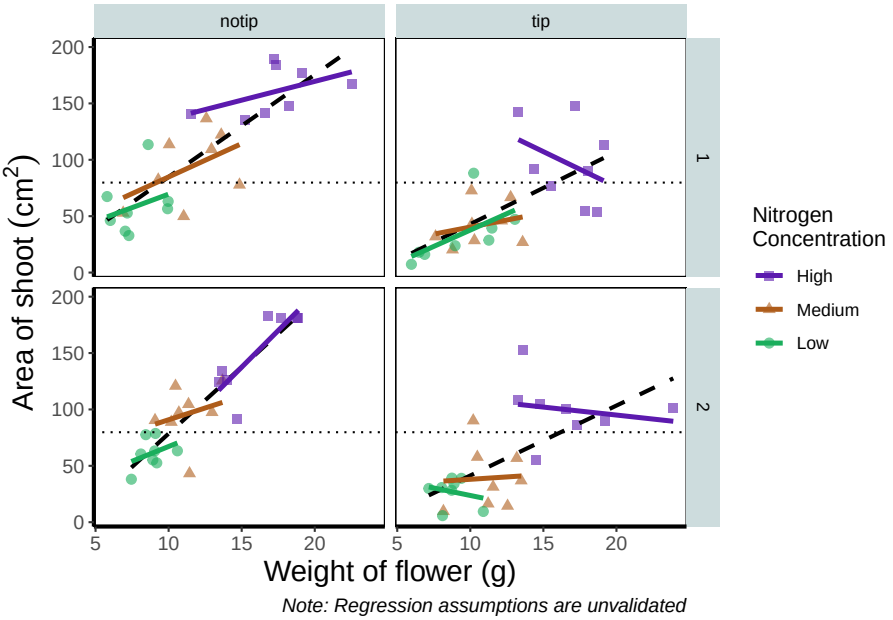
It's important to note that `ggplot2` is not **required** to make "fancy" and informative figures in R. If you prefer using base R graphics then feel free to continue as almost all `ggplot2` type figures can be created using base R (we often use either approach depending on what we're doing). The difference 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?



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`.

```
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:

```
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 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.

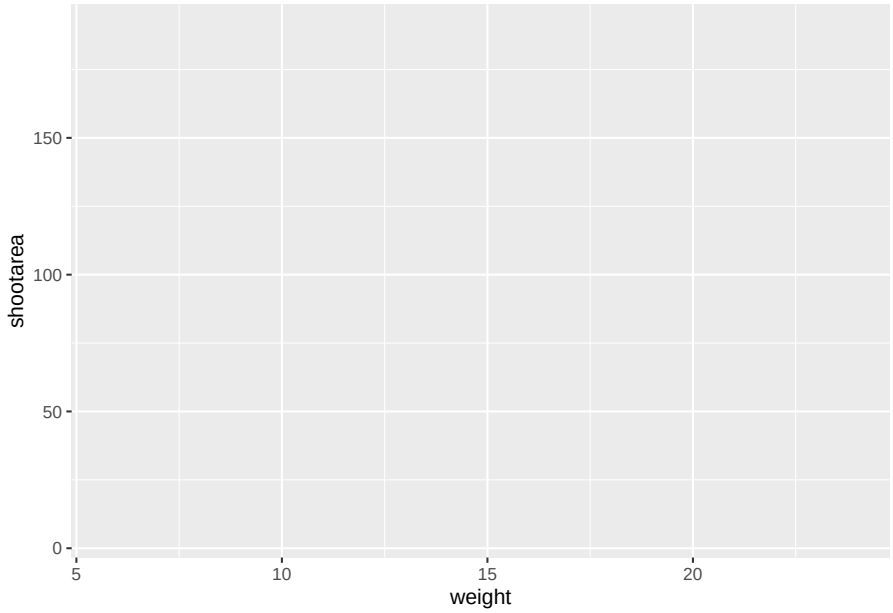
If we did not include the `aes()` function, then the `x =` and `y =` arguments would produce an error saying that the object was not found. A good rule to keep in mind when using `ggplot2` is that the variables which we want displayed on the figure must be included

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.

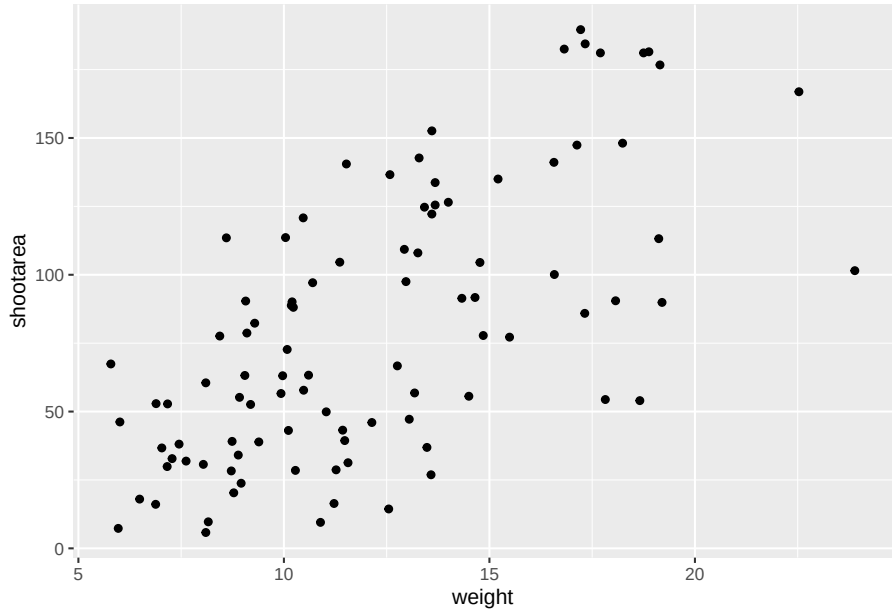
Geometries are the way that `ggplot2` displays information. For instance `geom_point()` tells `ggplot2` that you want the information to be displayed as points (making scatterplots possible for example). Given that the "final figure" uses points, this is clearly the appropriate geom to use here.

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

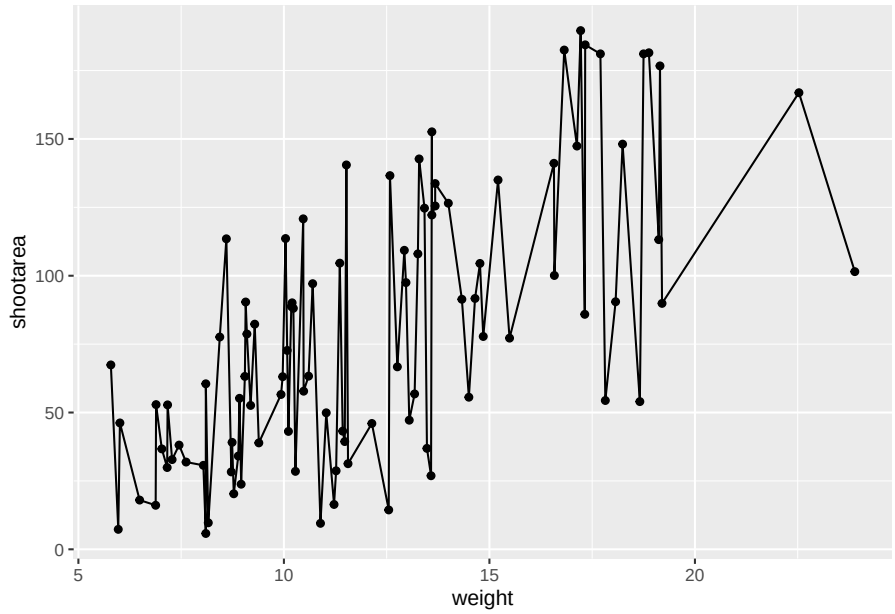
Figure 5.4: Geometry - how we want that data displayed

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).

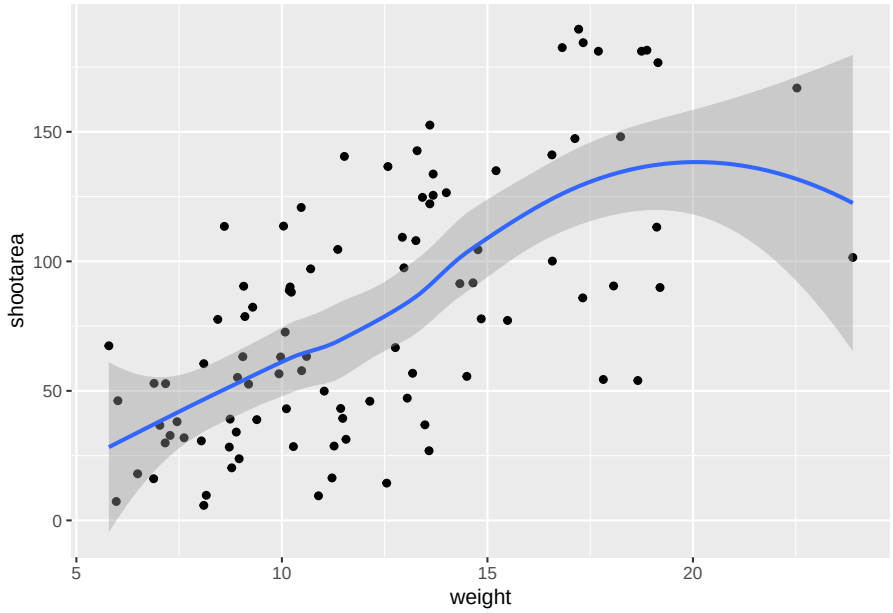
If what we wanted was a quick and dirty figure to get a grasp of the trend in the data we can stop here. From the scatterplot that we've produced, we can see that `shootarea` looks like it's increasing with `weight` in a linear fashion. So long as this answers the question we were asking from these data, 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][geoms], 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
```

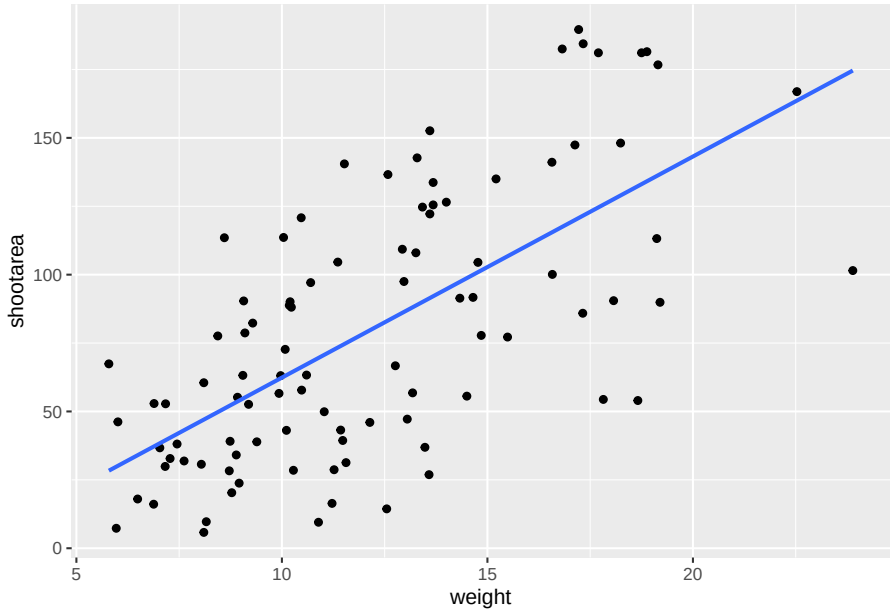


Better, but still not what we wanted. The challenge here is that drawing a line is actually somewhat complicated. The way our line above was drawn was by using a method called "LOESS" (locally estimated scatterplot smoothing) which gives something very close to a moving average; useful in some cases, 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.

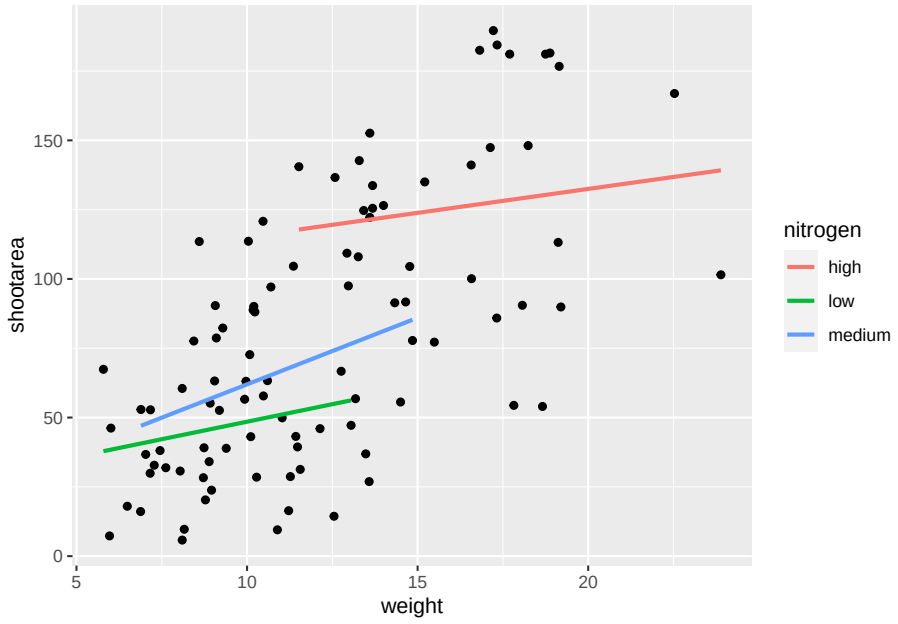

```
ggplot(aes(x = weight, y = shootarea), data = flower) +
  geom_point() +
    geom_smooth(method = "lm", se = FALSE)     # method and se
```



We get the straight line that we wanted, though it's still not matching the "final figure". 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.

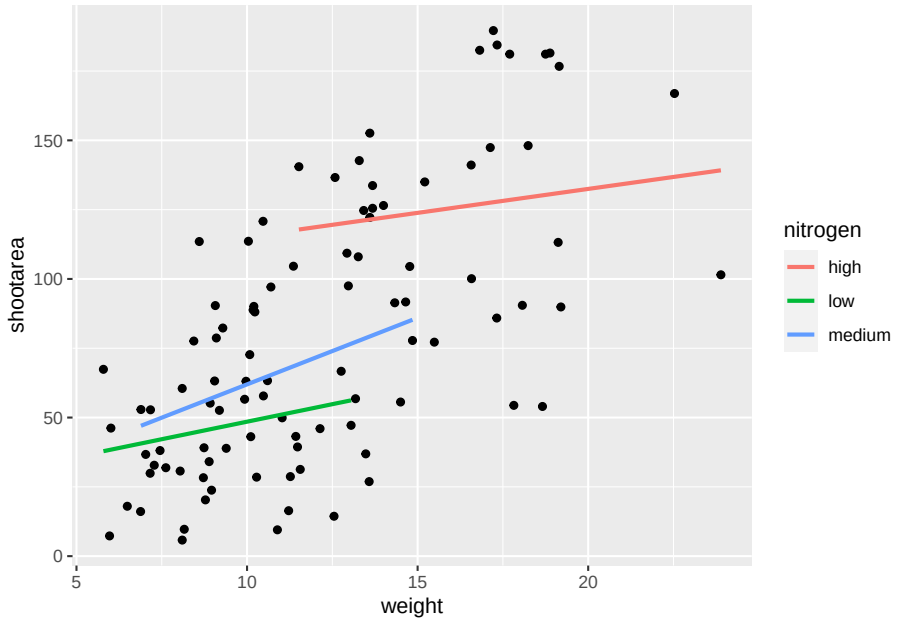
An aside: `ggplot2` was written with both UK English and American English in mind, so both `colour` and `color` spellings work in `ggplot2`.


```
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.

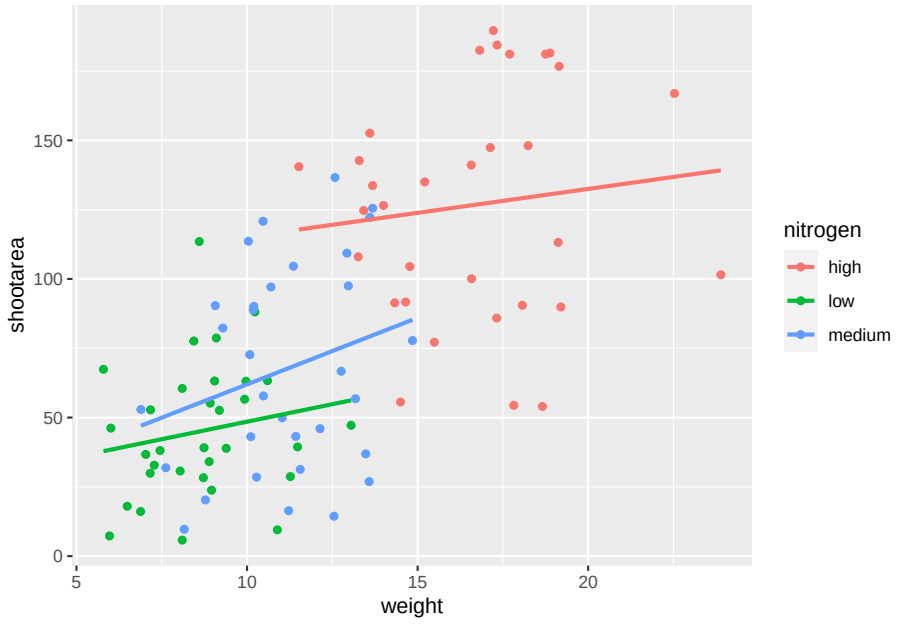

```
ggplot() +
    # Moved aes() and data into geoms
  geom_point(aes(x = weight, y = shootarea), data = flower) +
  geom_smooth(aes(x = weight, y = shootarea, colour = nitrogen),
                        data = flower, method = "lm", se = FALSE)
```



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".

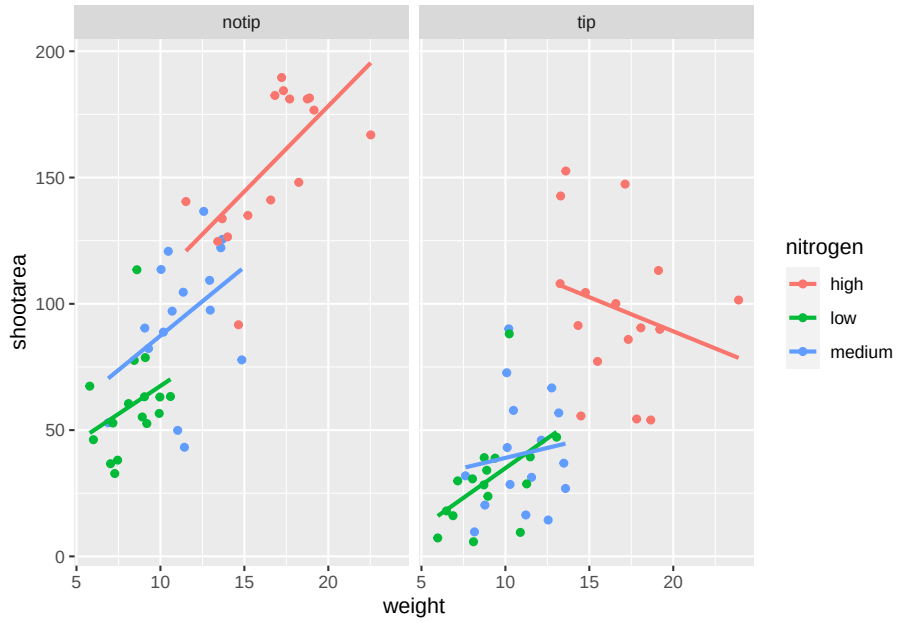
5.2.2 Wrapping grids

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:

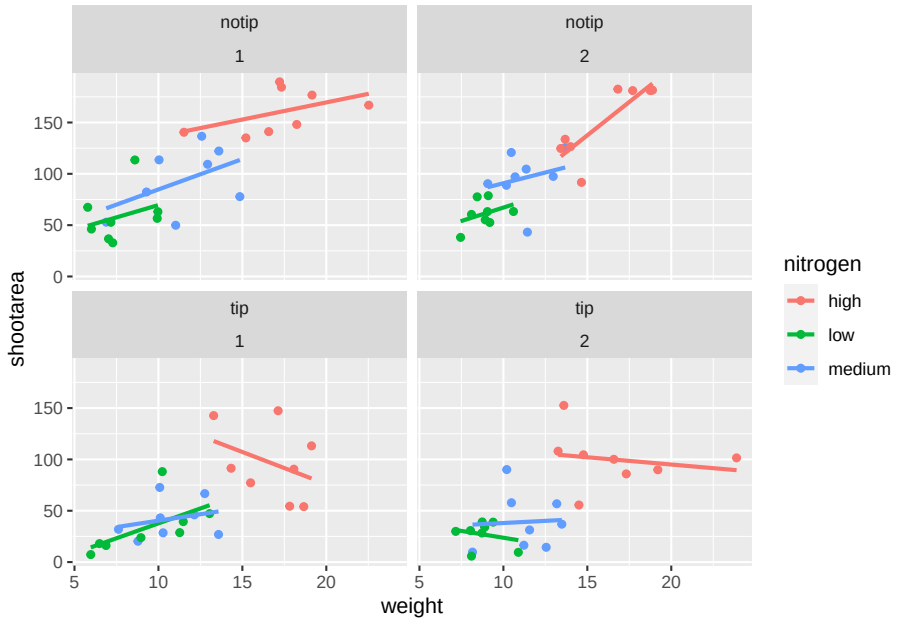

```
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
    # Splitting the single figure into multiple depending on treatment
  facet_wrap(~ treat)
```



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)
```

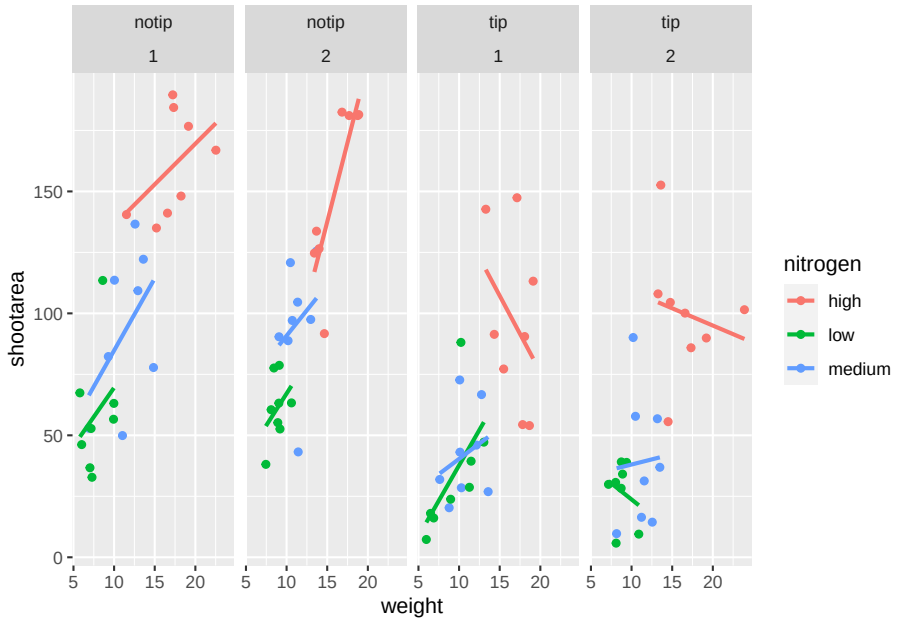


This facet layout is almost exactly what we want. Close but no cigar. In this case we actually want to be using `facet_grid()`, an alternative to `facet_wrap()`, which should put us back on track to make the "final figure".

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).

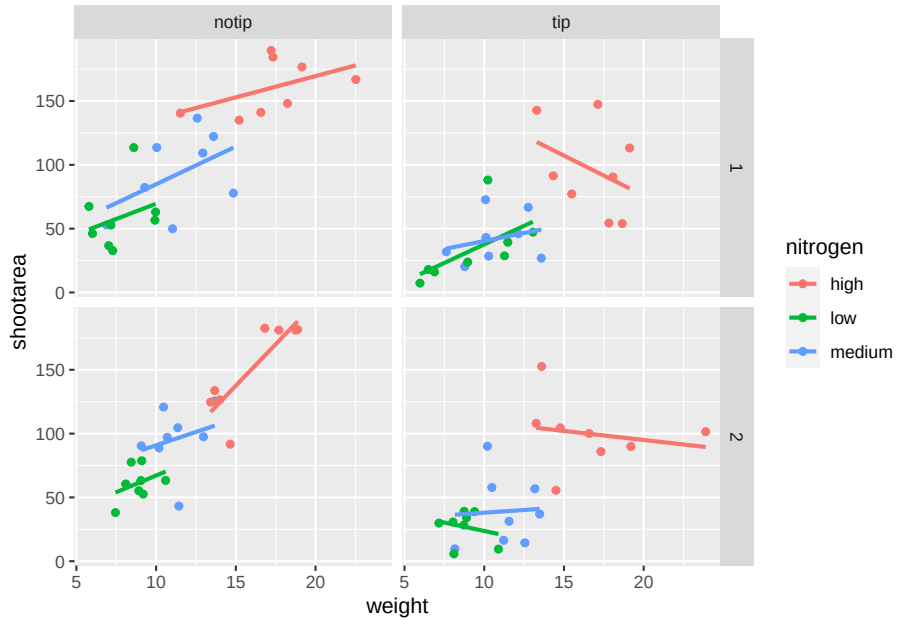
Let's try using `facet_grid` instead of `facet_wrap` to produce the following plot.


```
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).


```
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.

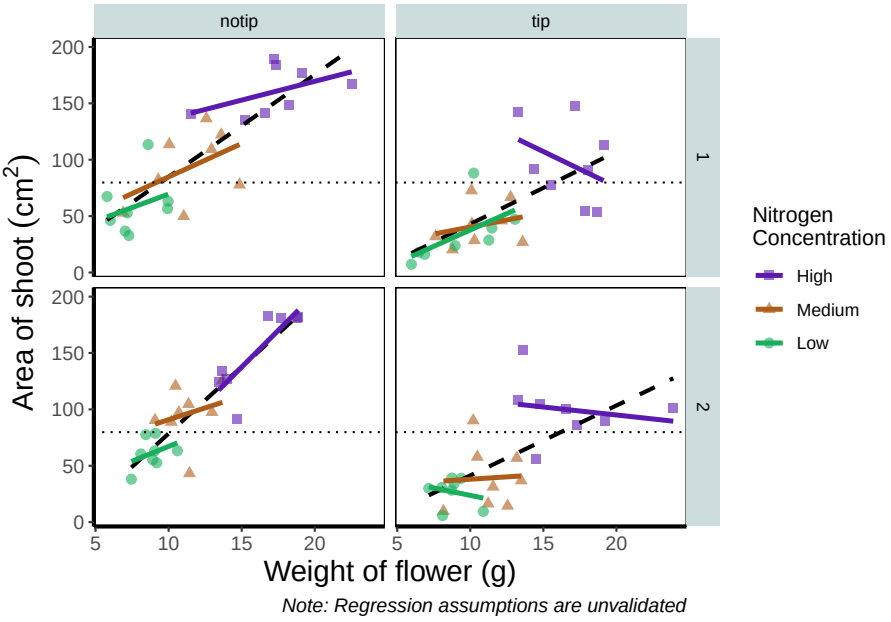
To plot both of the plots together we need to go back to our previous code and do something clever. We need to assign each figure to a separate object and then use these objects when we use `patchwork`. For instance, we can assigned our "final figure" plot to an object called `final_figure` (we're not 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:

```
p <- ggplot(df, mapping = aes(x = x, y = y))
```

And later to add additional layers:

```
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.



We'll now assign the code we wrote when creating our previous plot to an object called `rbook_figure`:

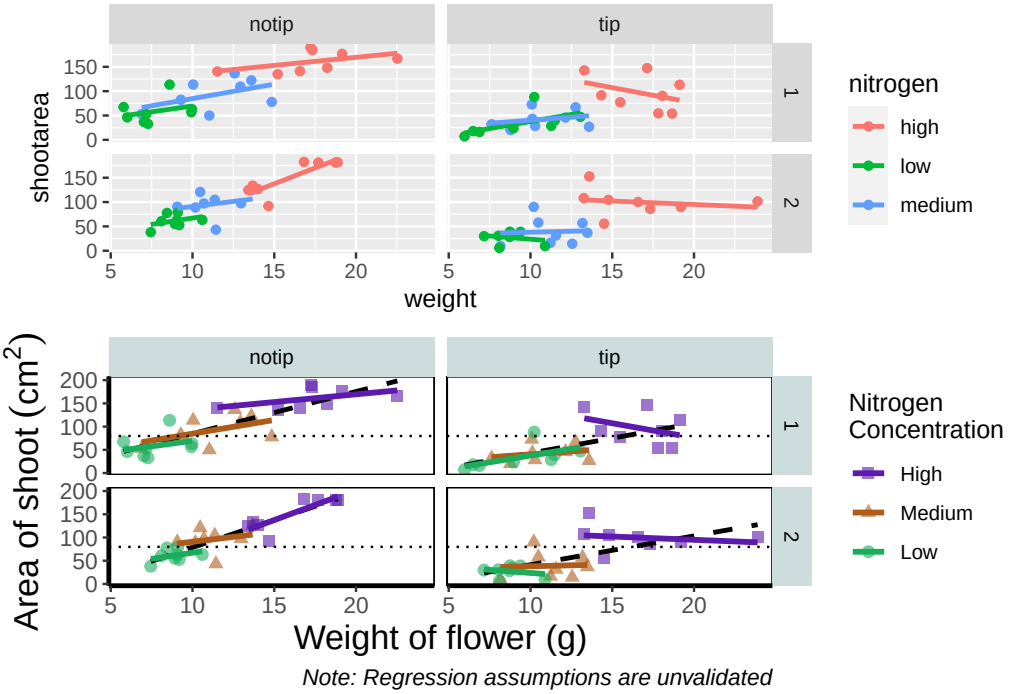
```
# Naming our figure object
rbook_figure <- ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flowe
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  facet_grid(block ~ treat)
```

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
```

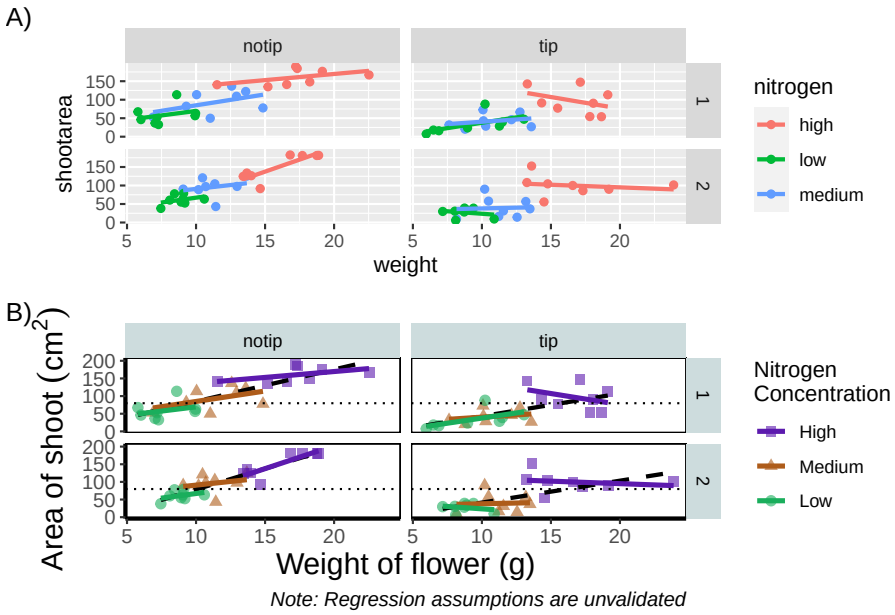


Play around: Try to create a side-by-side version of the above figure (hint: try the other operators).

We can take this one step further and assign nested `patchwork` figures to an object and use this in turn to create labels for individuals figures.

```
nested_compare <- rbook_figure / final_figure

nested_compare +
  plot_annotation(tag_levels = "A", tag_suffix = ")")
```



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

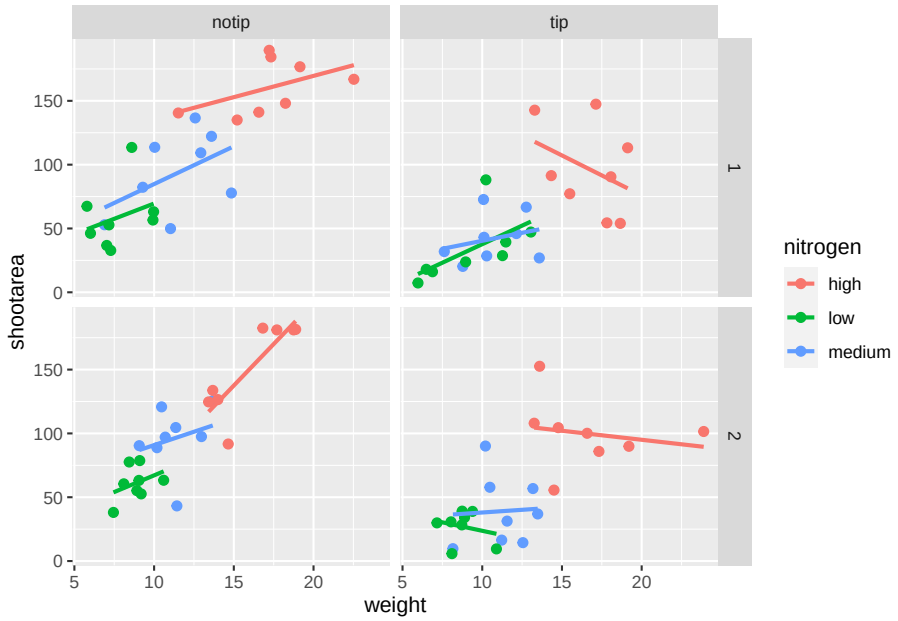
While we already have a great figure showing the main aspects of our data, it uses many default layer options. Whilst the default options are fine we may want to change them to get our plot looking exactly how we want it. Maybe we're going to use this figure in a presentation and we want to make sure 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) +
```

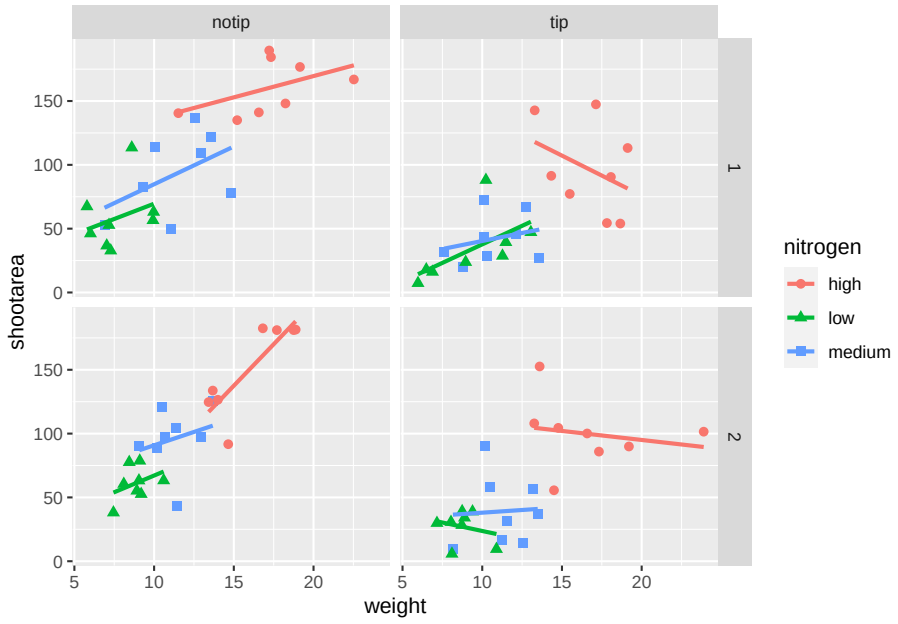
```
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat)
```



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.

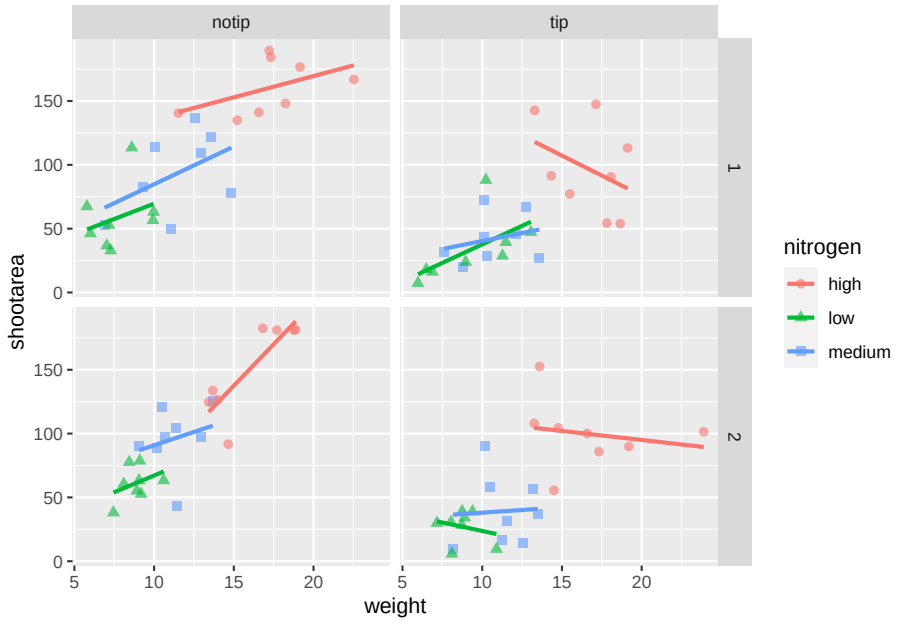

```
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    # Including shape argument to change the shape of the points
    geom_point(aes(shape = nitrogen), size = 2) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat)
```



Try including `shape = nitrogen` without also including `aes()` and see what happens.

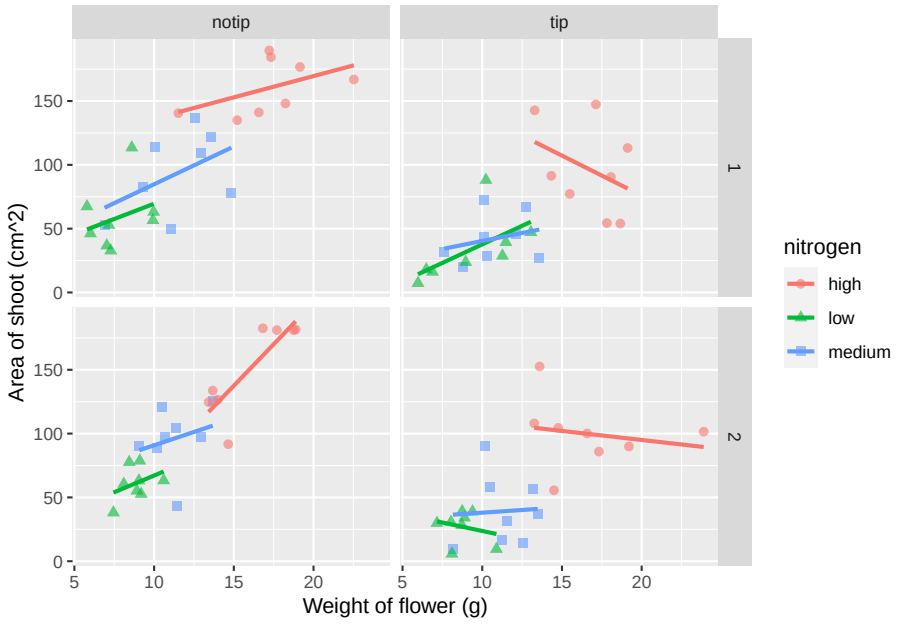
We're edging our way closer to our "final figure". Another thing we may want to be able to do is change the transparency of the points. While it's not actually that useful here, changing the transparency of points is really valuable when you have lots of data resulting in clusters of points obscuring 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)
```



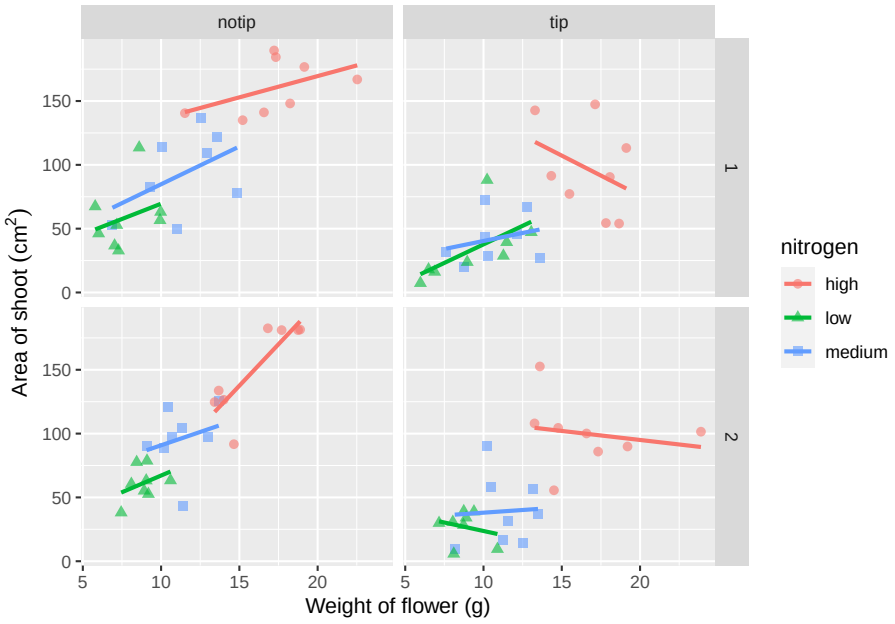
We can also include user defined labels for the x and y axis. There are a couple of ways to do this, but a more familiar way may be to use the same syntax as used in base R figures; using `xlab()` and `ylab()`. We'll specify that these belong to the `ggplot` by using the `+` symbol.

```
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)")
```



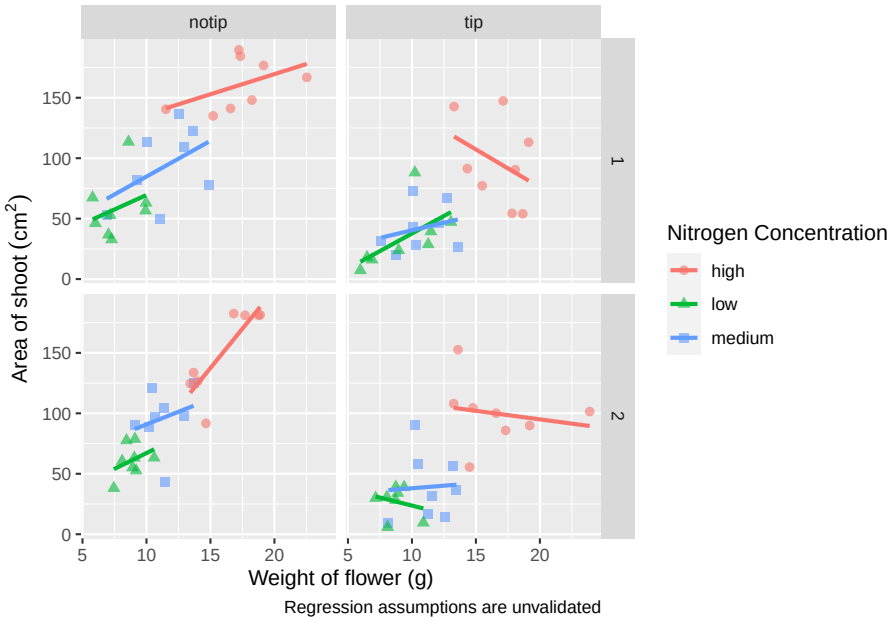
Great. Just as we wanted, though getting the "(cm^2)" to show the square as a superscript would be ideal. Here, we're going to accomplish that using a function which is part of base R called `bquote()` which allows for special characters to be shown.

```
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 straightforward 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:

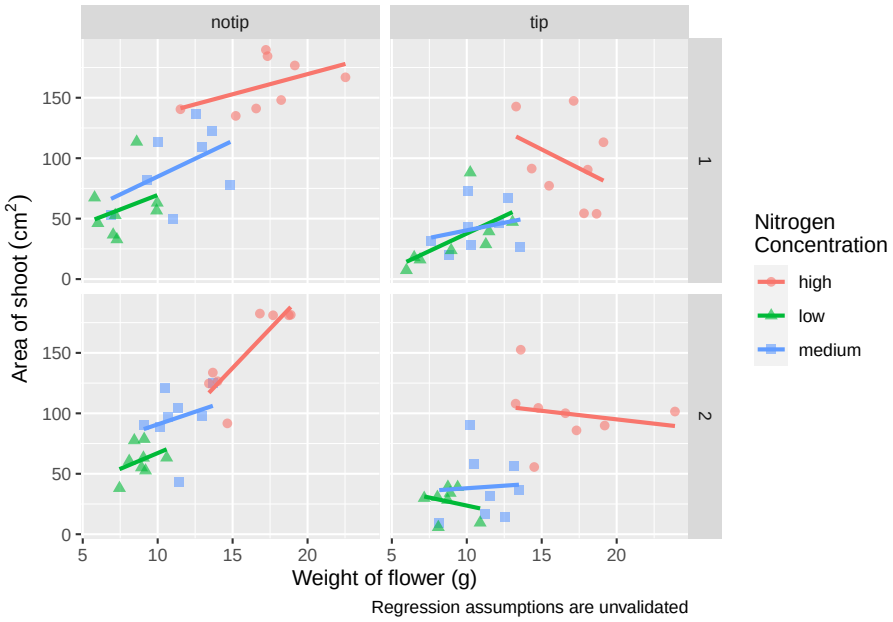
```
ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
    facet_grid(block ~ treat) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    # 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.

```
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


```
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()

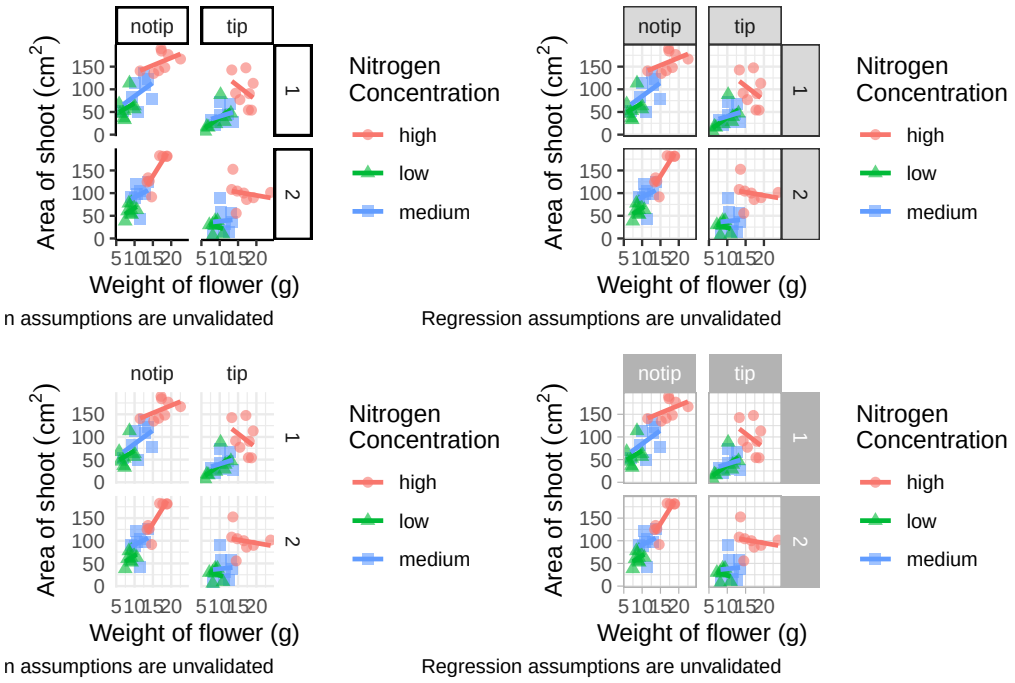
bw <- ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    geom_smooth(method = "lm", se = FALSE) +
```

```
    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 = "bla
                size = rel(1)), legend.key = element_blank(),
            strip.background = element_rect(fill = "white", colour = "black",
                size = rel(2)), complete = TRUE)
}
<bytecode: 0x7fb9babbd5f0>
<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:

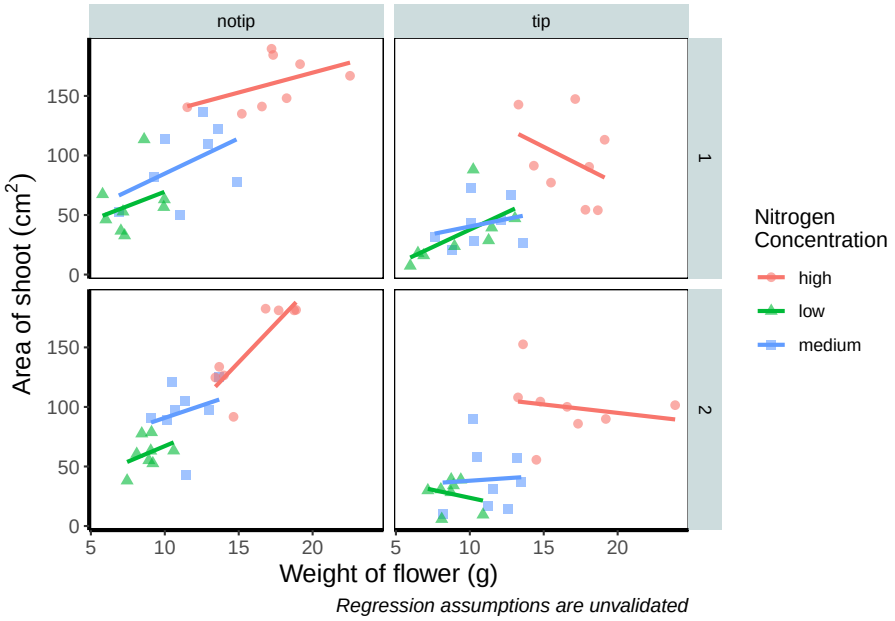
- Font size for axis titles = 13
- Font size for x axis text = 10
- Font size for y axis text = 10
- Font for caption = 10 and italics
- Background colour = white
- Background border = black
- Axis lines = black
- Strip colour (for facets) = light blue
- Strip text colour (for facets) = black
- Legend 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.


```
theme_rbook <- function(base_size = 13, base_family = "", base_line_size = base_size/2
                        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.

```
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
```

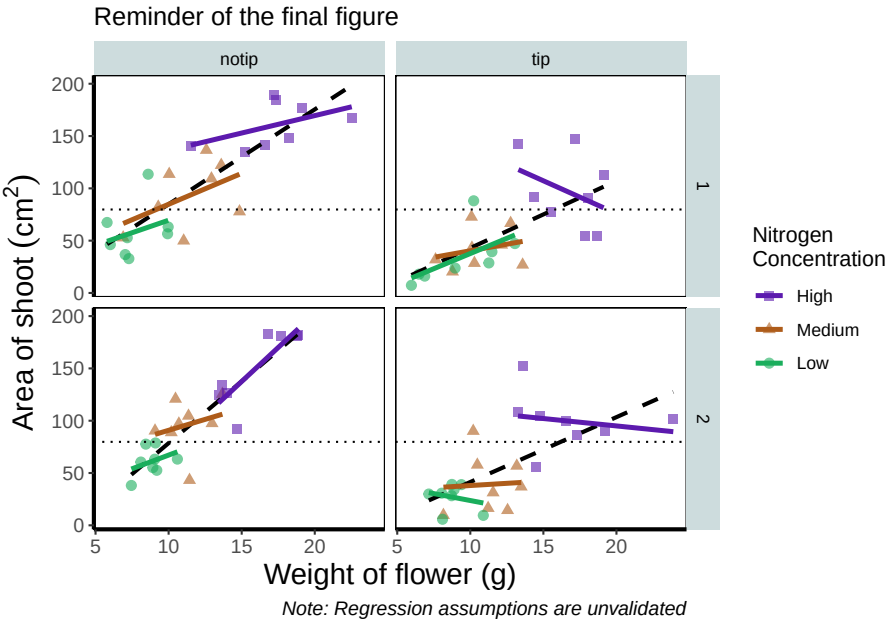


5.2.6 Prettification

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.

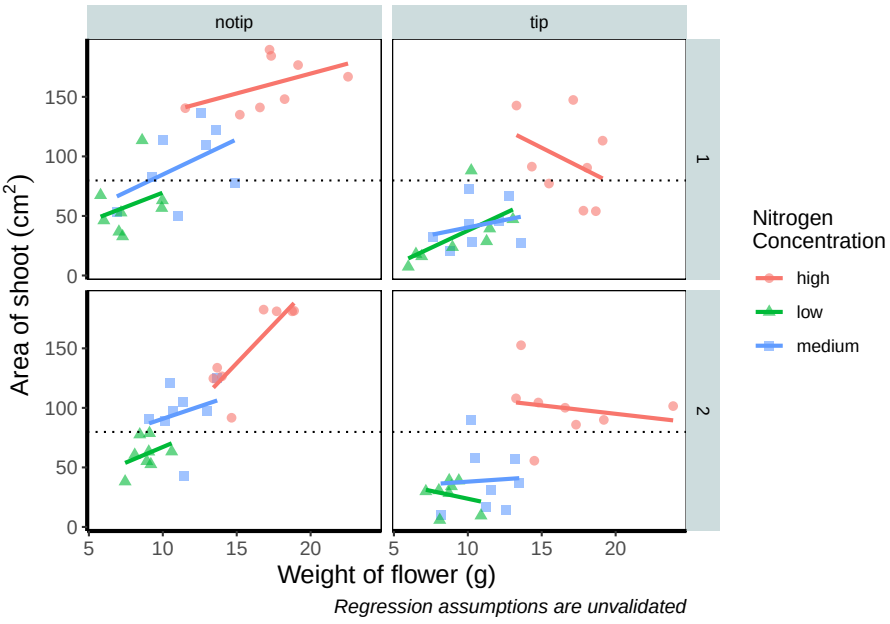
```
final_figure +
    labs(title = "Reminder of the final figure")
```



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", linety

    theme_rbook()
```

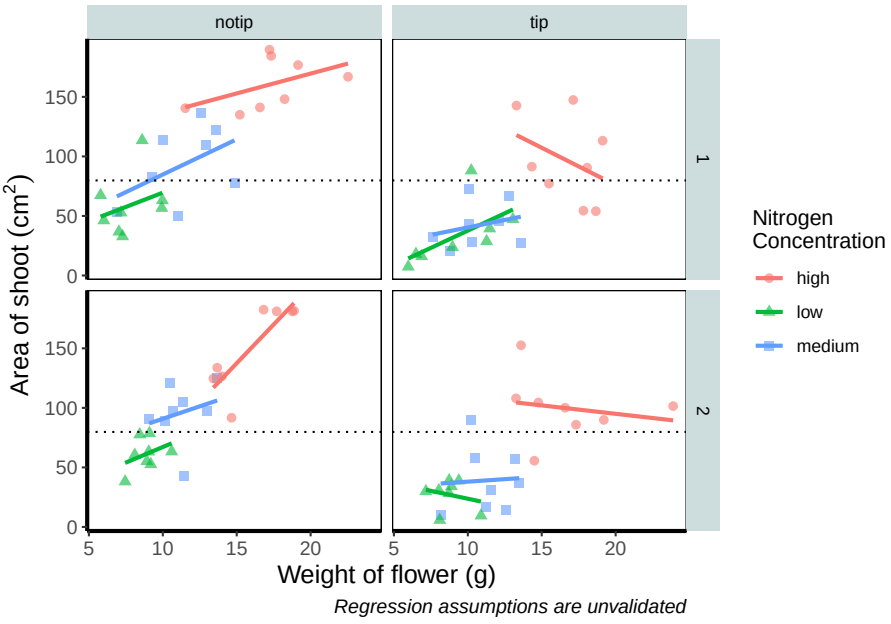


Notice how we included the function `mean(shootarea)` within the `geom_hline()` function? We could also do that externally to the `ggplot2` code and get the same result.

```
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) +

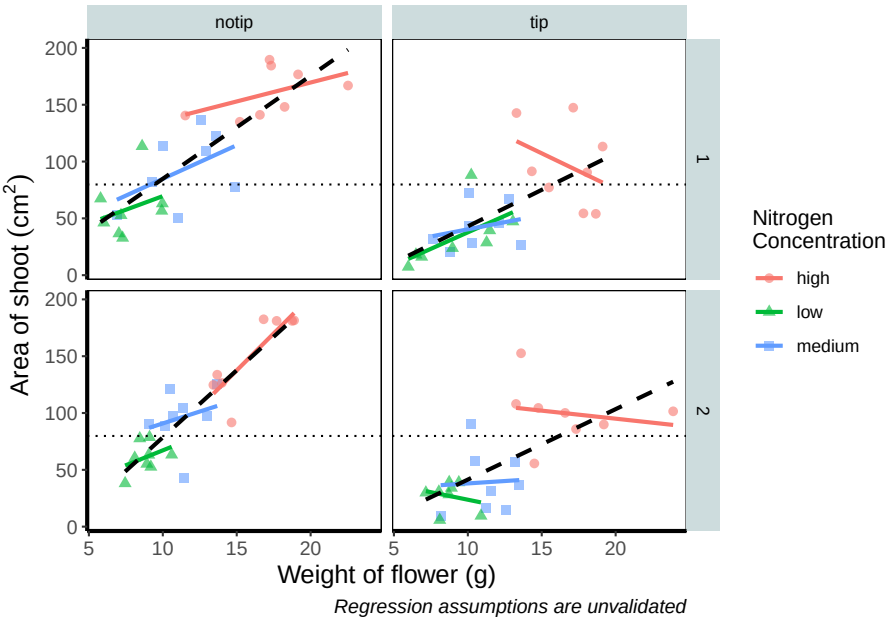
    theme_rbook()
```



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…


```
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()
```



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.

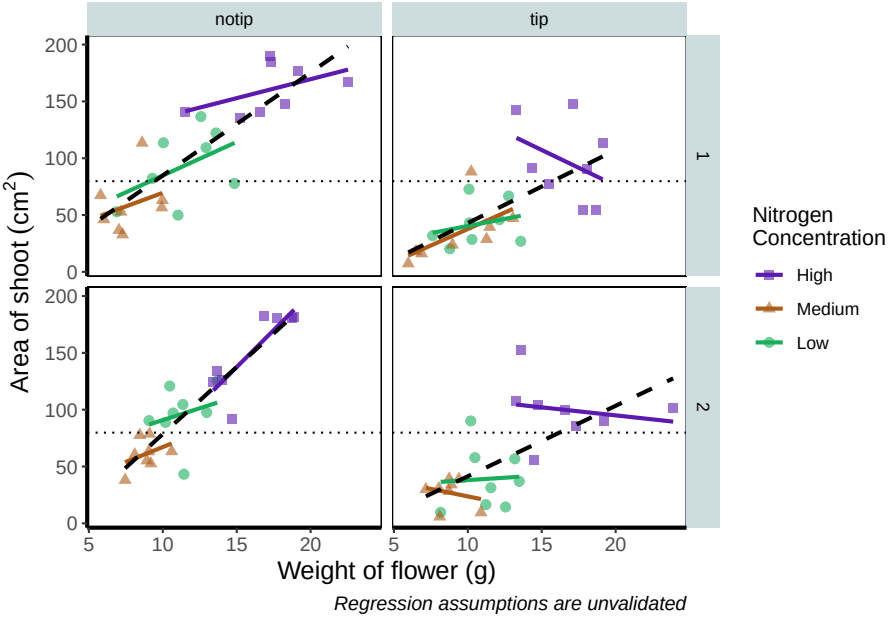
Instead of doing this within the specific geom, we'll use `scale_colour_manual()` as well as `scale_shape_manual()`. Doing it this way will allow us to do two things at once; change the shape and colour to our choosing, and assign labels to these (much like what we did with `xlab()` and `ylab()`). Doing 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:


```
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.

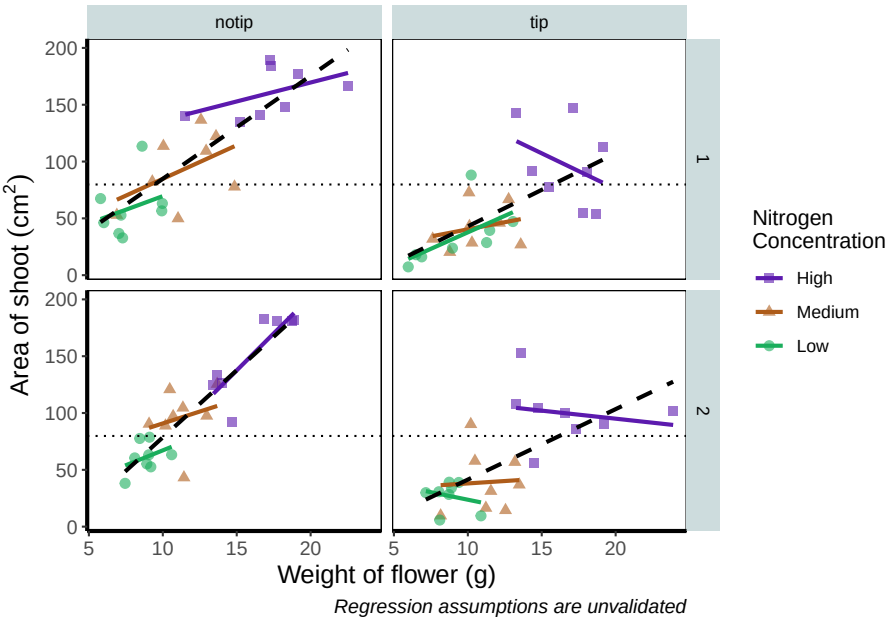
So how do we fix this? We need to do a little data manipulation to rearrange our factors so that the order goes High, Medium, Low. Let's do that:

```
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 = flowe
    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
```



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

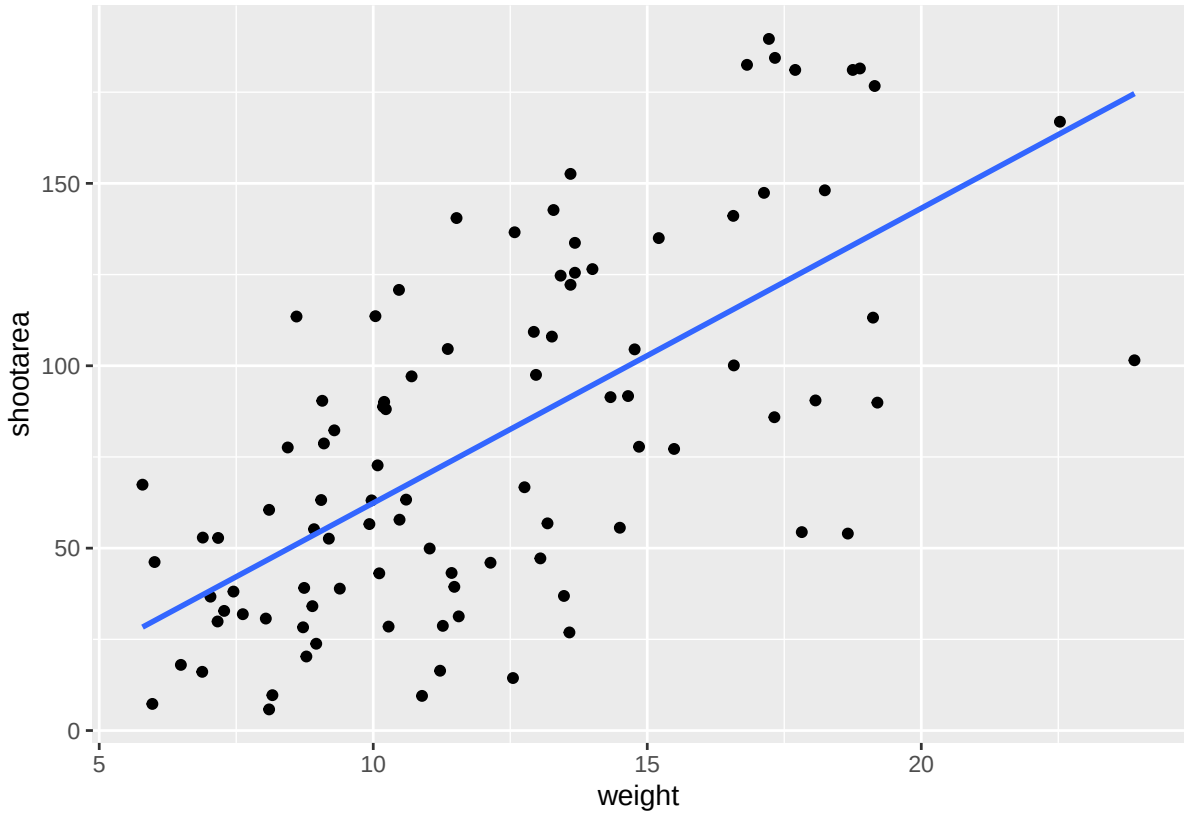
For this section we'll use various versions of the final figure without including the `facet_grid()`. We do so only to allow the changes to be more apparent. With this plot, we'll run through some tips and tricks that we wish we'd learnt when we started using `ggplot2`.

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'
```

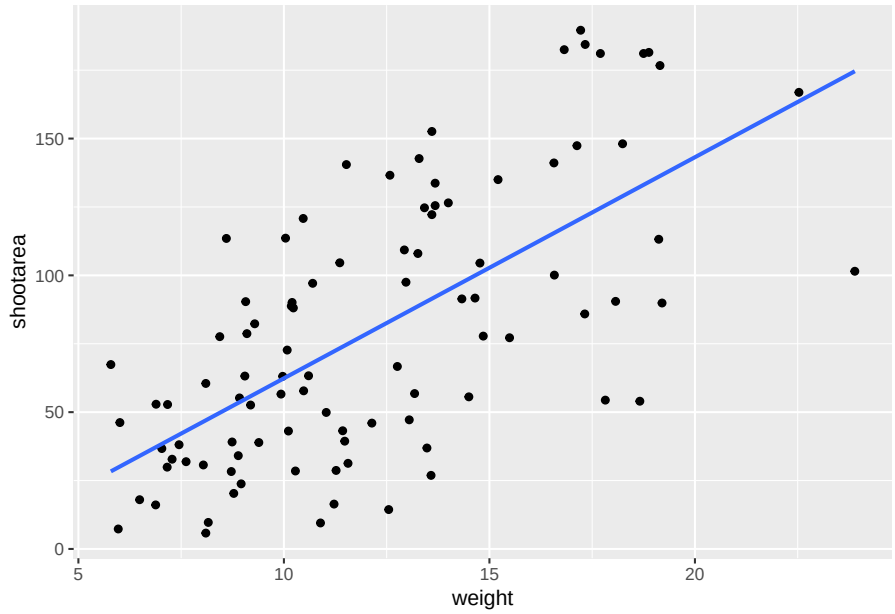


Nowhere in our dataset are there columns for either the y-intercept or the gradient needed to draw the straight line, yet we've managed to draw one. The statistics layer calculates these based on our data, without us necessarily knowing what we've done. It's also the engine behind converting your data 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)
```



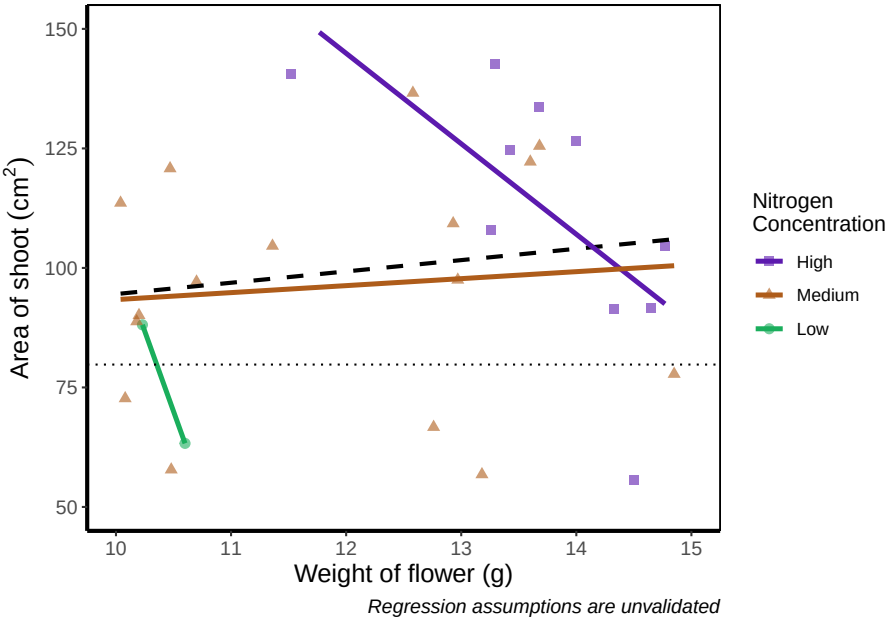
While in this example it doesn't make a difference which we use, in other cases we may want to use the calculated statistics in alternative ways. We won't get into it, but see `?after_stat` if you are interested.

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))
```

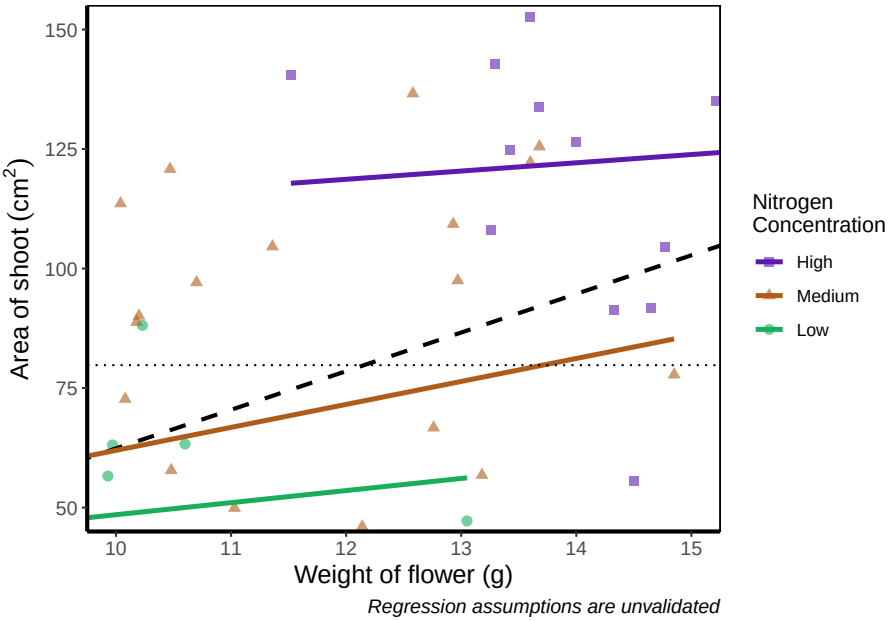


If you run this yourself you'll see warning messages telling us that n rows contain either missing or non-finite values. That's because we've essentially chopped out a huge part of our data using this method (everything outside of the ranges that we specified is "removed" from the data). As a result 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))
```



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

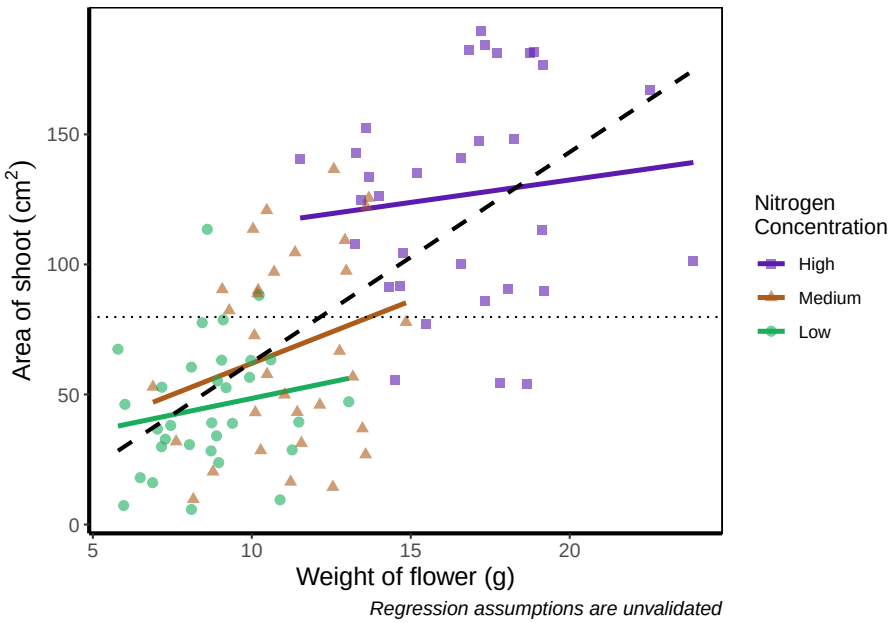
Remember that `ggplot2` works like painting. Let's imagine we were one of the great renaissance painters. We've just finished the focal point of our painting, the half naked Duke of Toulouse looking moody or something. Now that we've finished painting the honourable Duke, we proceed to paint the background, 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:

```
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()
```



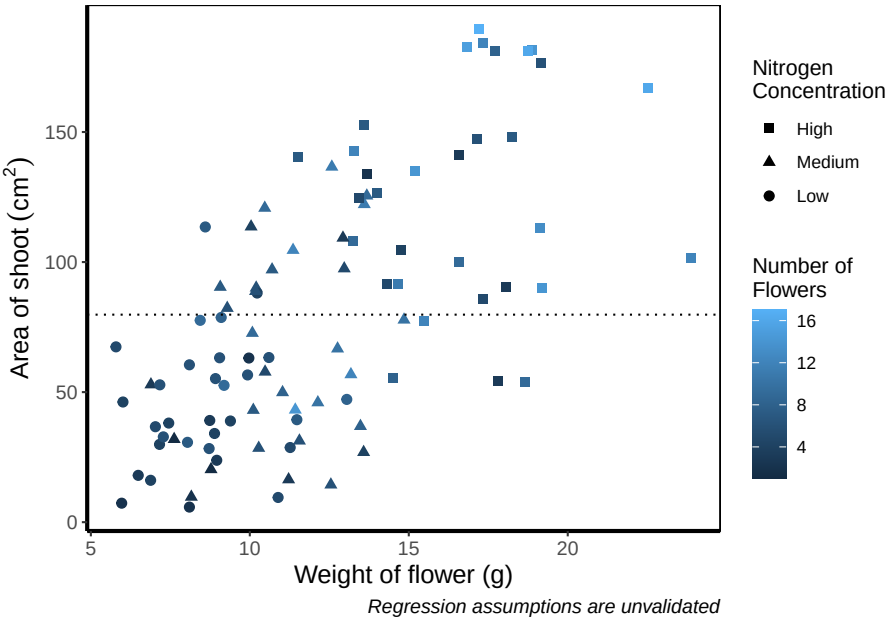
It's not the clearest example, but give it a shot with your own data.

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.

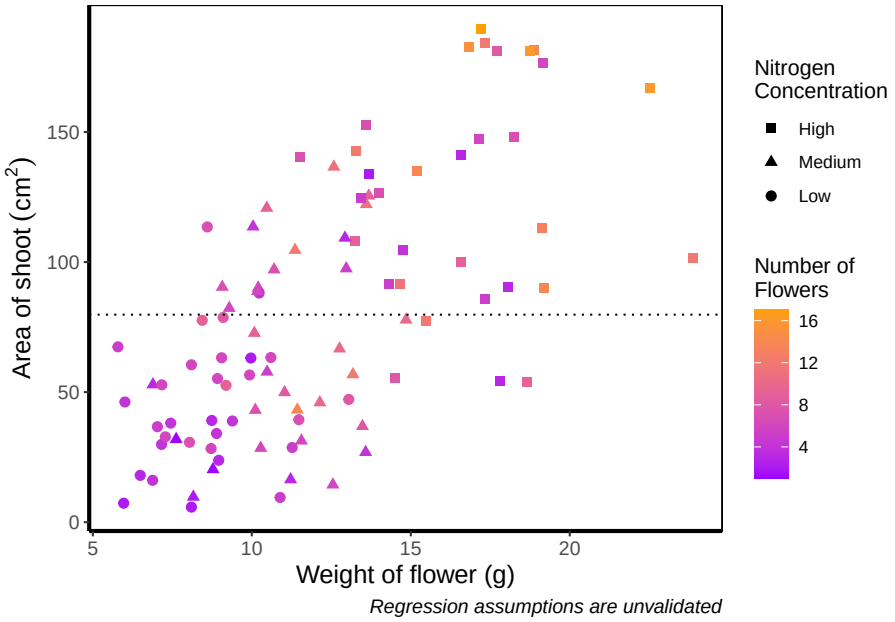
We'll start with the default option. Here we only need to change nitrogen (which is a factor) to flowers (which is continuous) in the `colour =` argument within `aes()`:


```
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()
```



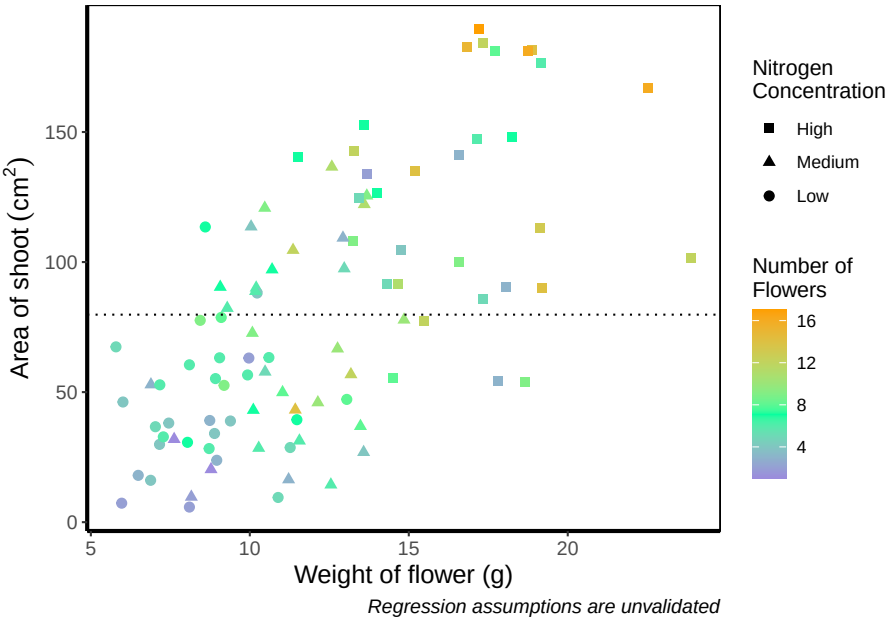
And as easily as that we have a colour gradient to show number of flowers. Dark blue shows low flower numbers and light blue shows higher flower numbers. While the code has worked, we have a hard time distinguishing between the different shades of blue. It would help ourselves (and our audience), if 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
    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()
```



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()`.

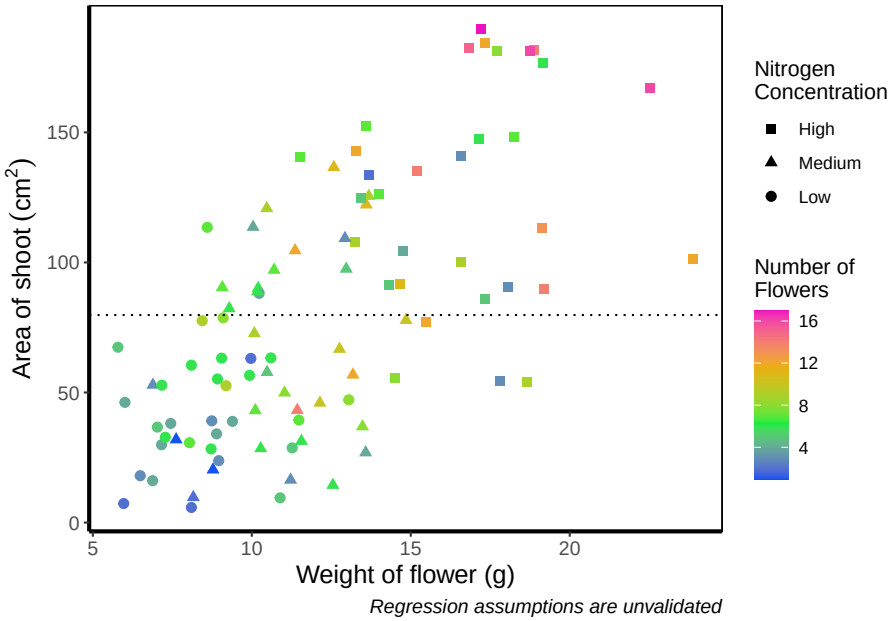

```
ggplot(aes(x = weight, y = shootarea), data = flower) +
    geom_point(aes(colour = flowers, shape = nitrogen), size = 2) +
    xlab("Weight of flower (g)") +
    ylab(bquote("Area of shoot"~(cm^2))) +
    geom_hline(aes(yintercept = 79.7833), size = 0.5, colour = "black", linetype = 3)
    labs(shape = "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()
```



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()
```



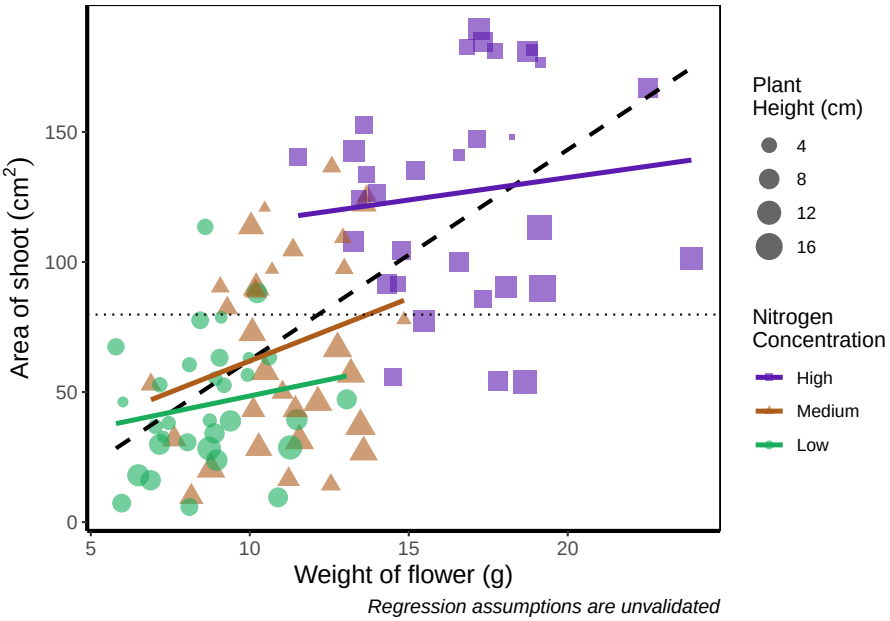
Slightly nauseating but it's doing what we wanted it to do, so we shouldn't really complain.

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.


```
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()
```



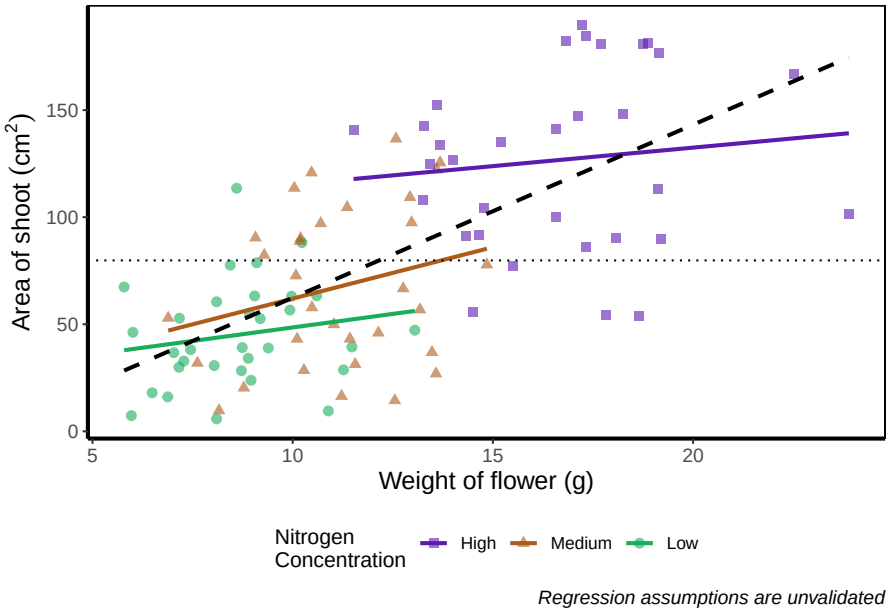
Now the sizes reflect the height of the plants, with bigger points representing taller plants and vice-versa.

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()`.


```
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()
```



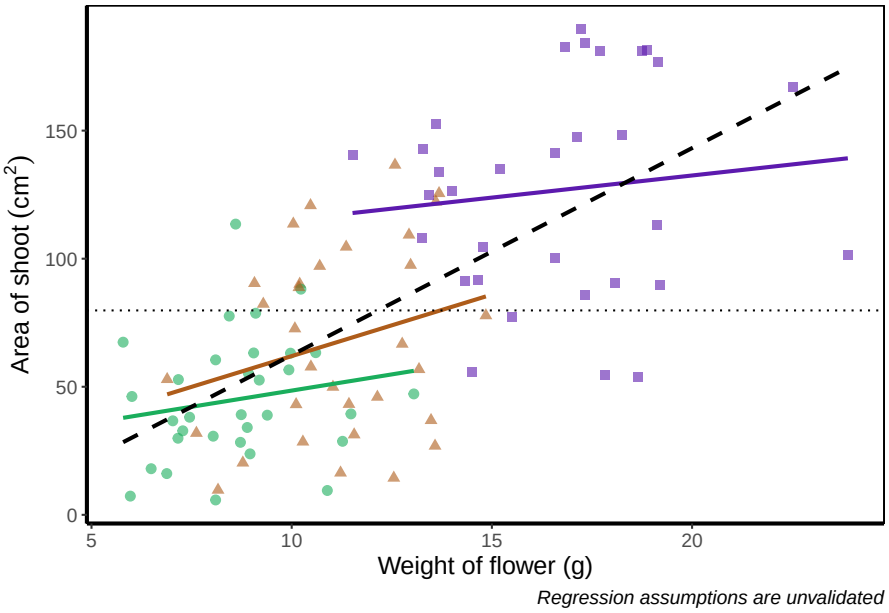
Play around: Try the code above but don't include `theme_rbook()` and see what happens? What about if you try `theme_rbook(legend.position = "bottom")`?

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()
```

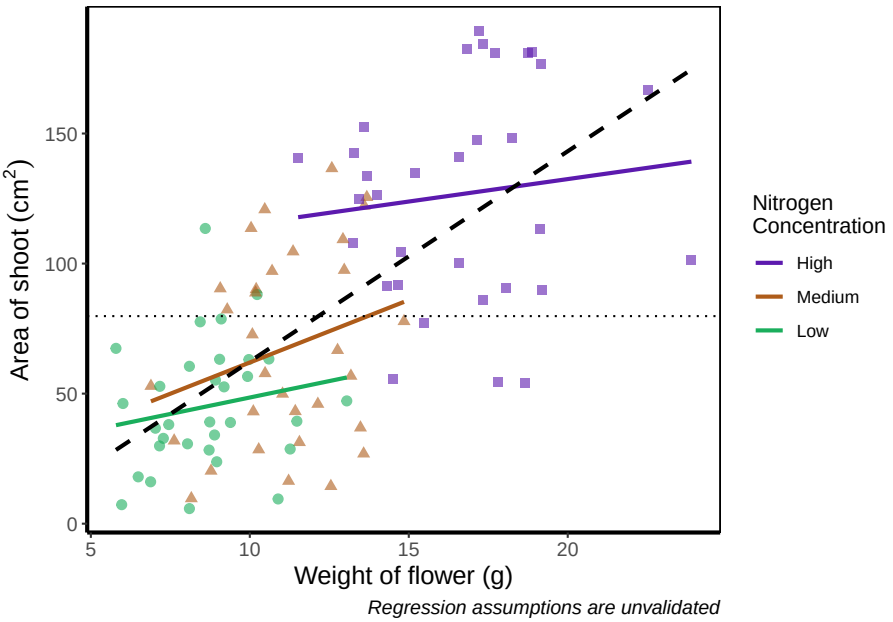


5.3.8 Hiding part of the legend

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`.


```
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()
```



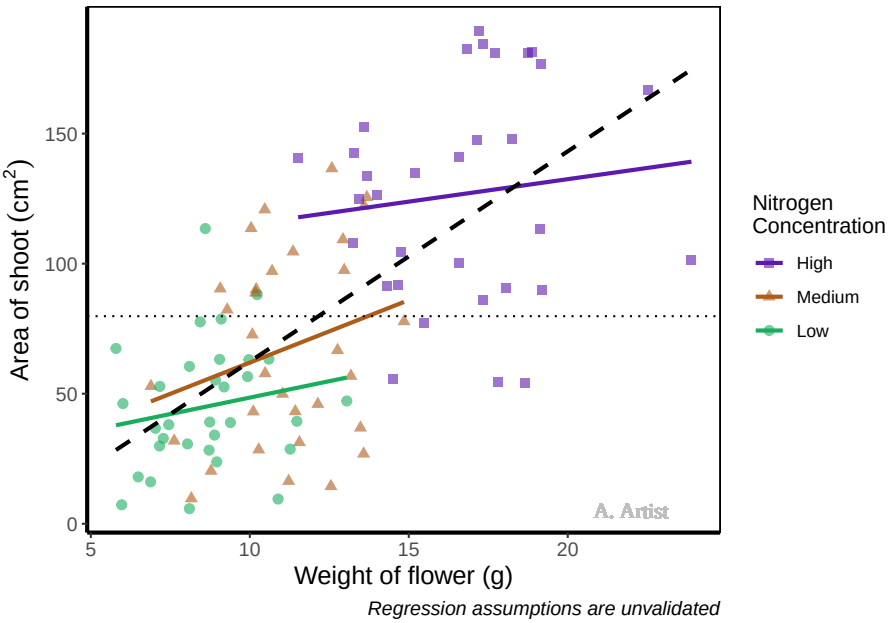
5.3.9 Writing on a figure

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()
```

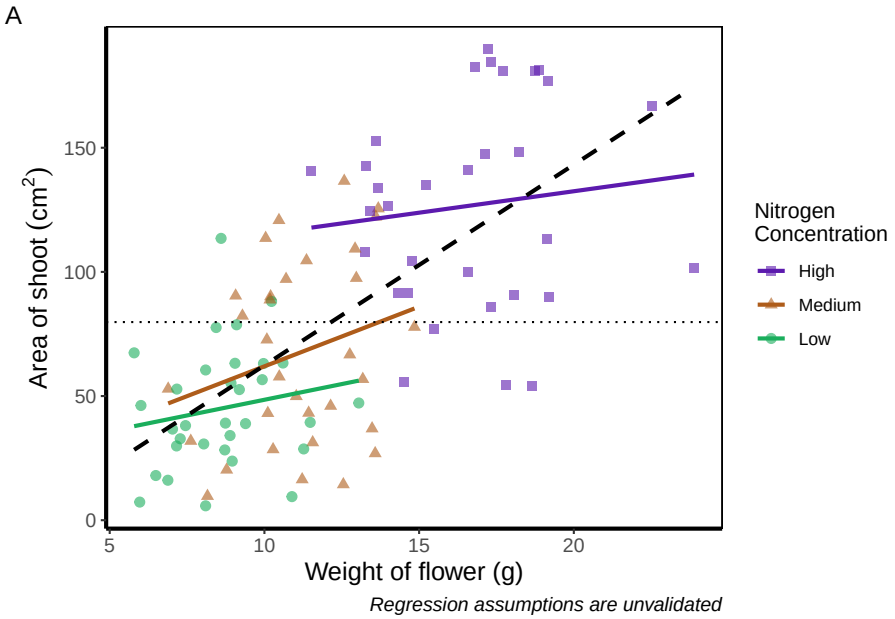


In reality there are more appropriate uses for `geom_text()` but whatever the reason, the mechanics remain the same. We specify what the x and y position are (on the scale of the axes), what we want written (using `label =`), the font (using `family =`). If you want to do more complex tasks with text 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:

```
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()
```

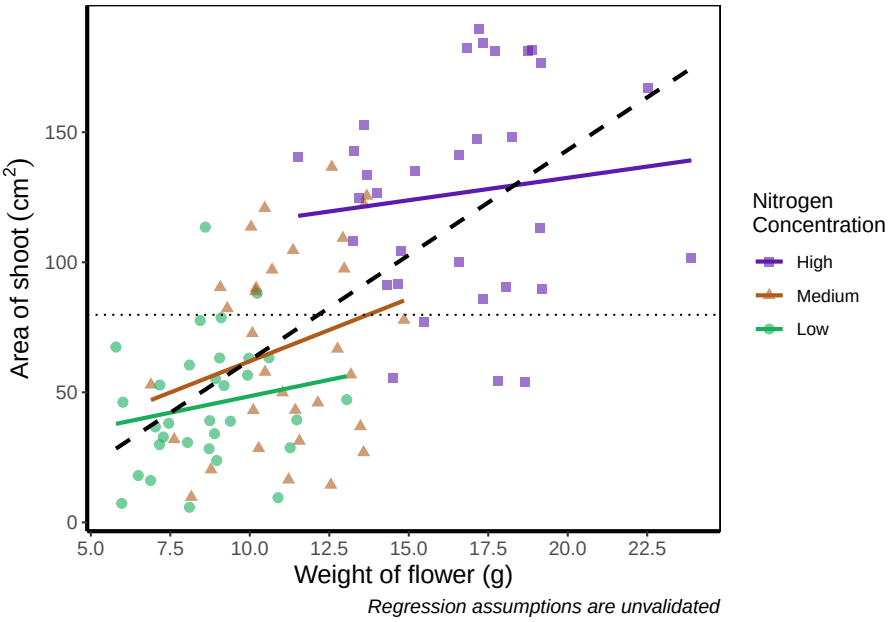


Doing so we get an "A" at the top left of the figure. If we weren't happy with the position of the "A", we can always use `geom_text()` instead and position it ourselves.

5.3.10 Axes tick marks and tick labels

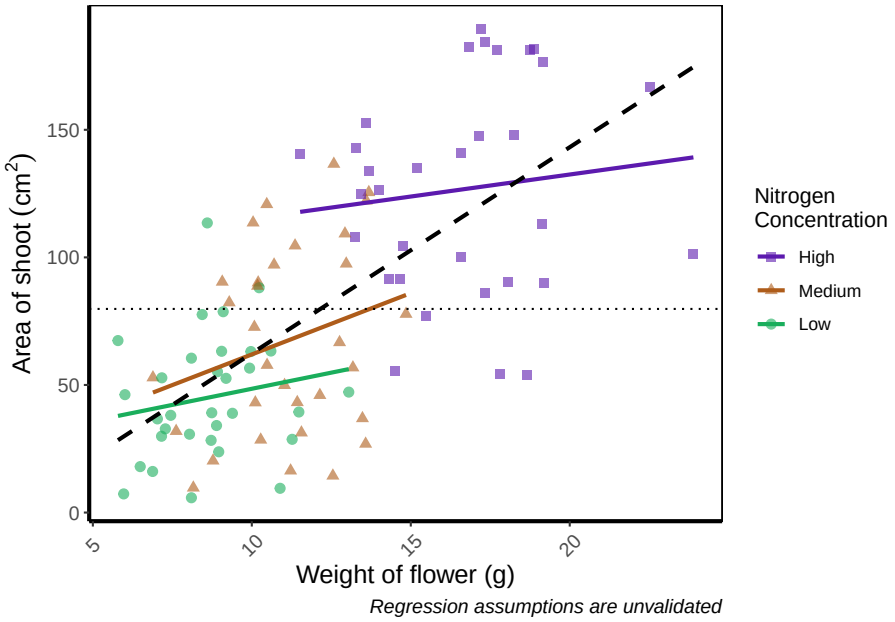
What are we to do if we want more or fewer ticks on an axis? We can do this using the appropriate layers; `scale_y_discrete()` and `scale_x_discrete()` for discrete data (e.g. factors); and `scale_y_continuous()` and `scale_x_continuous()` for continuous data. Within these layers, the argument we want 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()
```



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"
    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()
```



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()

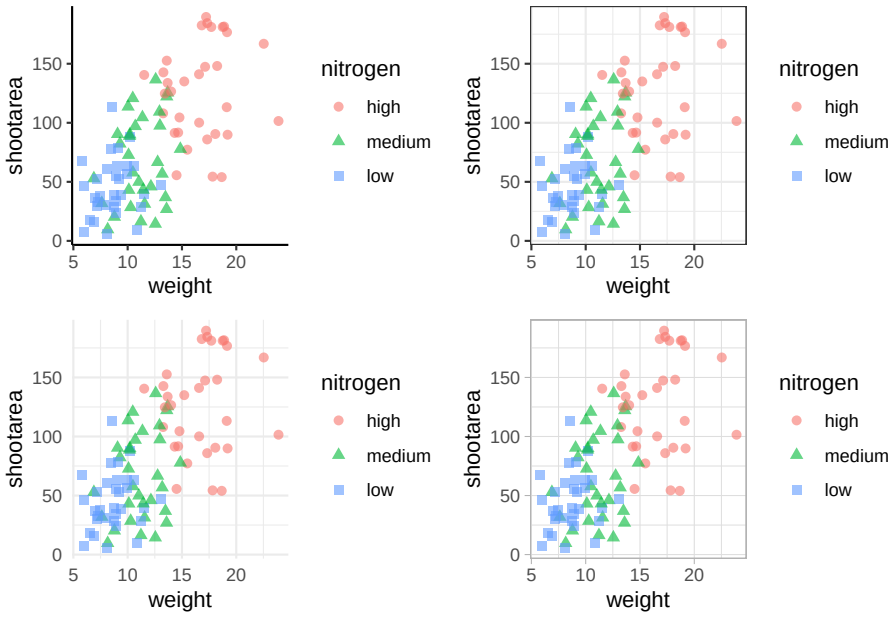
b <- ggplot(aes(x = weight, y = shootarea, colour = nitrogen), data = flower) +
    geom_point(aes(shape = nitrogen), size = 2, alpha = 0.6) +
    theme_bw()

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()

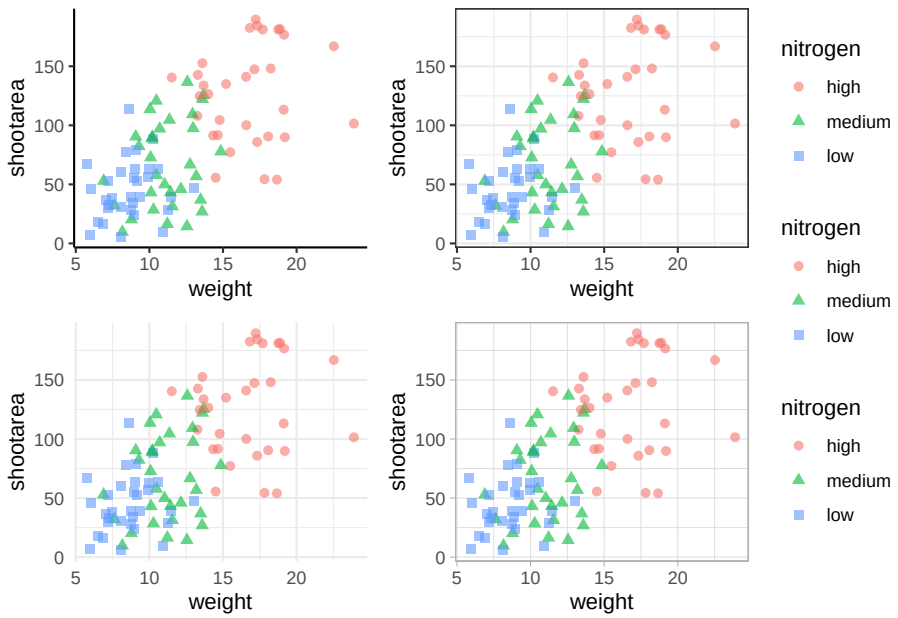
nested_1 <- a + b + c +d

nested_1
```



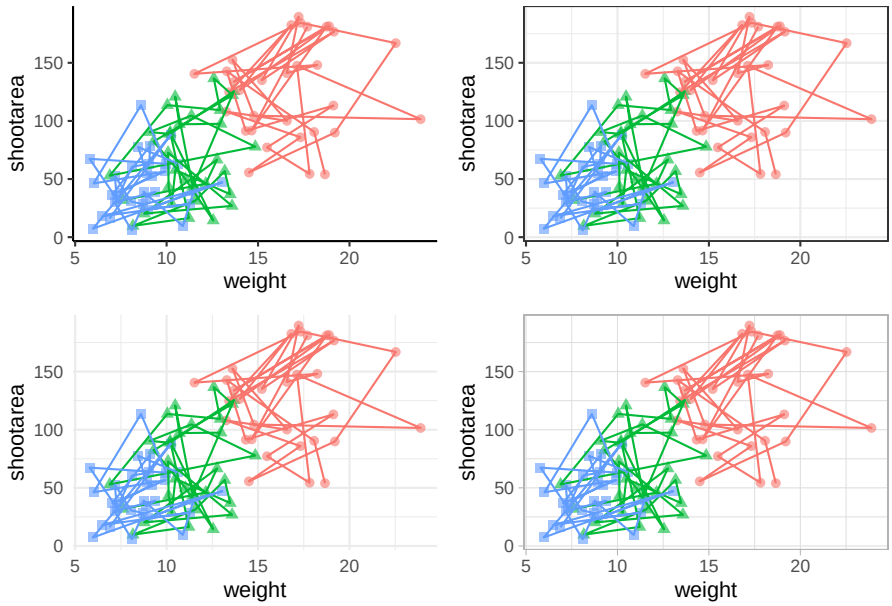
The problem with this figure is that we're using a lot of space to get legends squeezed into the middle. We can solve that by using the `plot_layout()` function and collecting the guides together:

```
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.

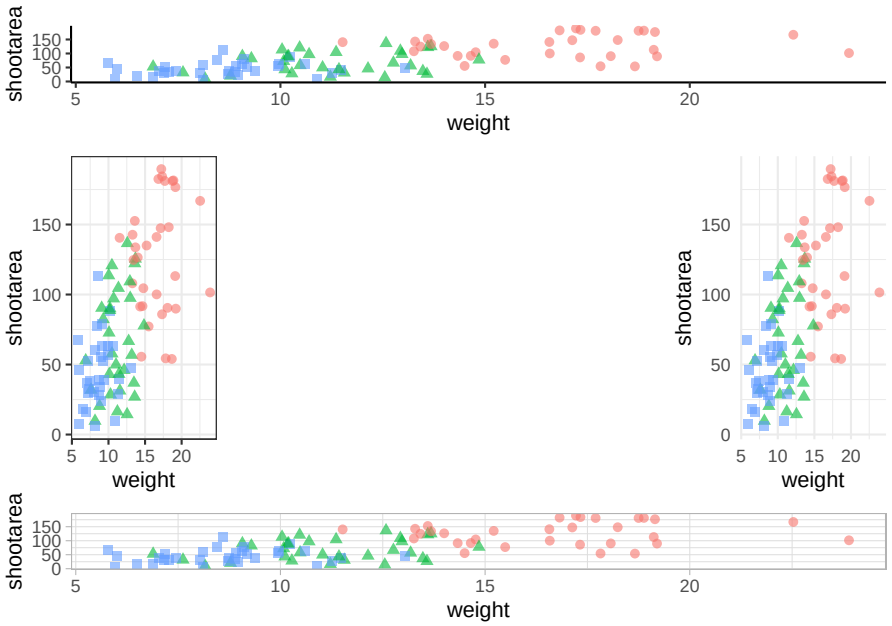
```
# note the use of '&' instead of '+'
nested_1 &
    theme(legend.position = 'none') &
    geom_path()
```



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.

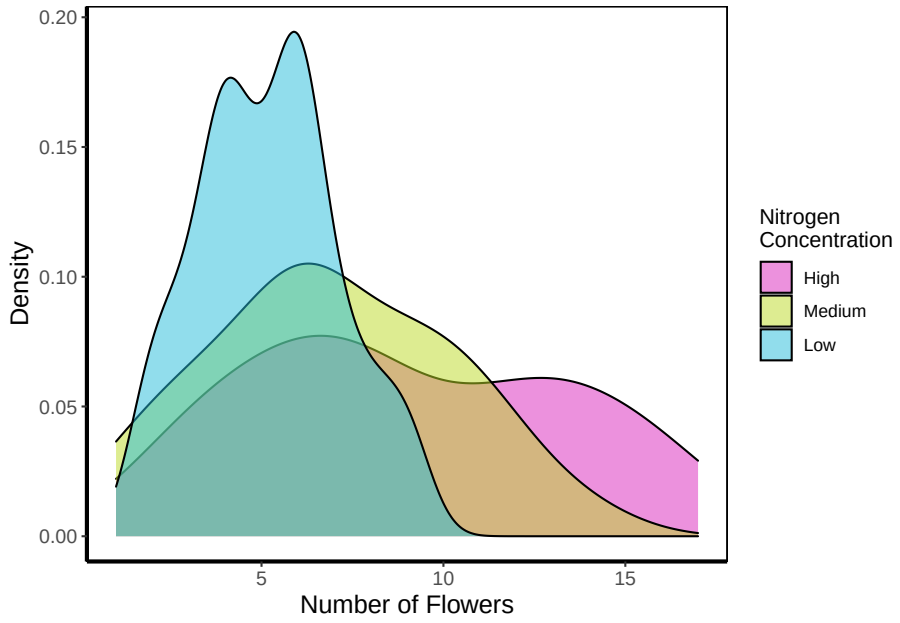
5.4.1 Density plot

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") +
```

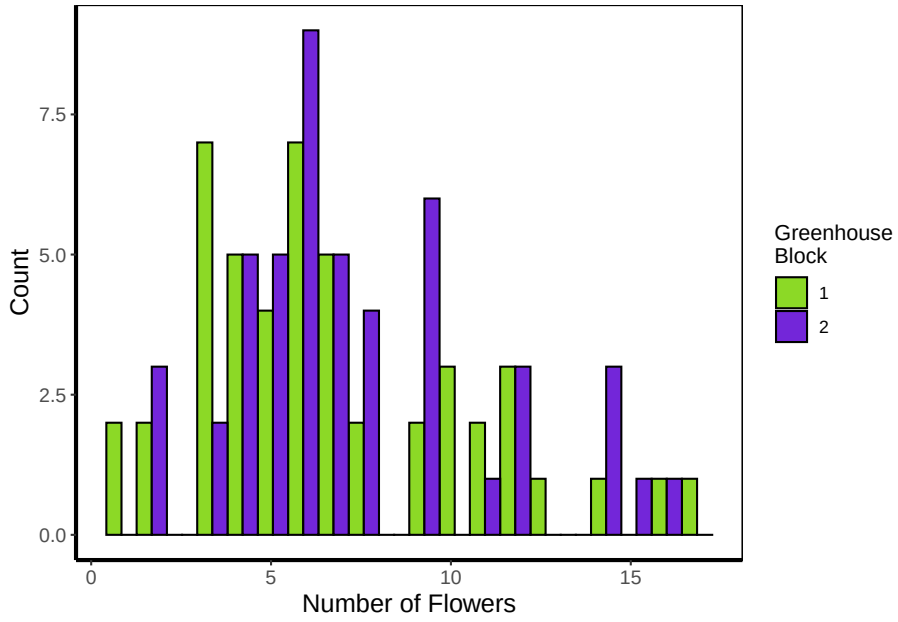
```
    scale_fill_manual(labels = c("High", "Medium", "Low"),
        values = c("#DB24BC", "#BCDB24", "#24BCDB")) +
    theme_rbook()
```



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.

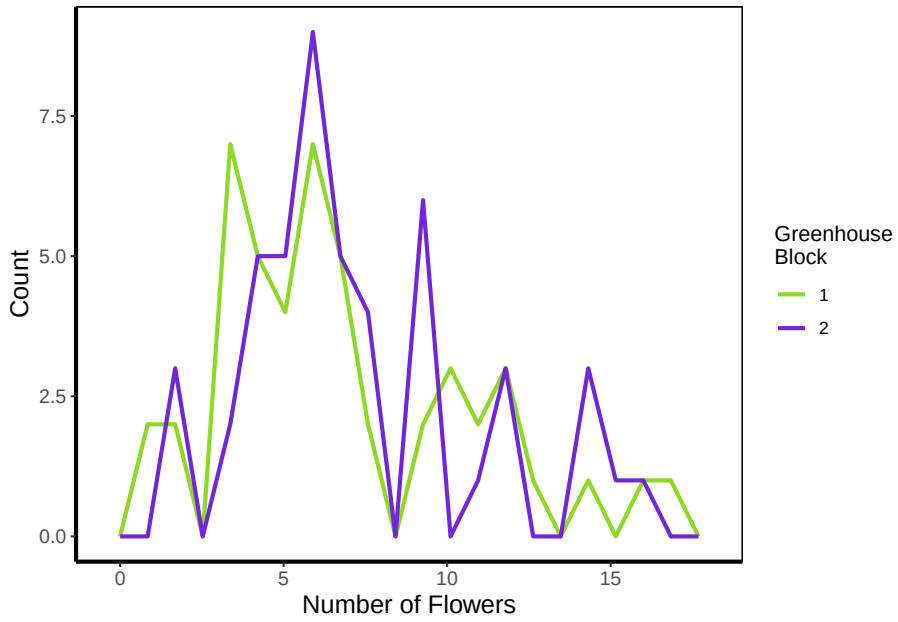
```
ggplot(flower) +
    geom_histogram(aes(x = flowers, fill = factor(block)), colour = "black", bins = 20
                   position = "dodge") +
    labs(y = "Count", x = "Number of Flowers", fill = "Greenhouse\nBlock") +
    scale_fill_manual(labels = c("1", "2"),
          values = c("#8CD926", "#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.

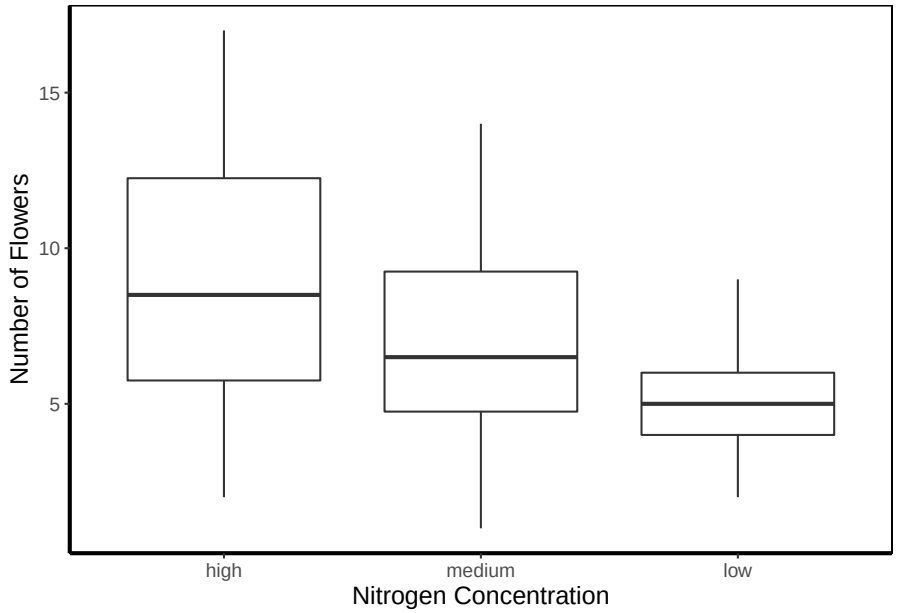
```
ggplot(flower) +
    geom_freqpoly(aes(x = flowers, colour = factor(block)), size = 1, bins = 20) +
    labs(y = "Count", x = "Number of Flowers", colour = "Greenhouse\nBlock") +
    scale_colour_manual(labels = c("1", "2"),
         values = c("#8CD926", "#7326D9")) +
    theme_rbook()
```



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()
```



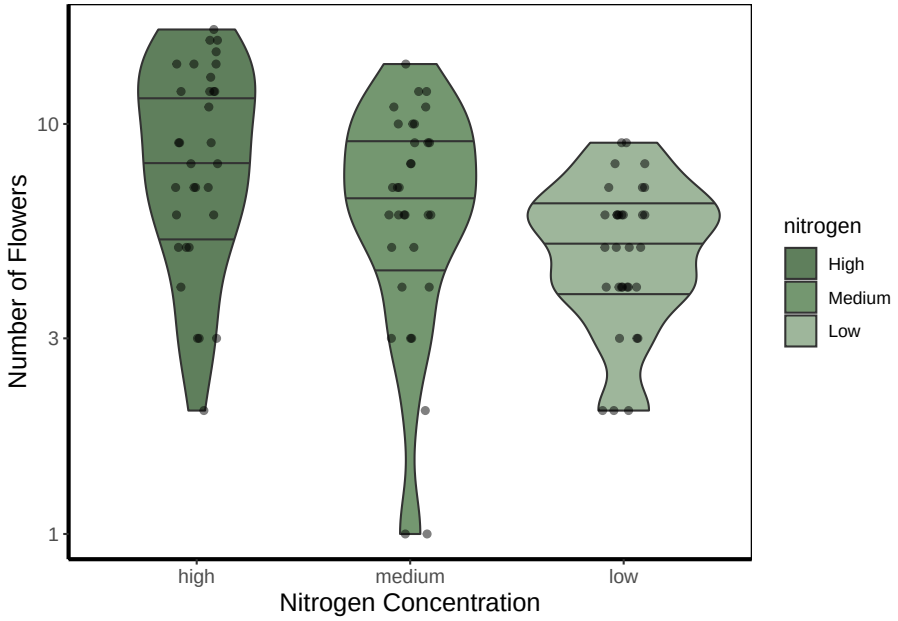
5.4.5 Violin plots

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()
```

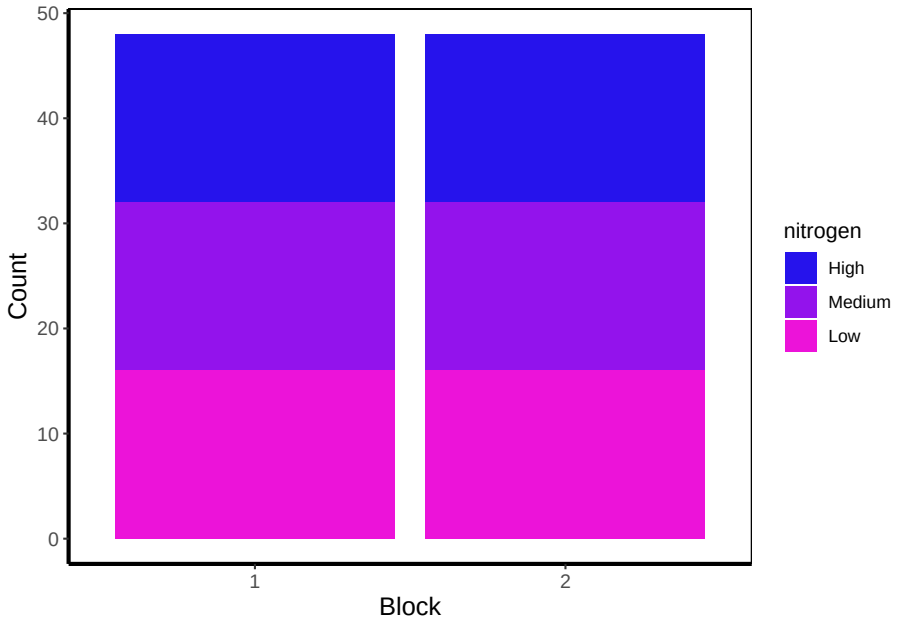


5.4.6 Barchart

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()
```



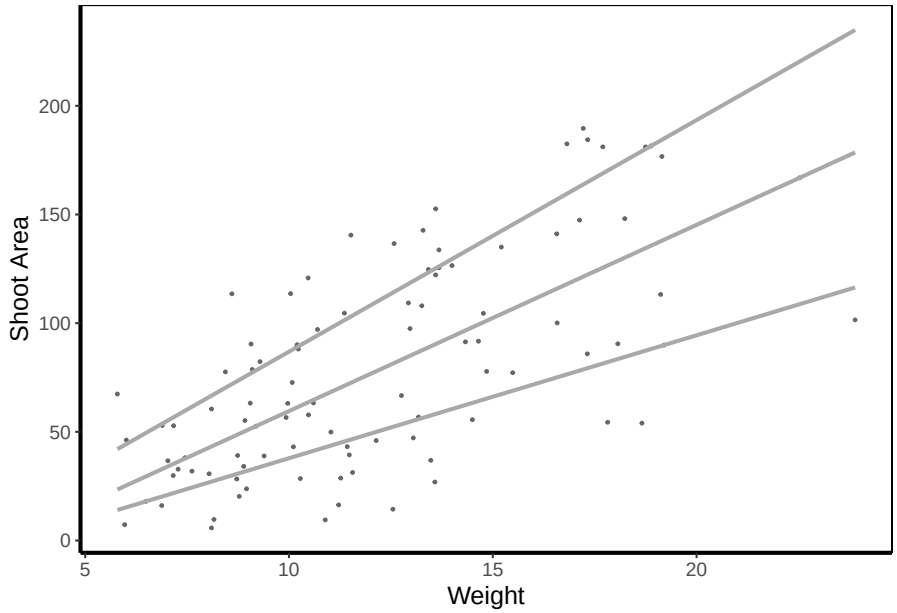
The barchart shows the numbers of observations in each block, with each bar split according to the number of observations in each nitrogen concentration. In this case they are equal because the dataset (and experimental design) was balanced.

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`.

```
library(quantreg)

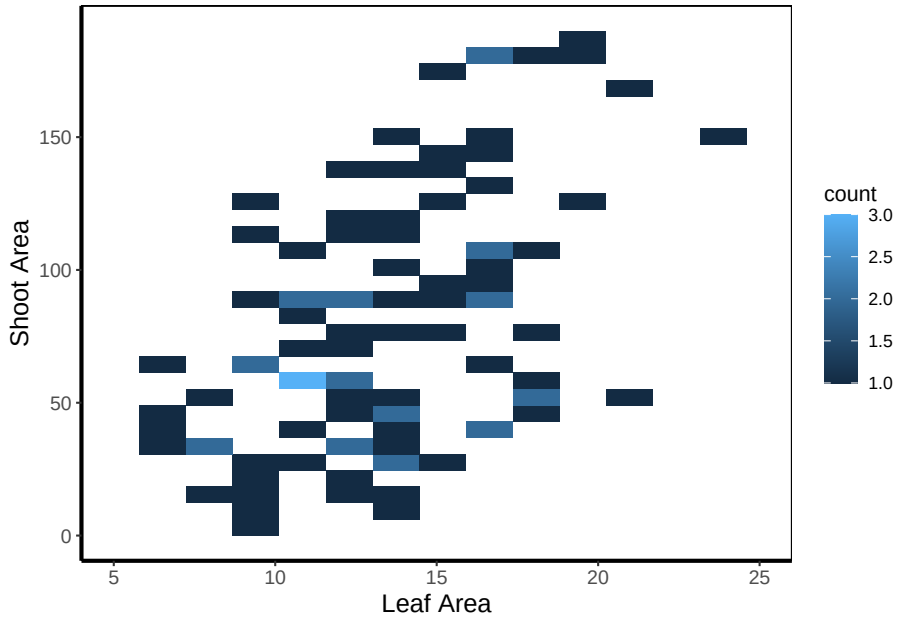
ggplot(aes(x = weight, y = shootarea), data = flower) +
    geom_point(size = 0.5, alpha = 0.6) +
    geom_quantile(colour = "darkgrey", size = 1) +
    labs(y = "Shoot Area", x = "Weight") +
    theme_rbook()
```



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()
```



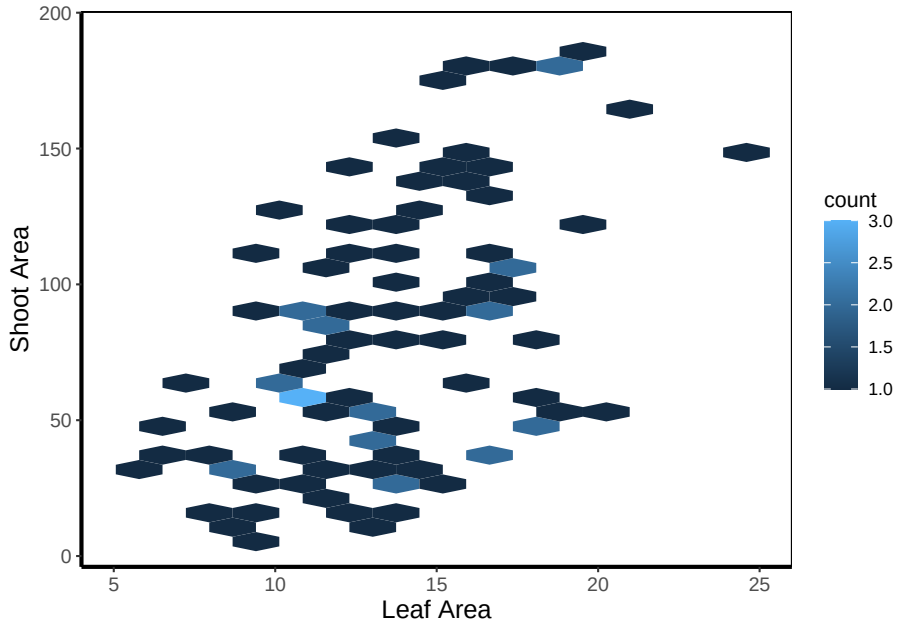
In this example, lighter blue squares show combinations of leaf area and shoot area where we have more data, and dark blue shows the converse.

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`.

```
library(hexbin)

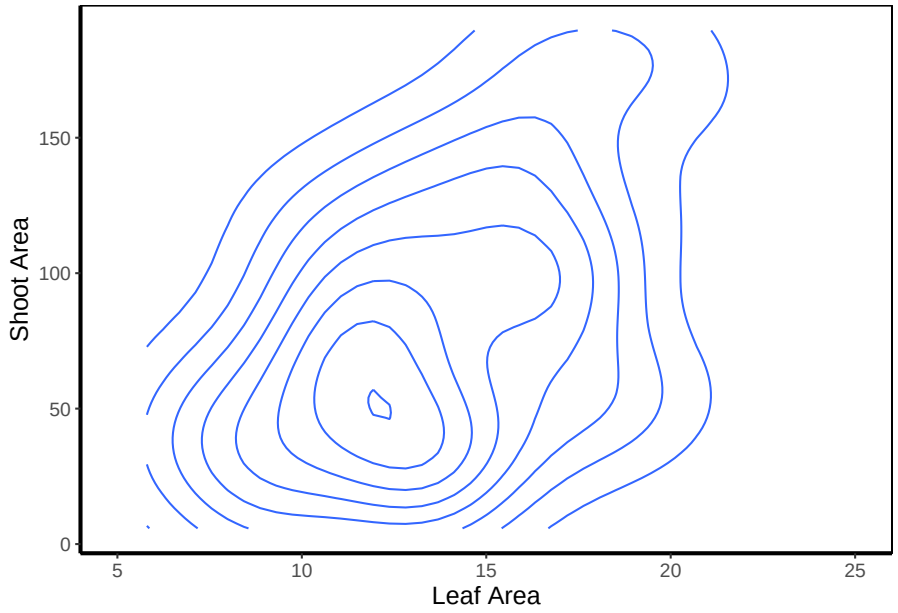
ggplot(aes(x = leafarea, y = shootarea), data = flower) +
    geom_hex() +
    labs(y = "Shoot Area", x = "Leaf Area") +
    coord_cartesian(xlim = c(5,25)) +
    theme_rbook()
```



5.4.10 Contour map

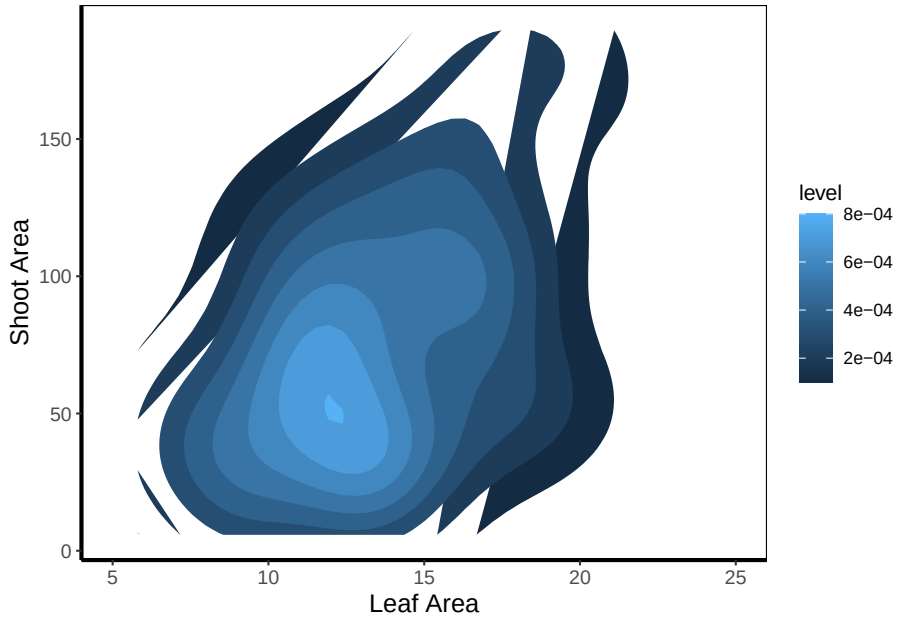
Similar to a heatmap we can make a contour map using `geom_density_2d()`. The way to read this figure is much the same way as you'd read a topographical map showing mountains or peaks. The central polygon represents the space (amongst shoot and leaf area) where there are most observations. As you "step" 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()
```



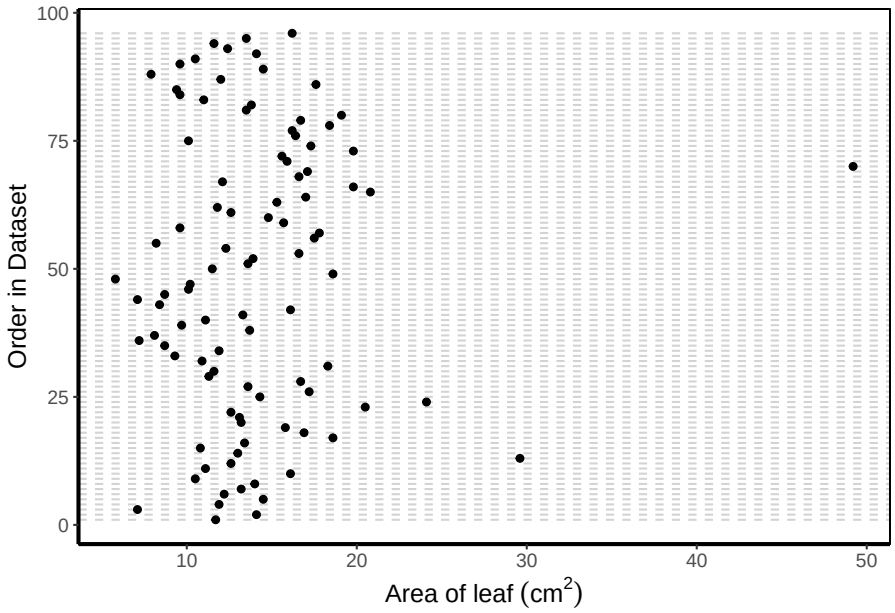
In this case it doesn't look too pretty given that we don't have a wide spread of data leading to partly formed polygons.

5.4.11 Cleveland dotplot

The cleveland dotplot is a figure well suited to detect unusual oberservations (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 horiztonal 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

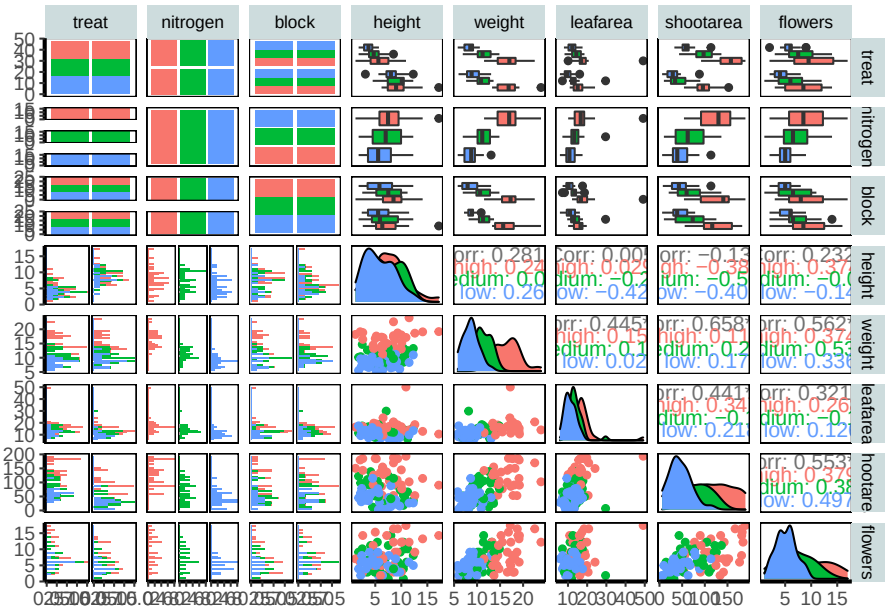
For the final figure we will need an additional package, called `GGally`. Within `GGally` is a function called `ggpairs()` which is equivalent to `pairs()` (part of base R). Both functions produce a scatterplot matrix of all variables of interest. Running diagonally down from left to right are plots 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.

```
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 here. If you want to see our solutions for this exercise you can find them here (don't peek at them too soon though!).

Chapter 6

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.

6.1 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) %+repla
##         theme(panel.border = element_blank(), panel.grid.major = element_blank(),
##             panel.grid.minor = element_blank(), axis.line = element_line(colour =
##                 size = rel(1)), legend.key = element_blank(),
##             strip.background = element_rect(fill = "white", colour = "black",
##                 size = rel(2)), complete = TRUE)
## }
## <bytecode: 0x7fb9babbd5f0>
## <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.

6.2 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)`).

```
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:

```
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.

```
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?

```
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.

```
# 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.

```
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`

```
porto_aberdeen_func <- multiply_columns(x = city$porto, y = city$aberdeen)
porto_aberdeen_func
##   [1] -0.419253670 -0.910957233 -0.532326087  1.716321636 -0.339464131
##   [6]  0.333632448  0.183033884  0.298921864 -0.670211868 -0.213764531
##  [11] -0.528739546  0.820495160 -0.090120181 -0.227659701 -0.532967191
##  [16] -0.046401966  3.143926281 -0.815863379 -0.157874028 -0.533109370
##  [21]  0.533074325 -0.363985997  0.131082480 -0.780084387  0.082563348
##  [26] -0.528360289  0.662443788 -3.776080807  2.001040382  0.266228083
##  [31] -0.519631898  0.009746243 -0.098948980  0.010610928  0.658479573
##  [36]  0.282858277  0.210615664 -0.135729682  0.085016588  0.643415400
##  [41] -1.123919564  0.097423292  0.012573030 -0.046716591  0.269130173
##  [46] -0.283013257 -0.052883613  0.148776192 -0.315916002 -0.091167412
##  [51] -0.014720866  2.065022128  3.410473028 -0.560276620  0.467105631
##  [56]  1.752729170  0.410934305  0.360705277  0.122095666  0.498641642
##  [61]  0.429711868  0.526192120  0.484735574  1.035217078 -0.164745132
##  [66] -0.036013588  1.475569359  1.905333043  0.192222470 -0.263071737
```

```
##  [71]  2.541001860 -0.013196036  0.789962787 -0.050136104  0.005903503
##  [76]  0.002309469 -0.253167292  0.015969110 -0.222058070 -0.203697370
##  [81] -0.553480371 -0.142200479 -0.111286319  0.188720122 -0.599459017
##  [86] -0.439905895 -0.678712743  0.458893908  0.099739805 -0.277243872
##  [91]  0.898528829  1.268034805 -0.703684191 -0.109726464  0.050154146
##  [96]  0.853301572  0.473474471  0.759643248 -0.449278991 -1.012464289
```

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]  0.0081174248 -1.0790187621  0.0876192797  0.7703837319 -3.1128419535
##  [16]  0.5439059134  1.8241759380 -2.8241128062  0.1395827605  2.6976504394
##  [21] -1.0471548429  0.1753196257 -0.0553889977  0.6991997343  0.1560719178
##  [26] -0.0805873355  2.6825902028  1.2256108068 -1.1067268490 -0.0155326133
##  [31]  0.7662484141  0.0003377045  0.1494598143  0.0157740360  3.6454143044
##  [36] -0.7264158440 -0.5873178549 -0.1153172705  0.0951497912  0.4342659384
##  [41] -0.2597199243 -0.1007186469 -0.0164872420 -0.8718883570  0.2930458062
##  [46]  0.7953359131 -0.4070342525  0.2057621238  0.4634782950  0.2615489082
##  [51]  0.0089647124 -0.2019771732 -0.5807726182  0.7571450256 -0.2183537996
```

```
##  [56]  2.0022719325  3.4766897213  0.0595399017  0.2254601117  0.3670975225
##  [61]  0.0321112861 -0.0231794332  0.5844854770  3.0666055451  0.1269319721
##  [66]  0.0019805649  2.0372250941  3.0121075838  0.2785756994  0.2855041687
##  [71] -1.1322660156  0.9805578365  0.1836145253  2.3116907882  0.0104708195
##  [76]  0.0181546405 -0.0070245199 -0.0113181656 -0.5510245929  0.1164797737
##  [81]  0.3109343299  0.1592398830  0.0673679504  0.5807288019  0.2636866847
##  [86]  0.0130752259 -0.1828129109 -0.5343313888  0.2427169745 -0.2249569378
##  [91]  0.4142372412  0.6983210393  1.4915613995  0.4722585951 -0.6253902086
##  [96] -0.1046335607  0.0921033386 -0.0634647342 -0.3312532496 -0.2242891403
```

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.

```
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 `NA`s 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.

6.3 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 `NA`s) 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)
```

6.4 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)`

6.5 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).

6.5.1 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:

```
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

```
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):

```
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 4^{th} column with the non-existent 5^{th} 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.

```
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.

6.5.2 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:

```
while(logical_condition){ expression }
```

A simple example of a while loop is:

```
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.

6.5.3 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)

6.5.4 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()`

```
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.

```
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.

6.6 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 here. If you want to see our solutions for this exercise you can find them here (don't peek at them too soon though!).

Chapter 7

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.

7.1 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).

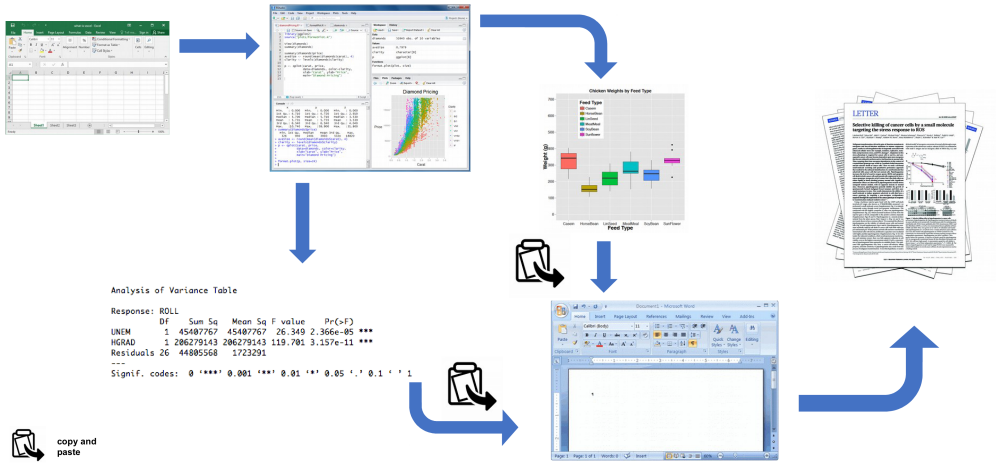
Technically, R markdown is a variant of another language (yet another language!) called Markdown and both are a type of 'markup' language. A markup language simply provides a way of creating an easy to read plain text file which can incorporate formatted text, images, headers and links to other documents. 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?!).

7.2 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:

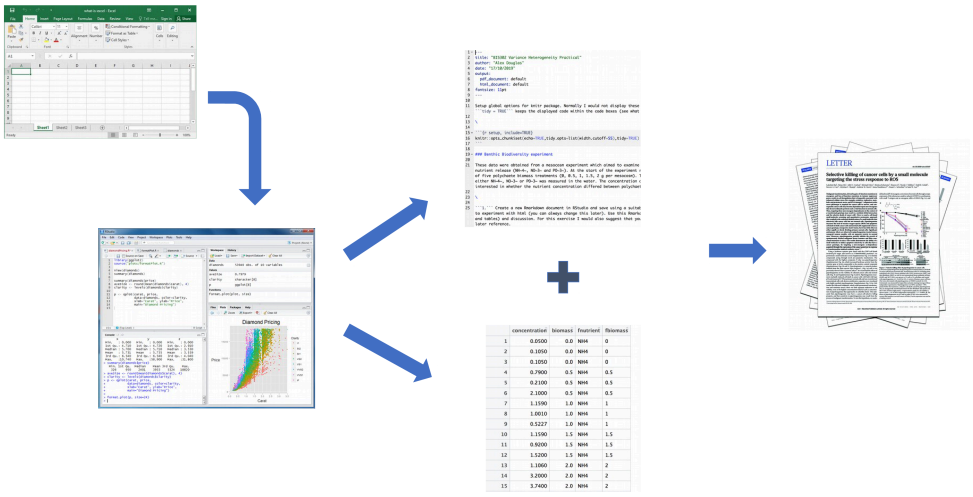


Your data is imported from your favourite spreadsheet software into RStudio (or R), you write your R code to explore and analyse your data, you save plots as external files, copy tables of analysis output and then manually combine all of this and your written prose into a single MS Word document (maybe 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.

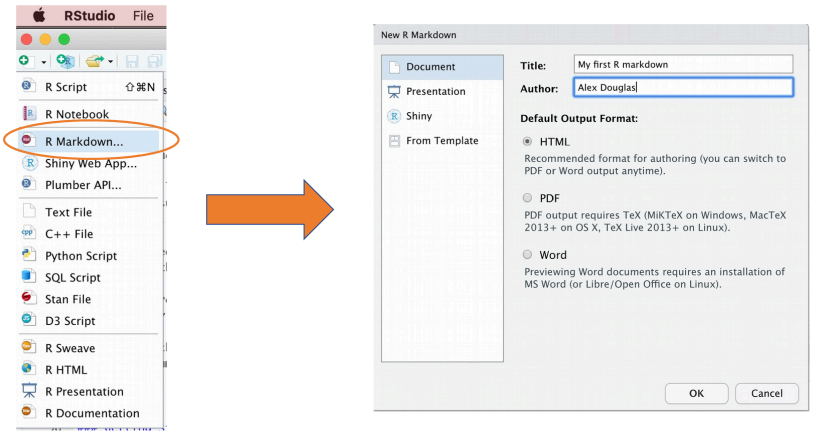
- Enhances transparency of your research. Your data and R markdown file can be included with your publication or thesis chapter as supplementary material or hosted on a GitHub repository (see the GitHub Chapter).
- Increases the efficiency of your workflow. If you need to modify or extend your current analysis you just need to update your R markdown document and these changes will automatically be included in your final document.

7.3 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.

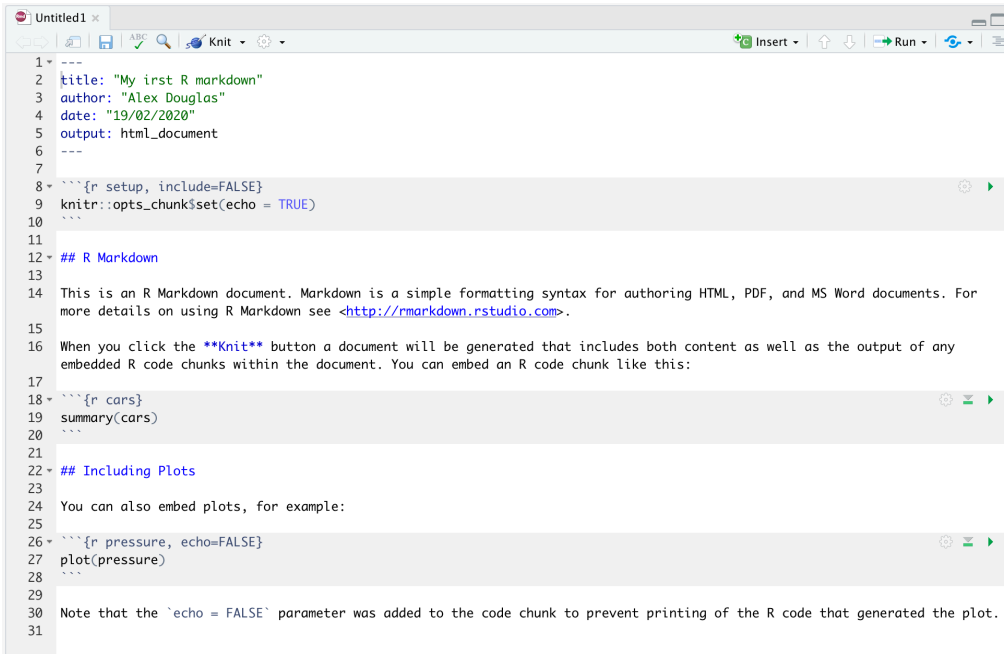
7.4 Create an R markdown document

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.



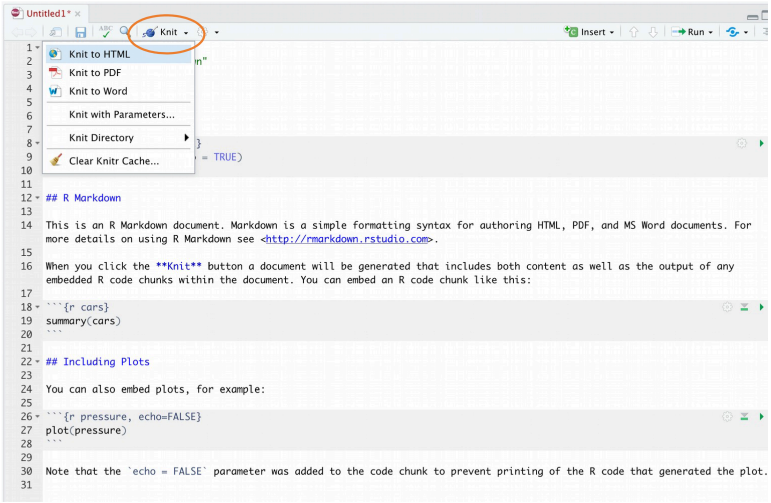
You will notice that when your new R markdown document is created it includes some

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.

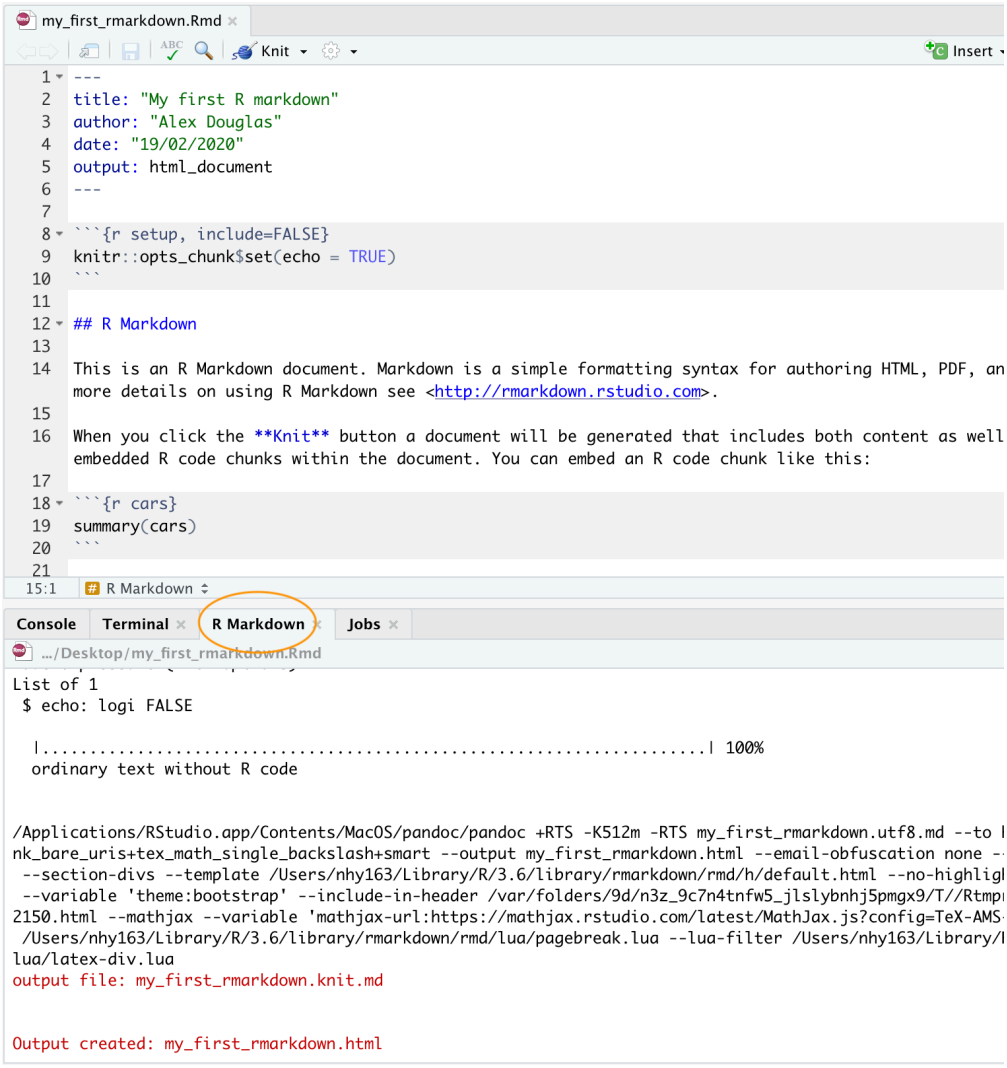


Once you've created your R markdown document it's good practice to save this file somewhere convenient. You can do this by selecting `File` -> `Save` from RStudio menu (or use the keyboard shortcut ctrl + s on Windows or cmd + s on a Mac) and enter an appropriate file name (maybe call it `my_first_rmarkdown`). Notice the file extension of your new R markdown file is `.Rmd`.

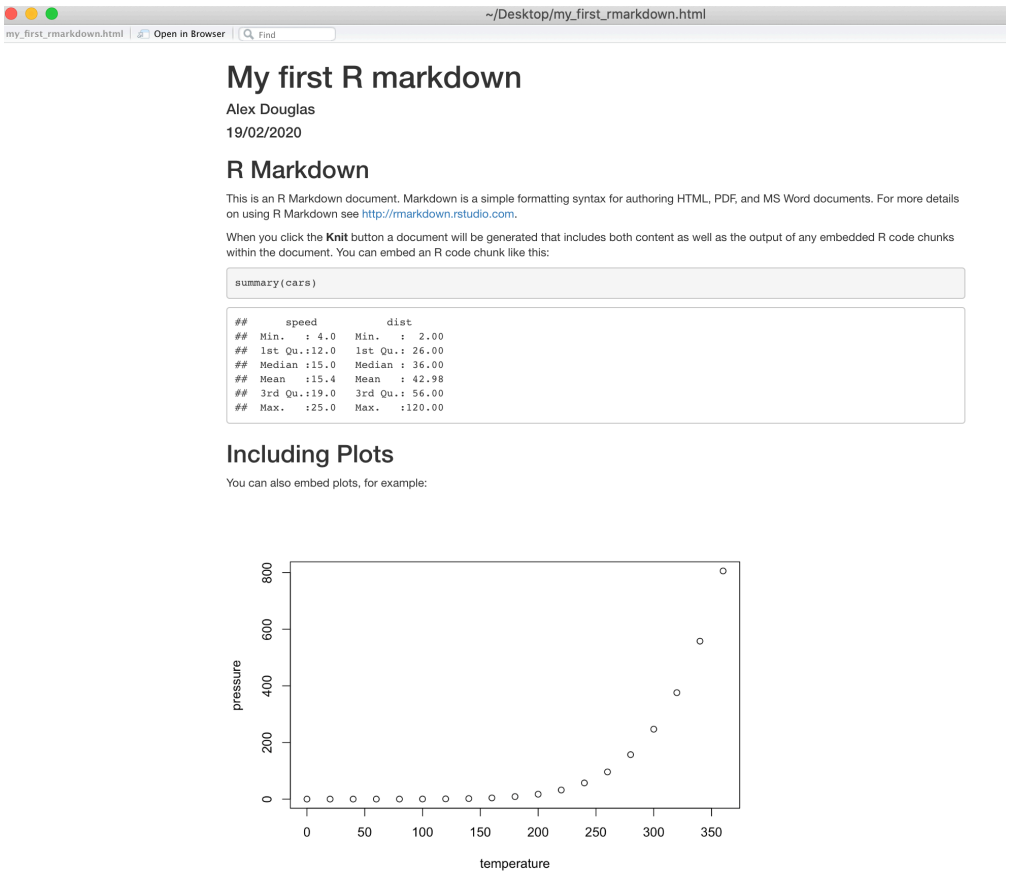
Now, to convert your `.Rmd` file to a HTML document click on the little black triangle next to the `Knit` icon at the top of the source window and select `knit to HTML`



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')

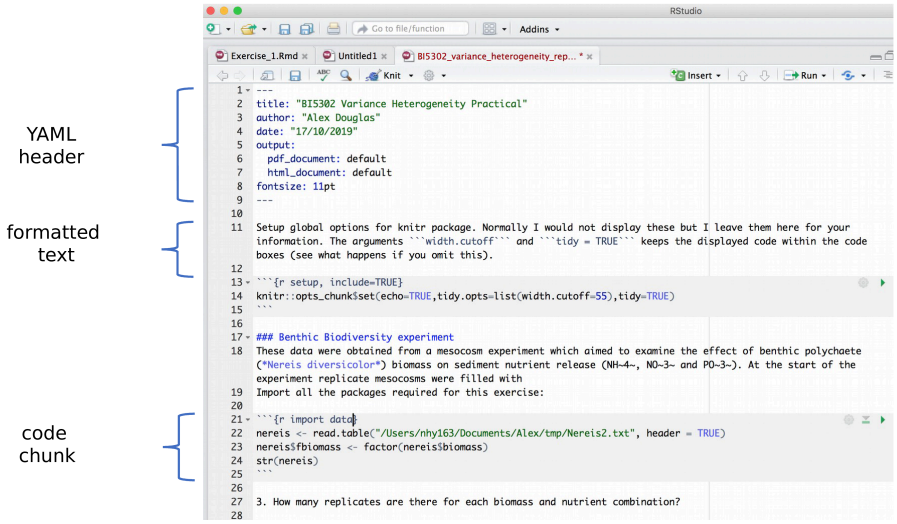
# alternatively if you don't want to load the rmarkdown package

rmarkdown::render('my_first_rmarkdown.Rmd', output_format = 'html_document')

# see ?render for more options
```

7.5 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.



7.5.1 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
---
```

7.5.2 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 mesocosm
were filled with 20 cm^2^ of **homogenised** marine sediment and assigned to one of fi
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:

Figure 7.1: Cute grey kitten

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")
```
```

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

7.5.3 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.

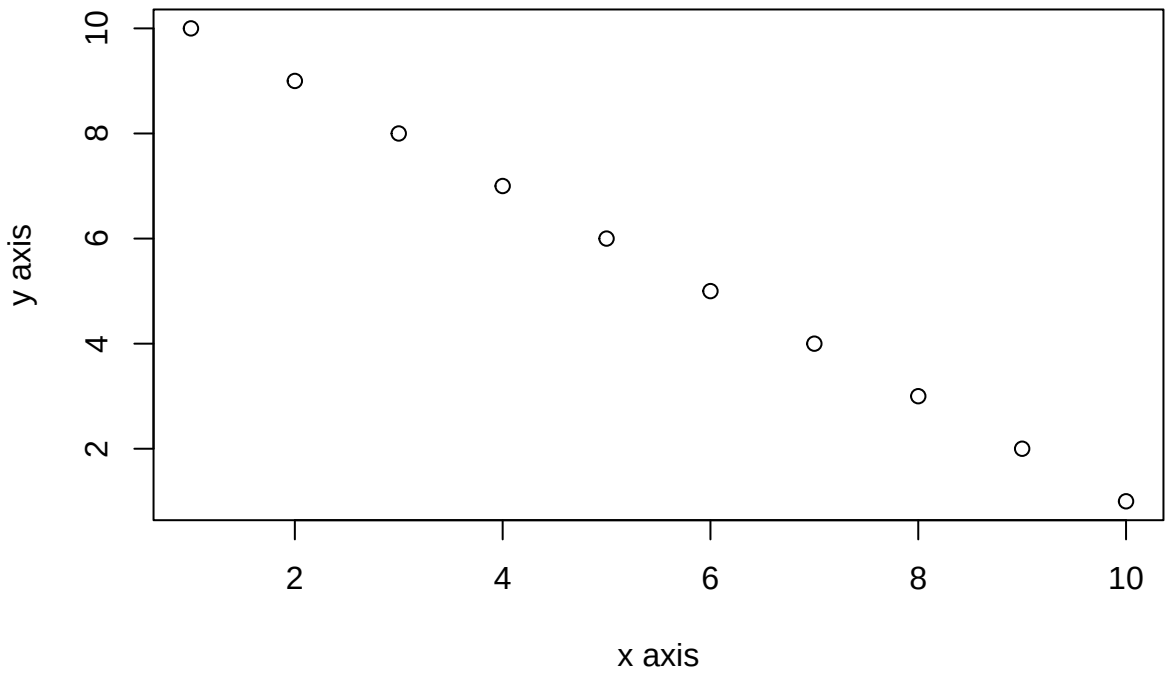
Chunk option	default value	Function
echo	`echo=TRUE`	If FALSE, will not display the code in the final document
results	`results='markup'`	If 'hide', will not display the code's results in the final document. If 'hold', will delay displaying all output pieces until the end of the chunk. If 'asis', will 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.

7.5.4 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")
```

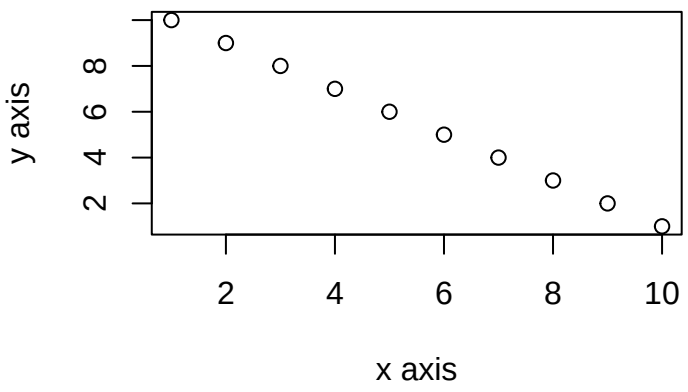


If you want to change the plot dimensions in the final document you can use the `fig.width=` and `fig.height=` chunk options (in inches!). You can also change the alignment of the figure using the `fig.align=` chunk option.

````
```{r, 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")
```

```
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")
```
````

```
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.

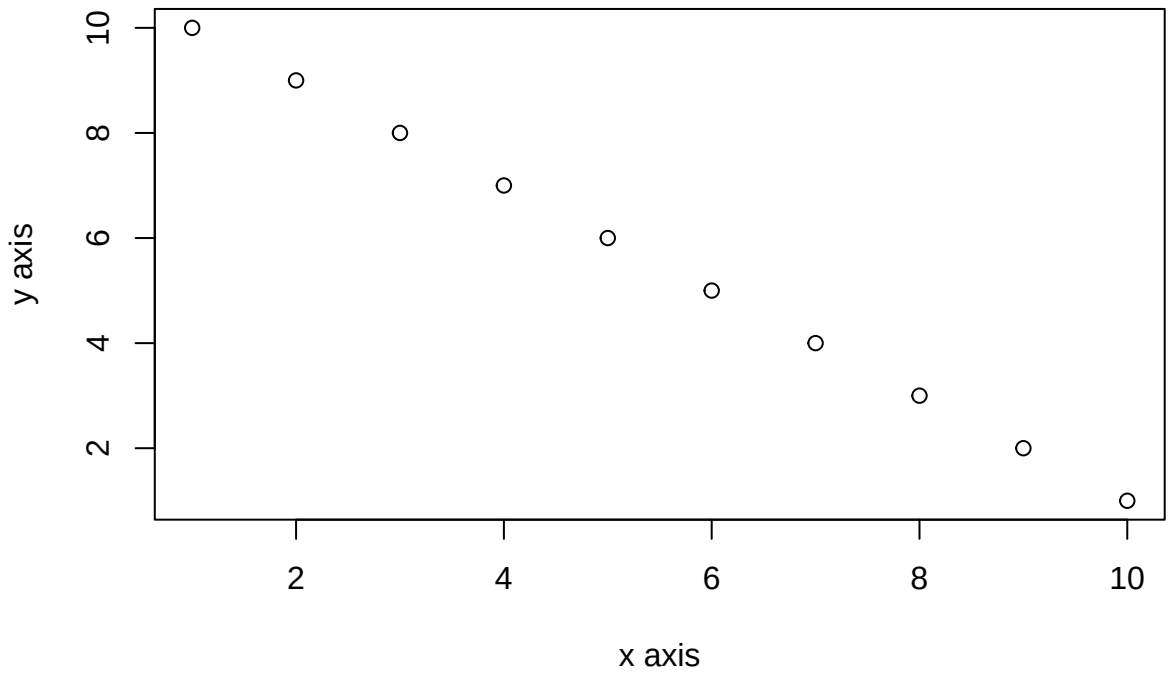


Figure 7.2: A simple plot

````
```{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")
```
````

```
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).

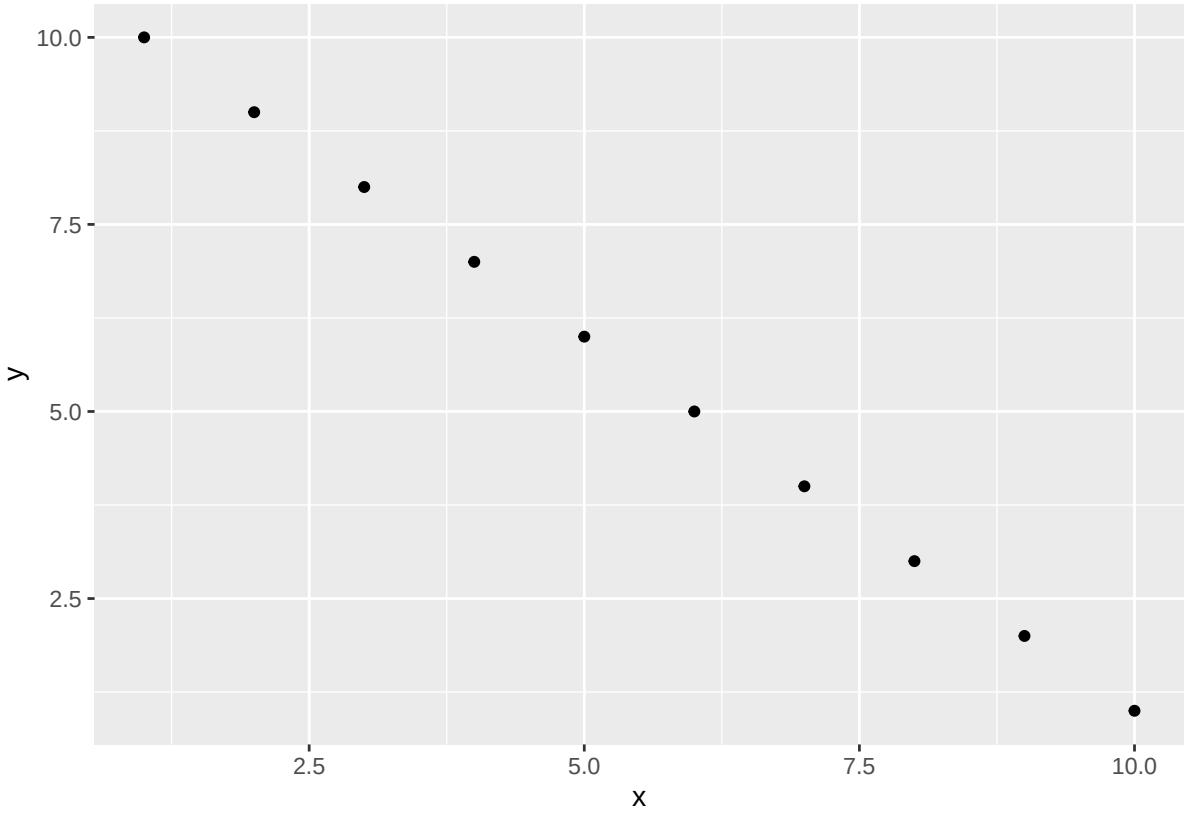
````
```{r, simple-ggplot}
x <- 1:10    # create an x variable
y <- 10:1    # create a y variable
```
````

```
dataf <- data.frame(x = x, y = y)

library(ggplot2)
ggplot(dataf, aes(x = x, y = y)) +
  geom_point()
```
```

```
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.
```

7.5.5 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

x	y
3	3
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.

7.5.6 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 fr
`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.

7.6 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)
```
````

7.7 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 8

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.

8.1 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' 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.

8.2 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

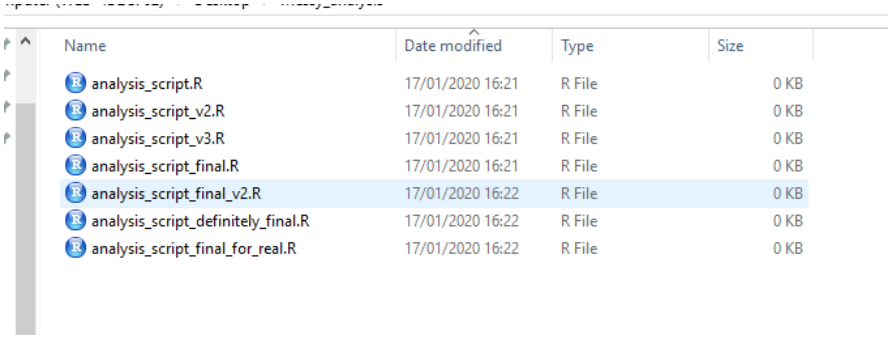
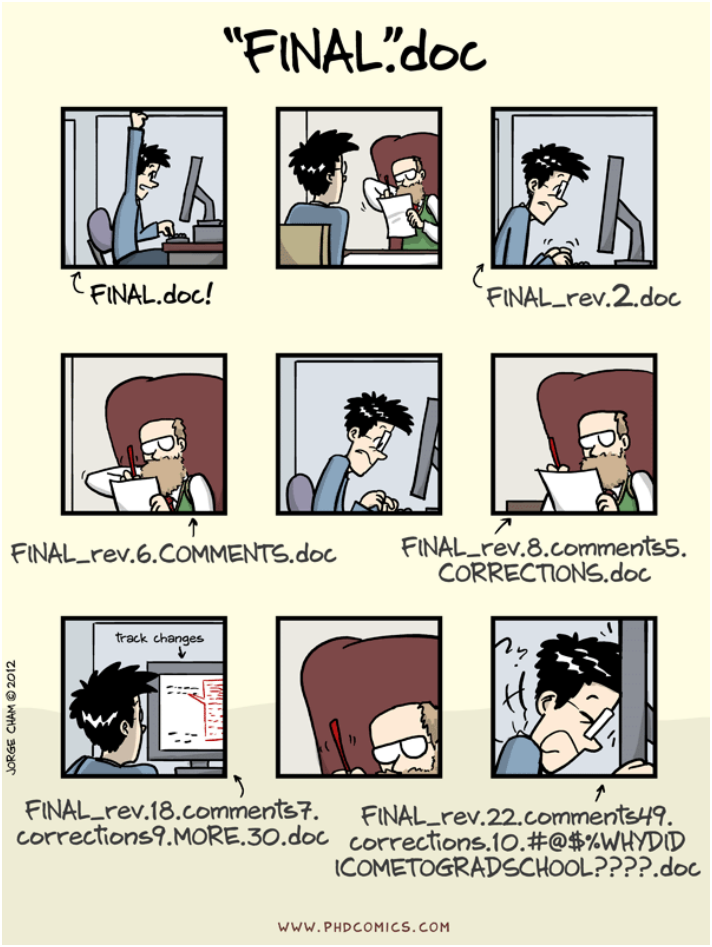


Figure 8.1: You need version control

usually arising from this (familiar?) scenario



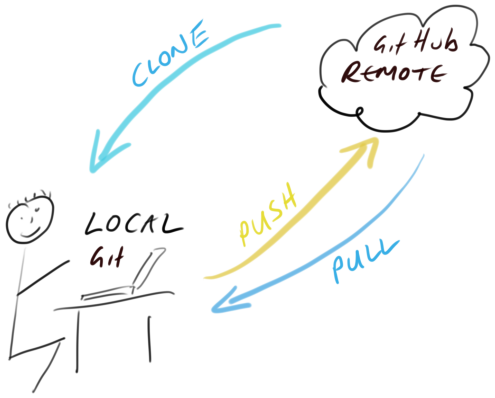
Version control automatically takes care of keeping a record of all the versions of a particular file and allows you to revert back to previous versions if you need to. Version control also helps you (especially the future you) keep track of all your files in a single place and it helps others (especially 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.

8.3 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.

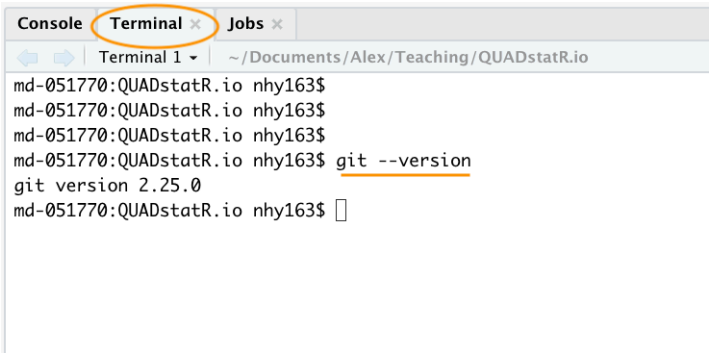


8.4 Getting started

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.

8.4.1 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).

8.4.2 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).

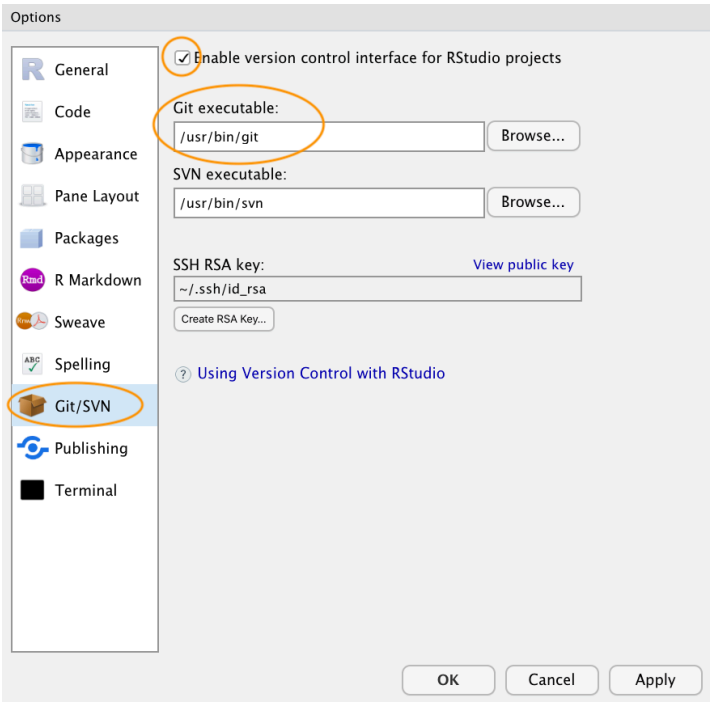
If this was successful, you should see no error messages from these commands. To verify that you have successfully configured Git type the following into the Terminal

```
git config --global --list
```

You should see both your `user.name` and `user.email` configured.

8.4.3 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.



8.4.4 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.

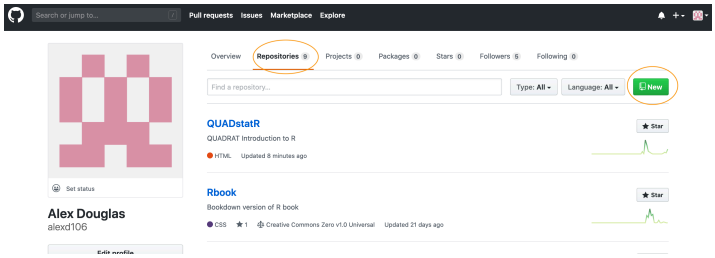
Once you've completed all those steps you should have both Git and GitHub setup up ready for you to use (Finally!).

8.5 Setting up a project in RStudio

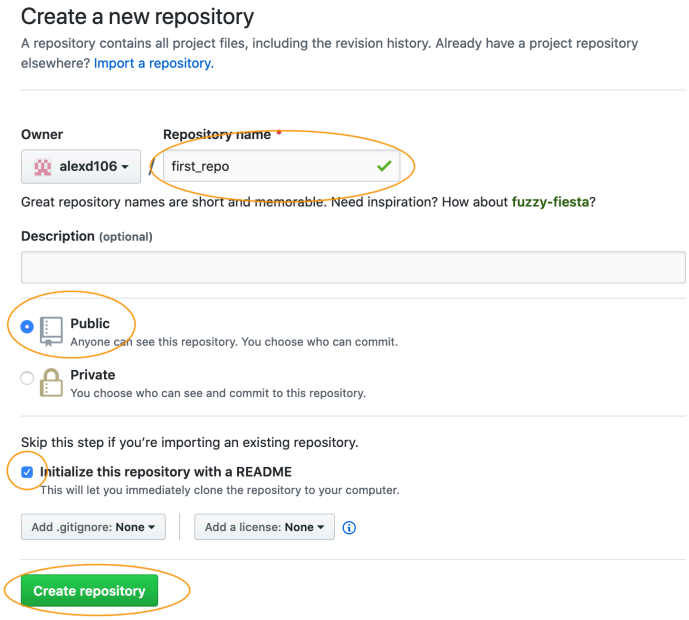
Now that you're all set up, let's create your first version controlled RStudio project. There are a couple of different approaches you can use to do this. You can either setup a remote GitHub repository first then connect an RStudio project to this repository (we'll call this Option 1). Another option 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.

8.5.1 Option 1 - GitHub first

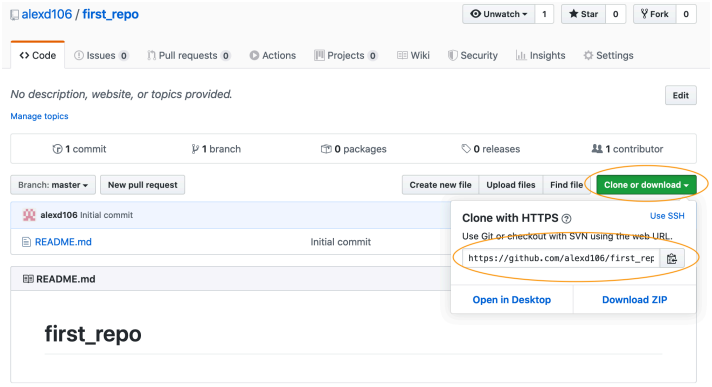
To use the GitHub first approach you will first need to create a **repository (repo)** on GitHub. Go to your GitHub page and sign in if necessary. Click on the 'Repositories' tab at the top and then on the green 'New' button on the right



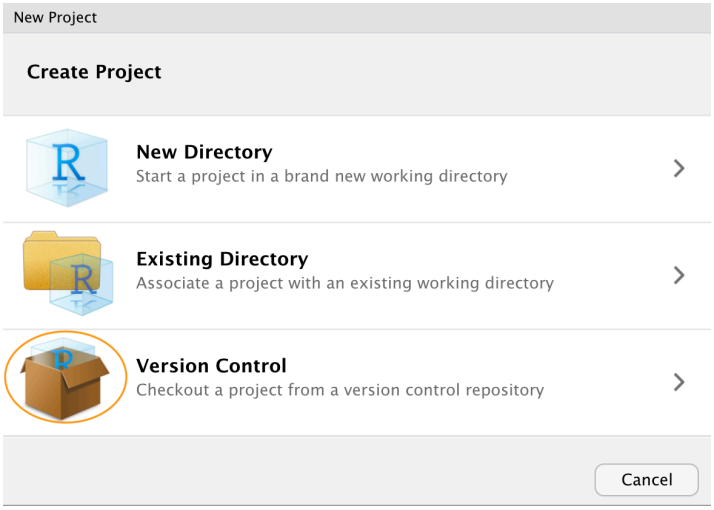
Give your new repo a name (let's call it `first_repo` for this Chapter), select 'Public', tick on the 'Initialize this repository with a README' (this is important) and then click on 'Create repository' (ignore the other options for now).



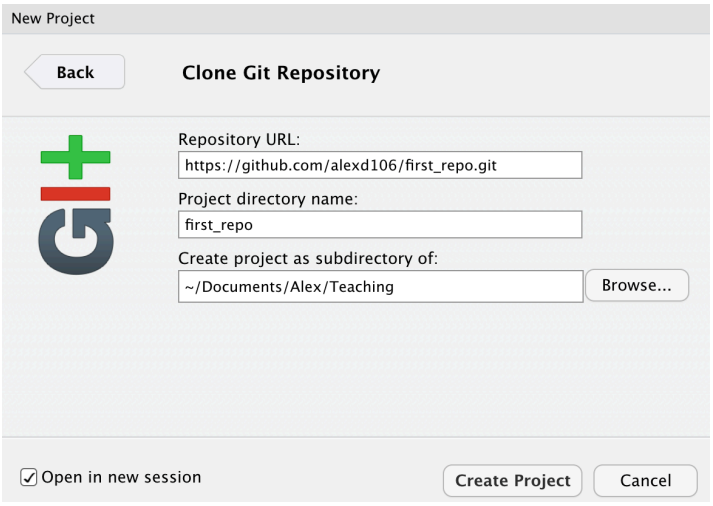
Your new GitHub repository will now be created. Notice the README has been rendered in GitHub and is in markdown (.md) format (see Chapter 8 on R markdown if this doesn't mean anything to you). Next click on the green 'Clone or Download' button and copy the `https//...` URL that pops up for later (either highlight it all and copy or click on the copy to clipboard icon to the right).



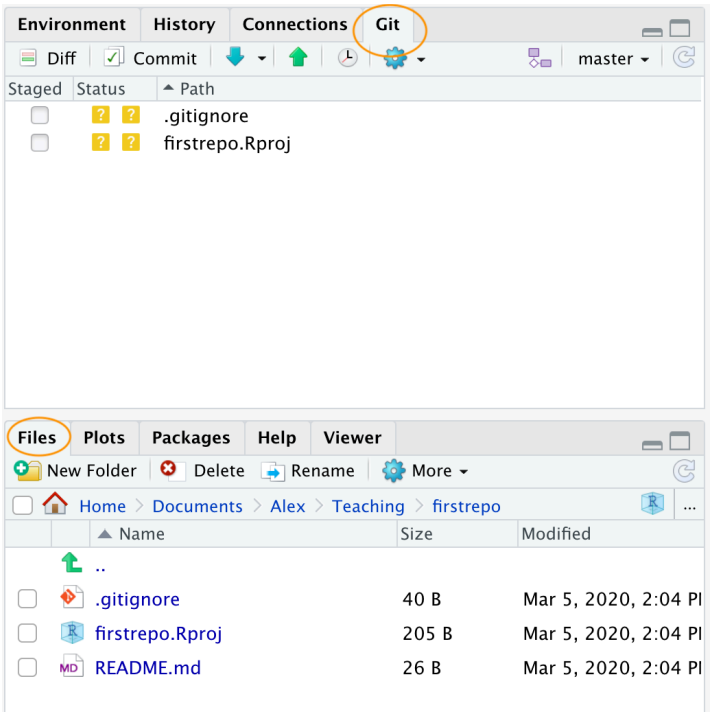
Ok, we now switch our attention to RStudio. In RStudio click on the `File` -> `New Project` menu. In the pop up window select `Version Control`.



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.



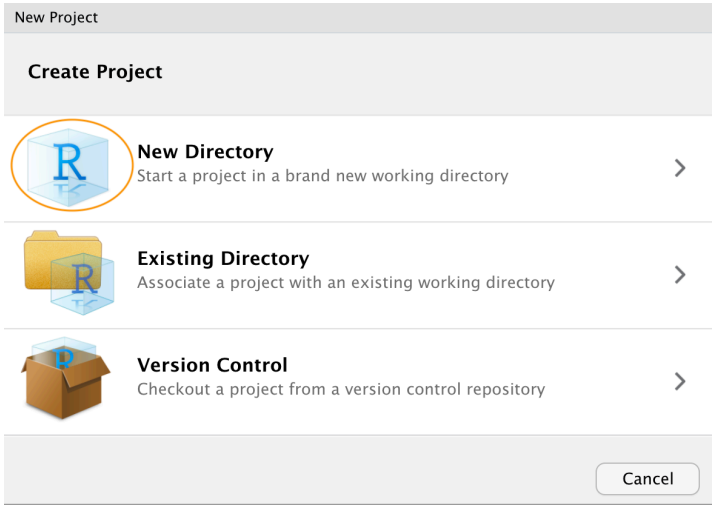
RStudio will now create a new directory with the same name as your repository on your local computer and will then **clone** your remote repository to this directory. The directory will contain three new files; `first_repo.Rproj` (or whatever you called your repository), `README.md` and `.gitignore`. 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.



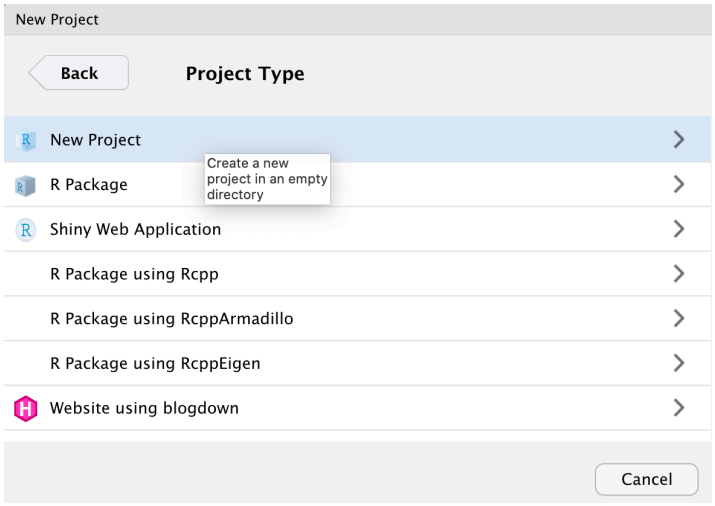
8.5.2 Option 2 - RStudio first

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.

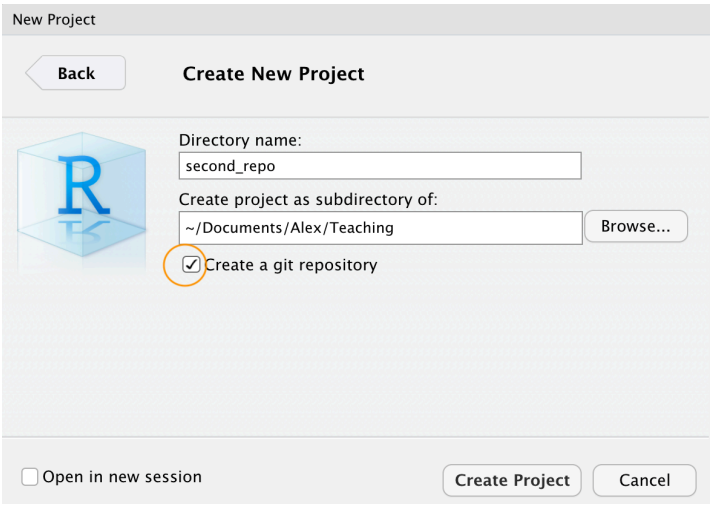
In RStudio click on the `File -> New Project` menu and select the `New Directory` option.



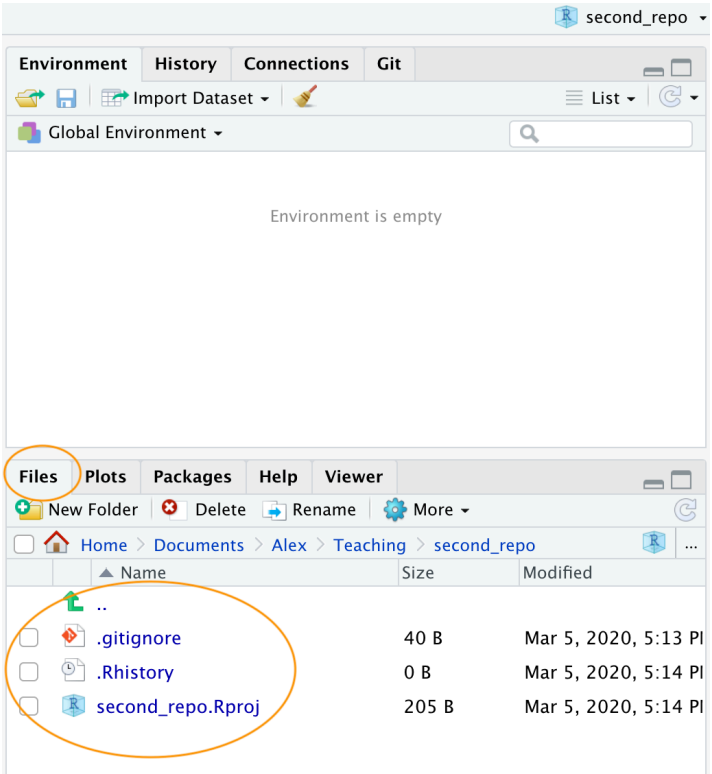
In the pop up window select the `New Project` option



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

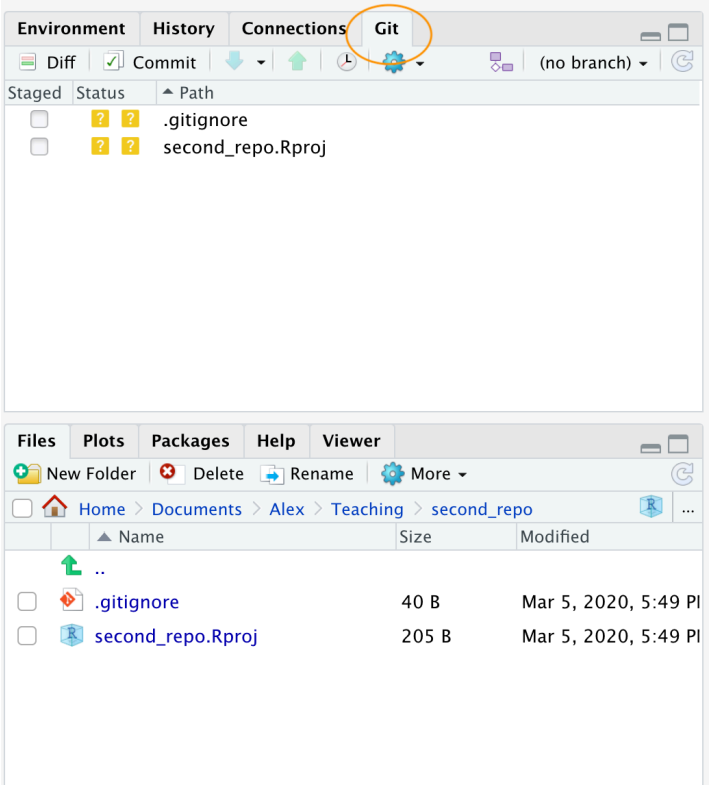


This will create a version controlled directory called `second_repo` on your computer that contains two files, `second_repo.Rproj` and `.gitignore` (there might also be a `.Rhistory` file but ignore this). You can check this by looking in the `Files` tab in RStudio (usually in the bottom right pane).

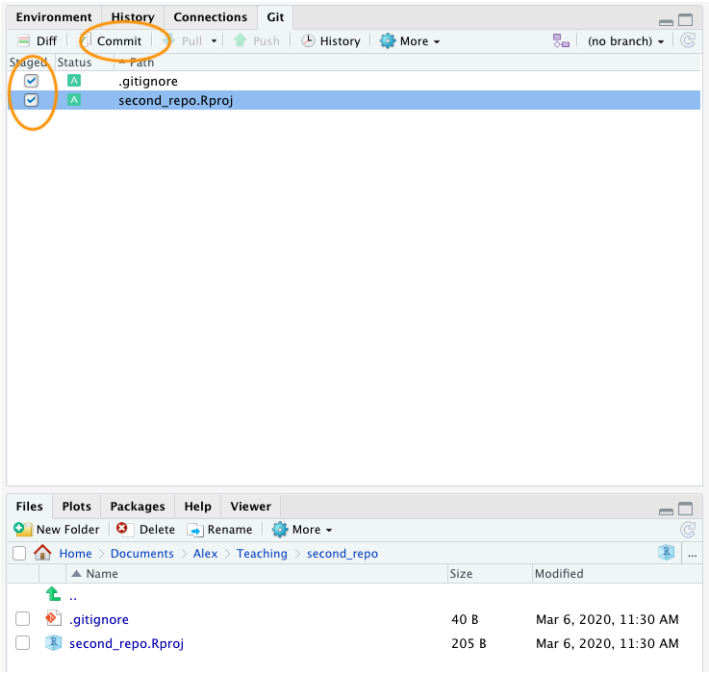


OK, before we go on to create a repository on GitHub we need to do one more thing - we need to place our `second_repo.Rproj` and `.gitignore`files under version control. Unfortunately we haven't covered this in detail yet so just follow the next few instructions (blindly!) and we'll revisit them in the Using Git section of this Chapter.

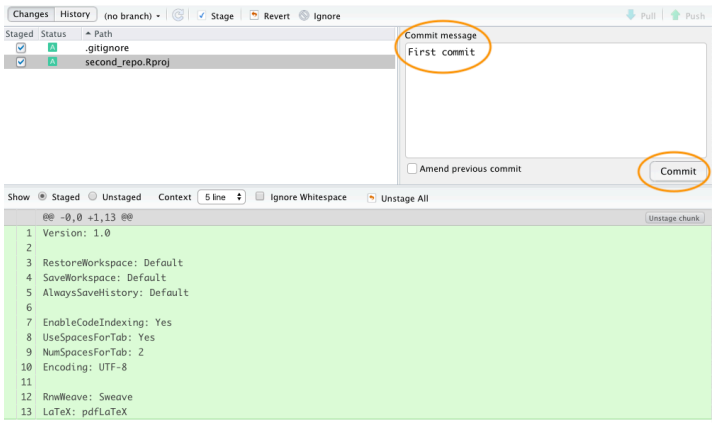
To get our two files under version control click on the 'Git' tab which is usually in the top tight pane in RStudio



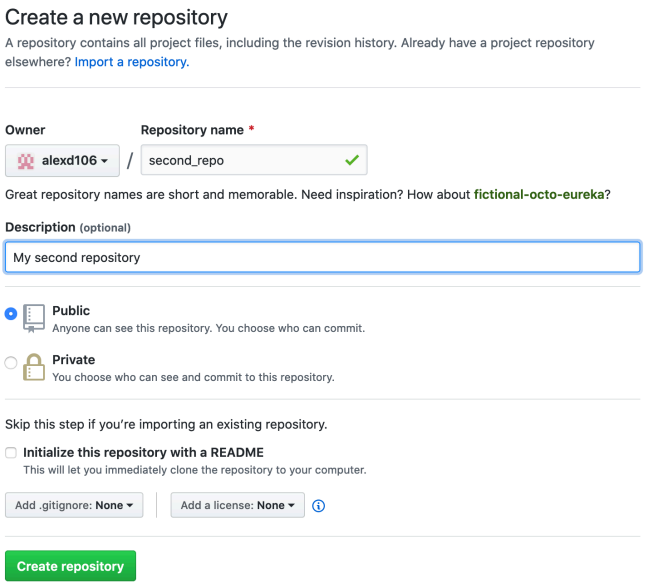
You can see that both files are listed. Next, tick the boxes under the 'Staged' column for both files and then click on the 'Commit' button.



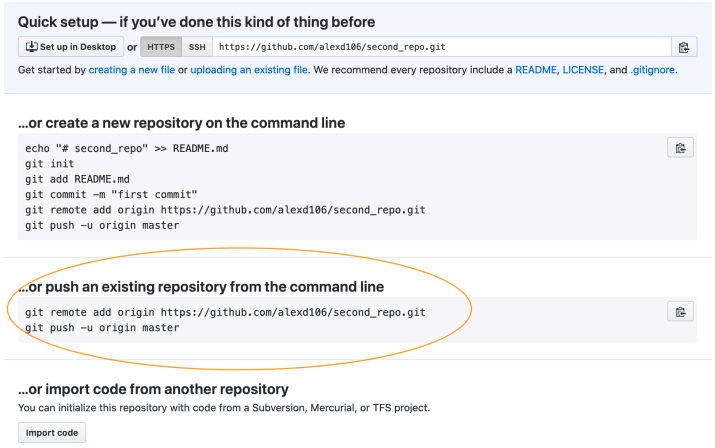
This will take you to the 'Review Changes' window. Type in the commit message 'First commit' in the 'Commit message' window and click on the 'Commit' button. A new window will appear with some messages which you can ignore for now. Click 'Close' to close this window and also close the 'Review Changes' 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.



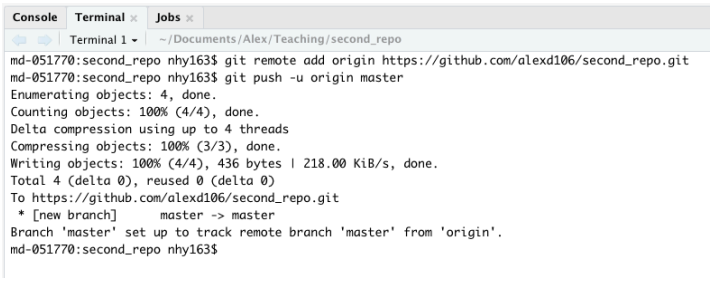
Highlight and copy the first line of code (note: yours will be slightly different as it will include your GitHub username not mine)

```
git remote add origin https://github.com/alexd106/second_repo.git
```

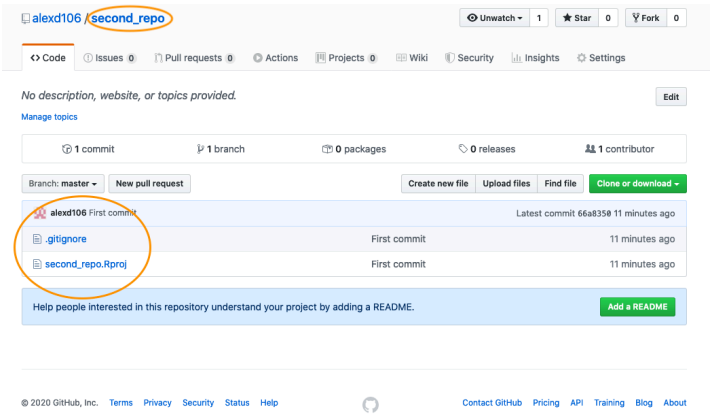
Switch to RStudio, click on the 'Terminal' tab and paste the command into the Terminal. Now go back to GitHub and copy the second line of code

```
git push -u origin master
```

and paste this into the Terminal in RStudio. You should see something like this

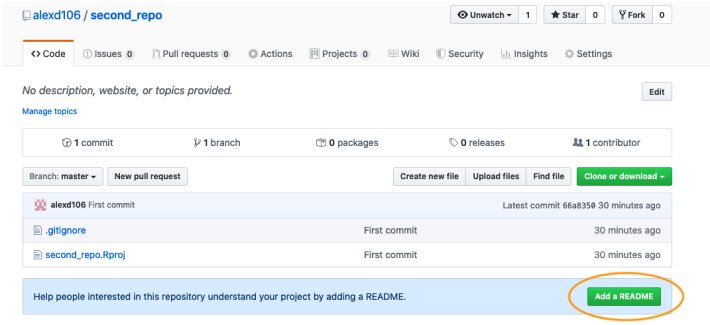


If you take a look at your repo back on GitHub (click on the `/second_repo` link at the top) you will see the `second_repo.Rproj` and `.gitignore` files have now been **pushed** to GitHub from your local repository.

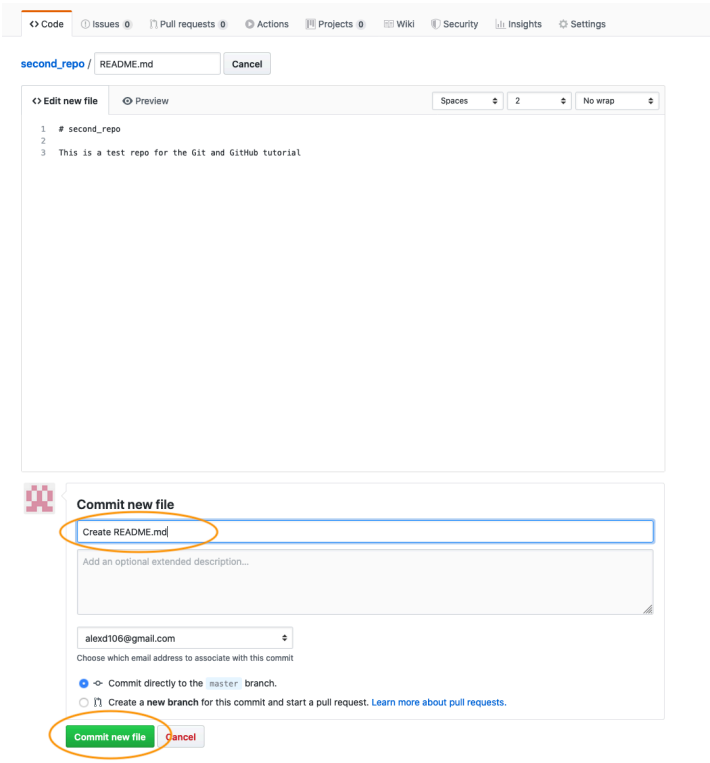


The last thing we need to do is create and add a **README** file to your repository. A README file describes your project and is written using the same Markdown language you learned in the R markdown Chapter. A good README file makes it easy for others (or the future you!) to use your code and reproduce 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

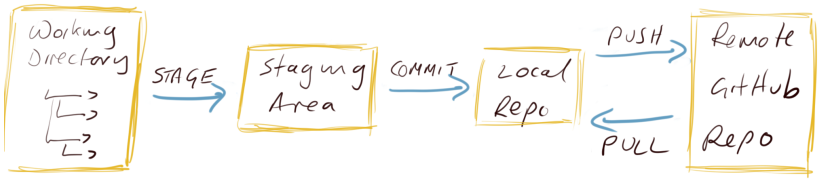
over time.

8.6 Using Git

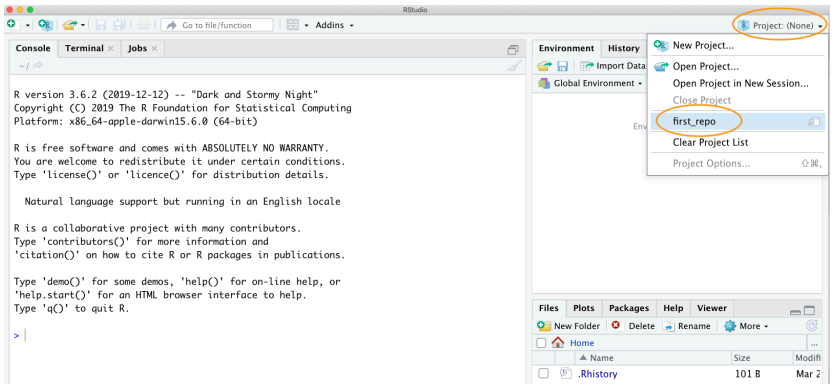
Now that we have our project and repositories (both local and remote) set up, it's finally time to learn how to use Git in RStudio!

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

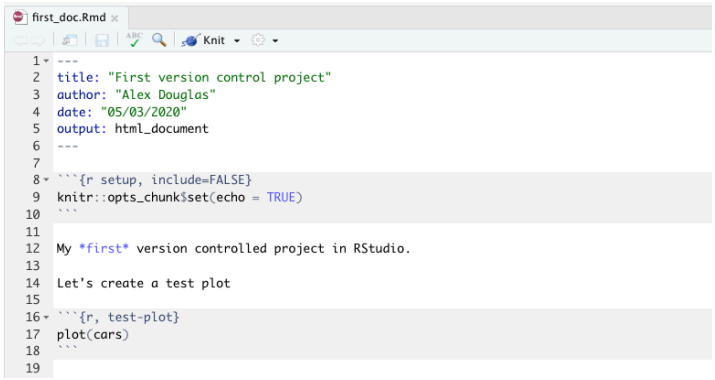


OK, let's go through an example to help clarify this workflow. In RStudio open up the `first_repo.Rproj` you created previously during Option 1. Either use the `File -> Open Project` menu or click on the top right project icon and select the appropriate project.

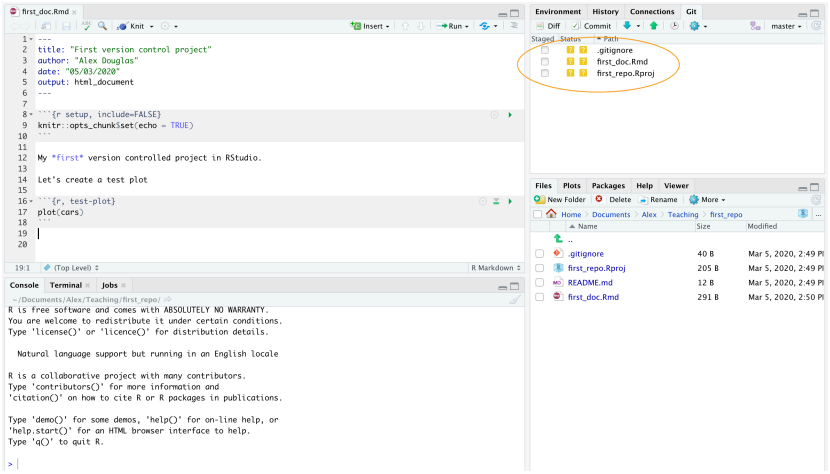


Create an R markdown document inside this project by clicking on the `File -> New File -> R markdown` menu (remember from the R markdown Chapter?).

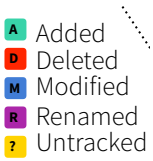
Once created, we can delete all the example R markdown code (except the YAML header) as usual and write some interesting R markdown text and include a plot. We'll use the inbuilt `cars` dataset to do this. Save this file (cmd + s for Mac or ctrl + s in Windows). Your R markdown document should look something like the following (it doesn't matter if it's not exactly the same).



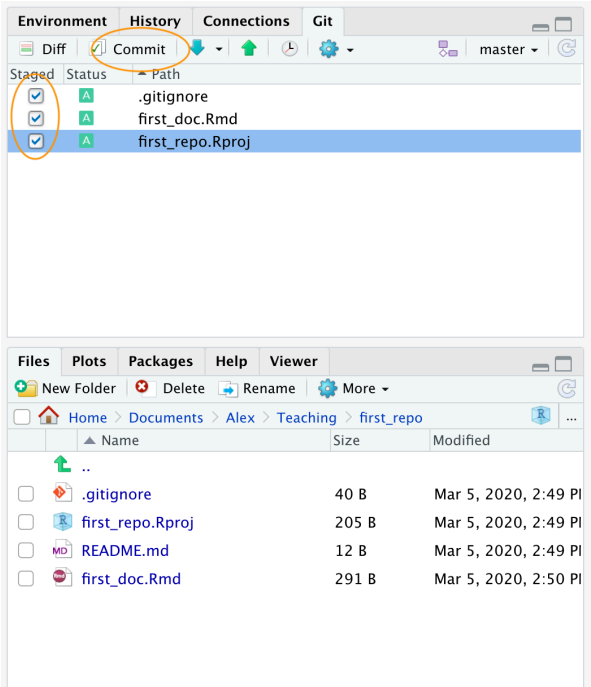
Take a look at the 'Git' tab which should list your new R markdown document (`first_doc.Rmd` in this example) along with `first_repo.Rproj`, and `.gitignore` (you created these files previously when following Option 1).



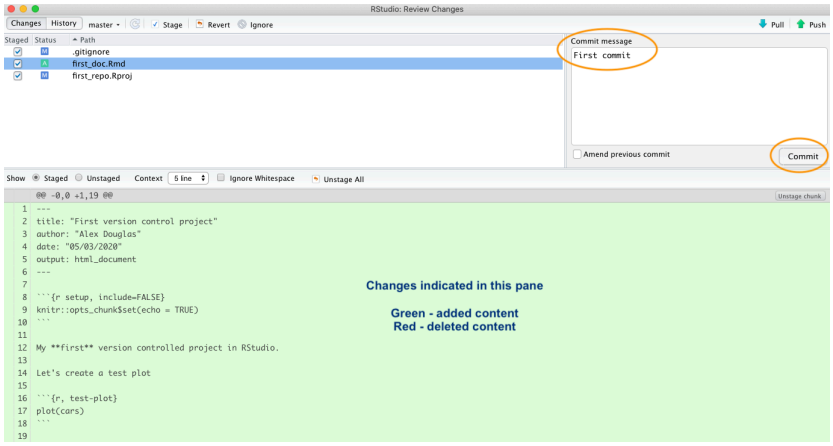
Following our workflow, we now need to **stage** these files. To do this tick the boxes under the 'Staged' column for all files. Notice that there is a status icon next to the box which gives you an indication of how the files were changed. In our case all of the files are to be added (capital A) as we have just created them.



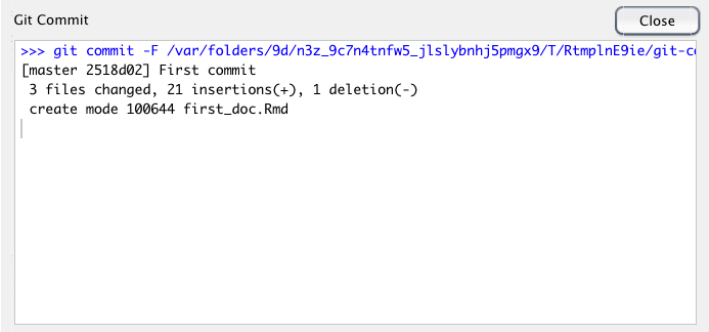
After you have staged the files the next step is to **commit** the files. This is done by clicking on the 'Commit' button.



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.

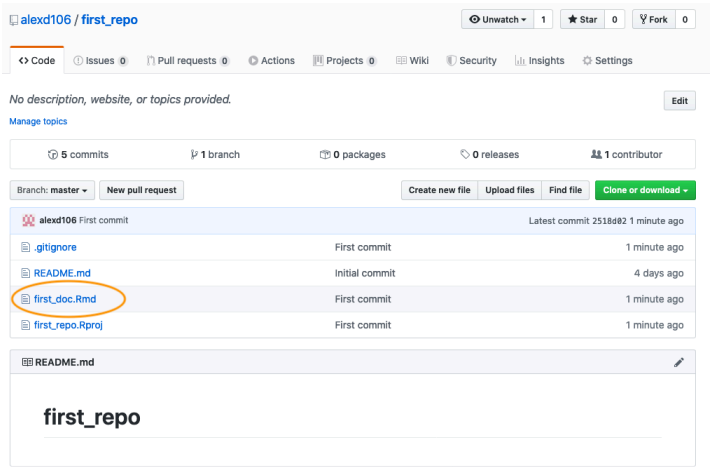


A summary of the commit you just performed will be shown. Now click on the 'Close' button to return to the 'Review Changes' window. Note that the staged files have now been removed.

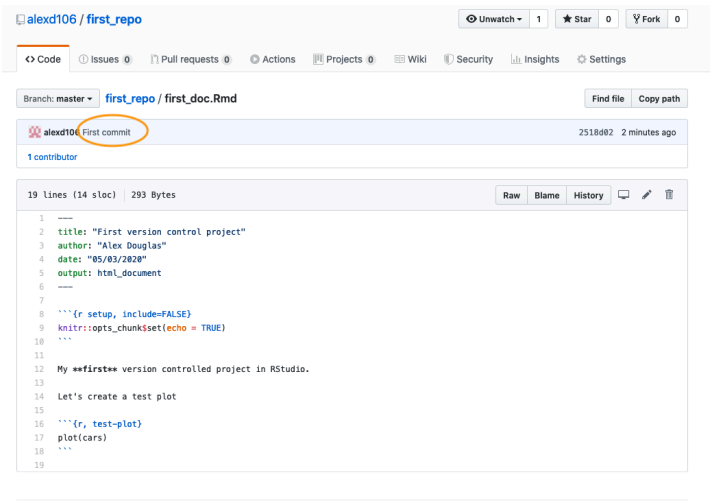


Now that you have committed your changes the next step is to **push** these changes to GitHub. Before you push your changes it's good practice to first **pull** any changes from GitHub. This is especially important if both you and your collaborators are working on the same files as it keeps you local 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.

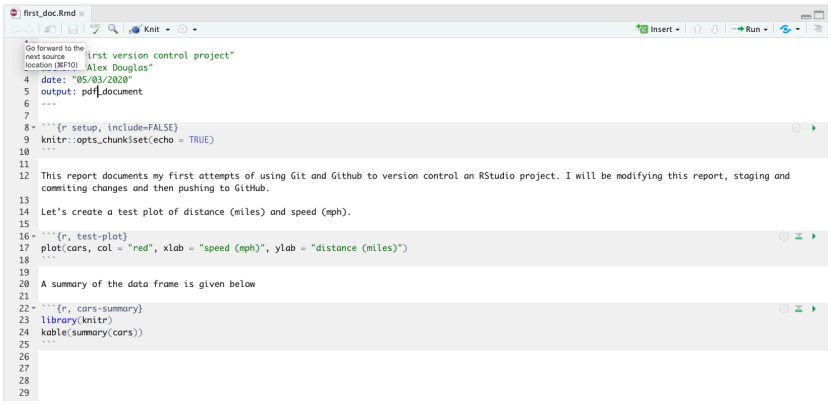


8.6.1 Tracking changes

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.

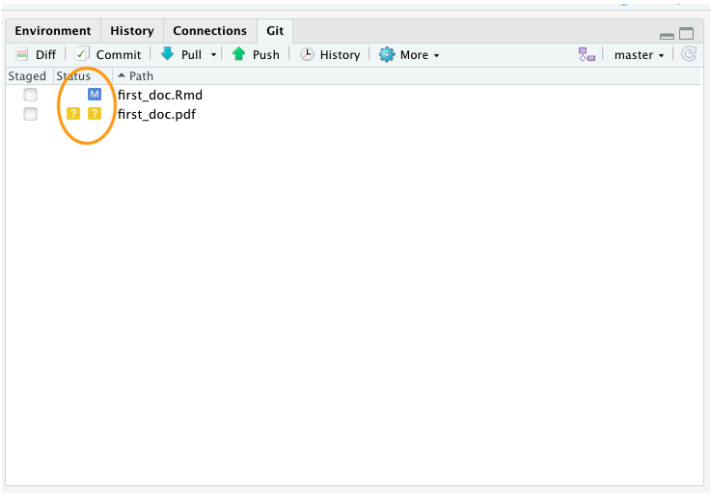
In RStudio open up the `first_repo.Rproj` file you created previously (if not already open) then open the `first_doc.Rmd` file (click on the file name in the `Files` tab in RStudio).

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`.

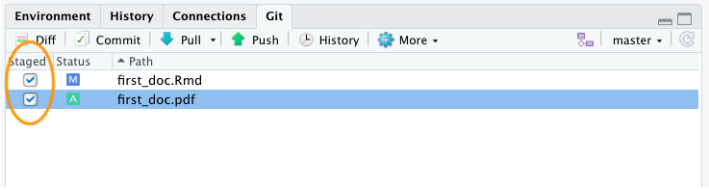


Now save these changes and then click the `knit` button to render to pdf. A new pdf file named `first_doc.pdf` will be created which you can view by clicking on the file name in the `Files` tab in RStudio.

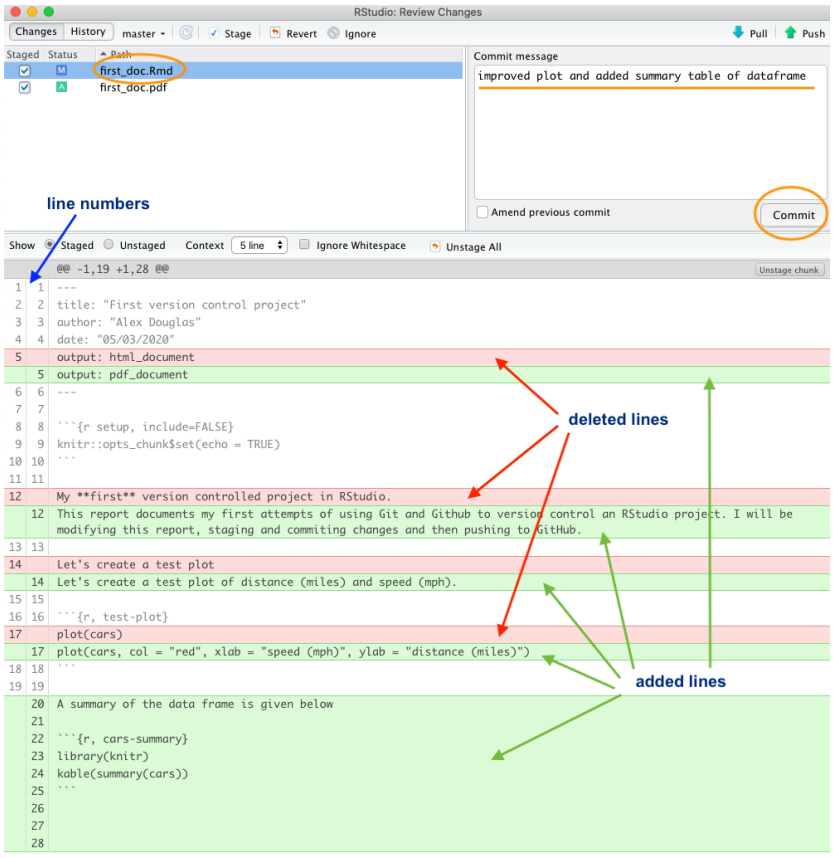
Notice that these two files have been added to the `Git` tab in RStudio. The status icons indicate that the `first_doc.Rmd` file has been modified (capital M) and the `first_doc.pdf` file is currently untracked (question mark).



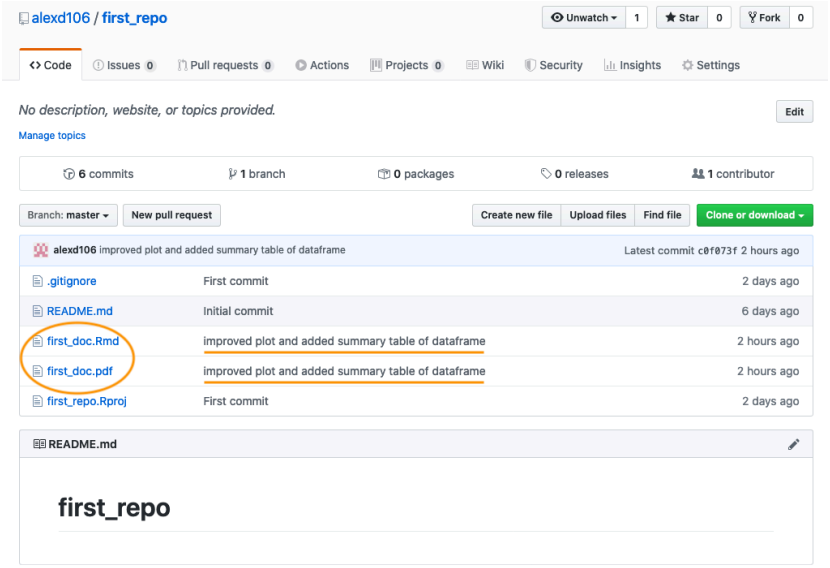
To stage these files tick the 'Staged' box for each file and click on the 'Commit' button to take you to the 'Review Changes' window



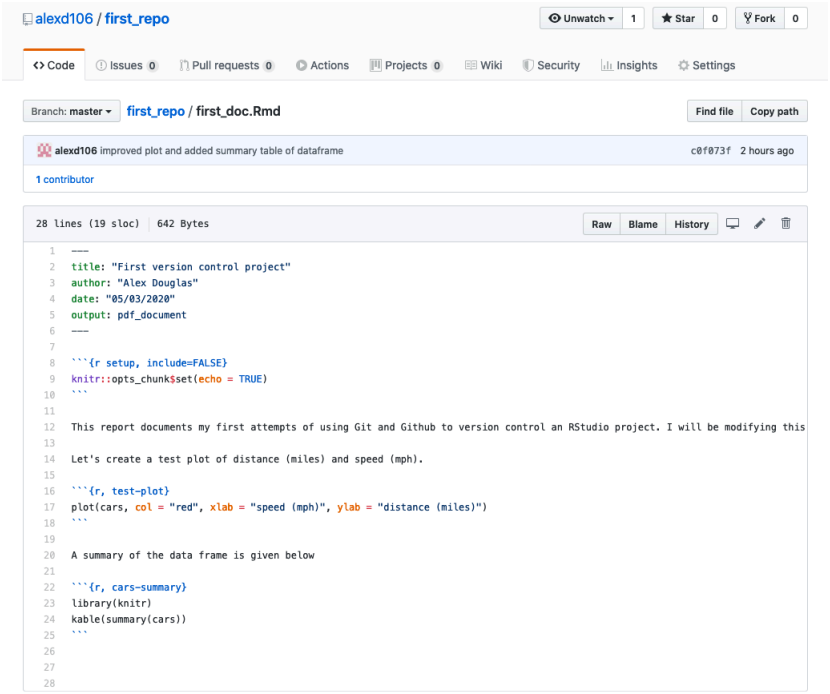
Before you commit your changes notice the status of `first_doc.pdf` has changed from untracked to added (A). You can view the changes you have made to the `first_doc.Rmd` by clicking on the file name in the top left pane which will provide you with a useful summary of the changes in the bottom pane (technically 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.



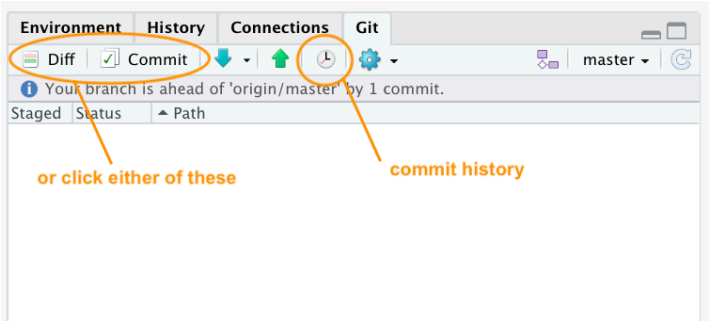
To view the changes in `first_doc.Rmd` click on the file name for this file.



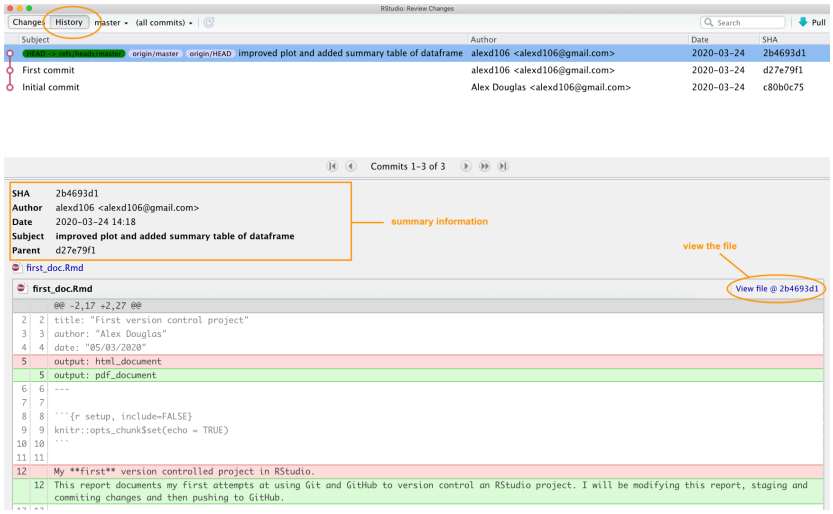
8.6.2 Commit history

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.

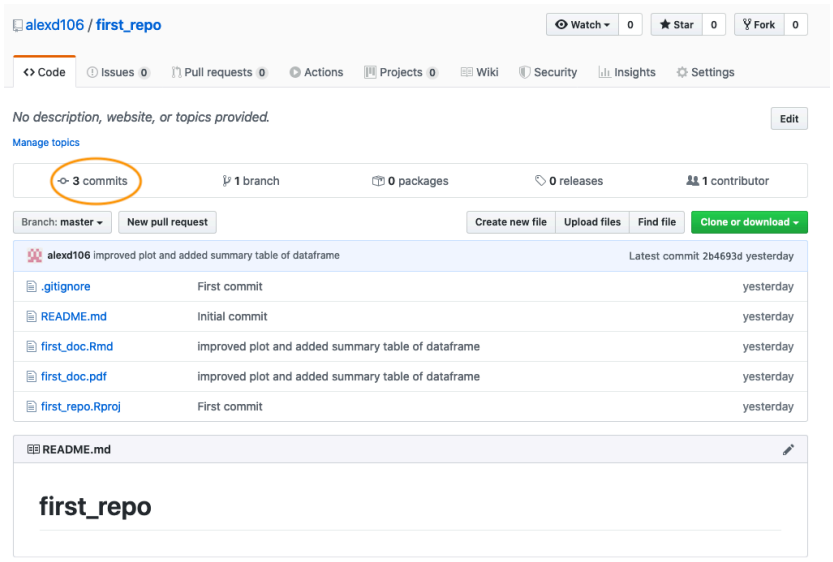
To view your commit history in RStudio click on the 'History' button (the one that looks like a clock) in the Git pane to bring up the history view in the 'Review Changes' window. You can also click on the 'Commit' or 'Diff' buttons which takes you to the same window (you just need to additionally click on the 'History' button in the 'Review Changes' window).



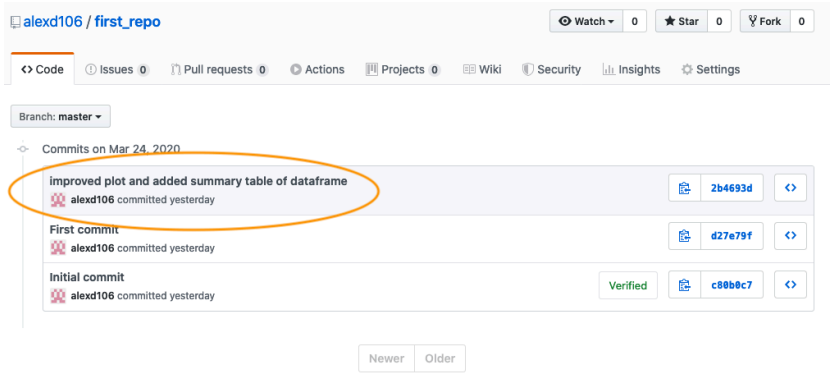
The history window is split into two parts. The top pane lists every commit you have made in this repository (with associated commit messages) starting with the most recent one at the top and oldest at the bottom. You can click on each of these commits and the bottom pane shows you the changes you have 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').



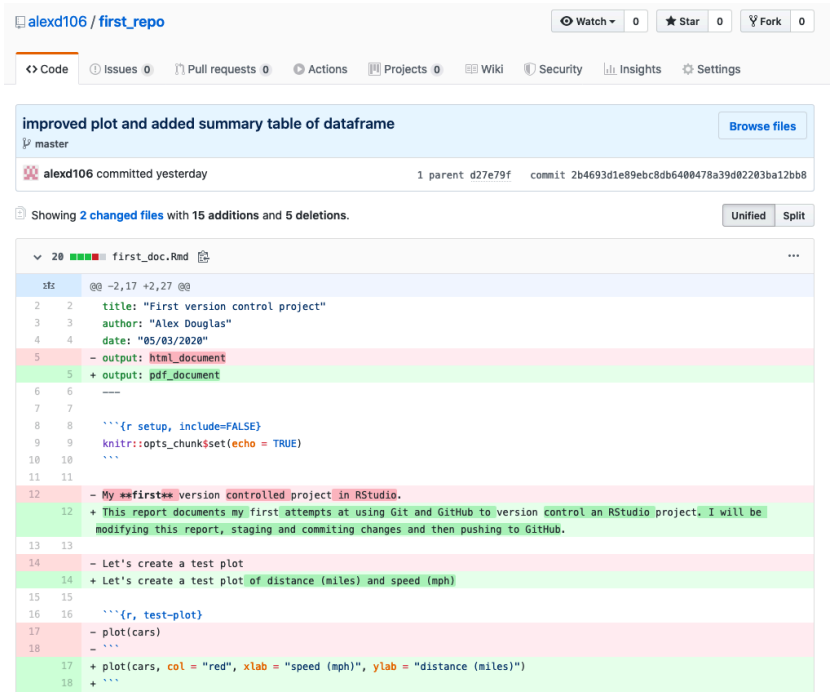
You can also view your commit history on GitHub website but this will be limited to only those commits you have already pushed to GitHub. To view the commit history navigate to the repository and click on the 'commits' link (in our case the link will be labelled '3 commits' as we have made 3 commits).



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.

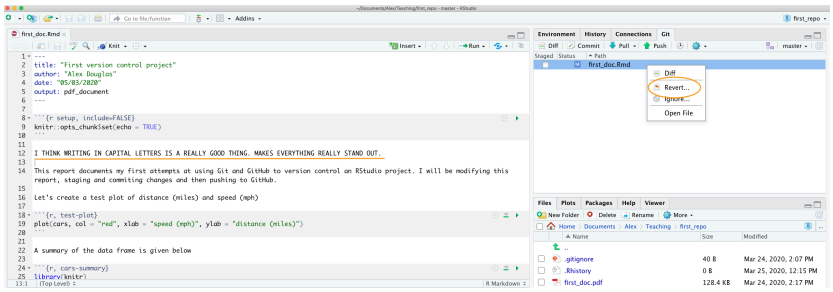


8.6.3 Reverting changes

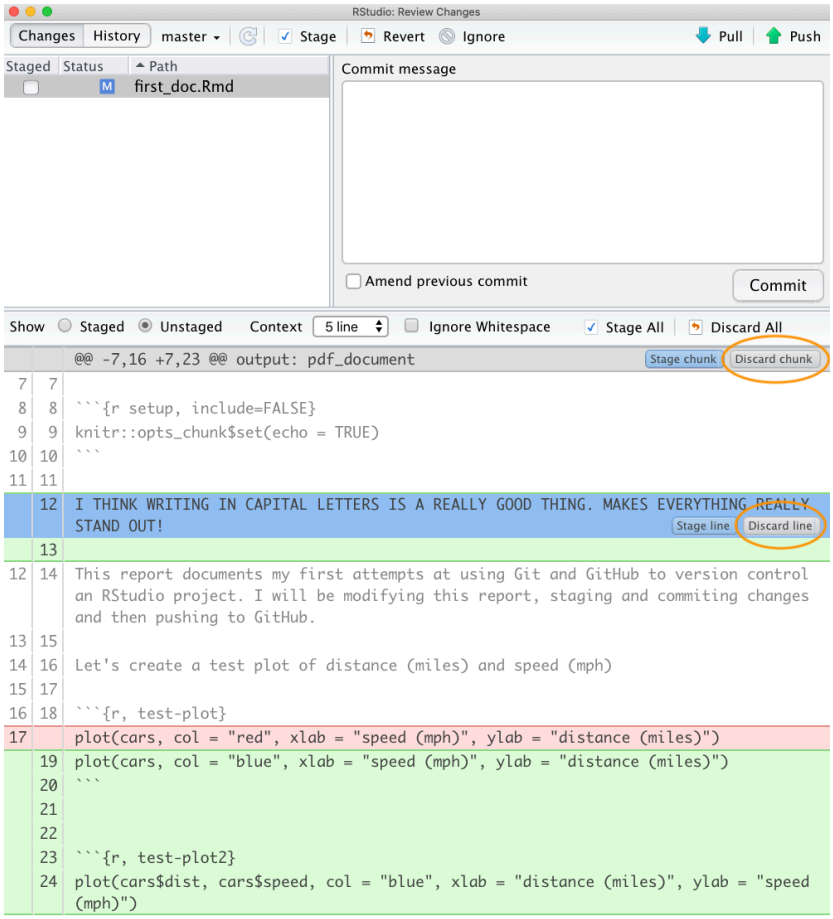
One the great things about using Git is that you are able to revert to previous versions of files if you've made a mistake, broke something or just prefer and earlier approach. How you do this will depend on whether the changes you want to discard have been staged, committed or pushed to GitHub. We'll 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.

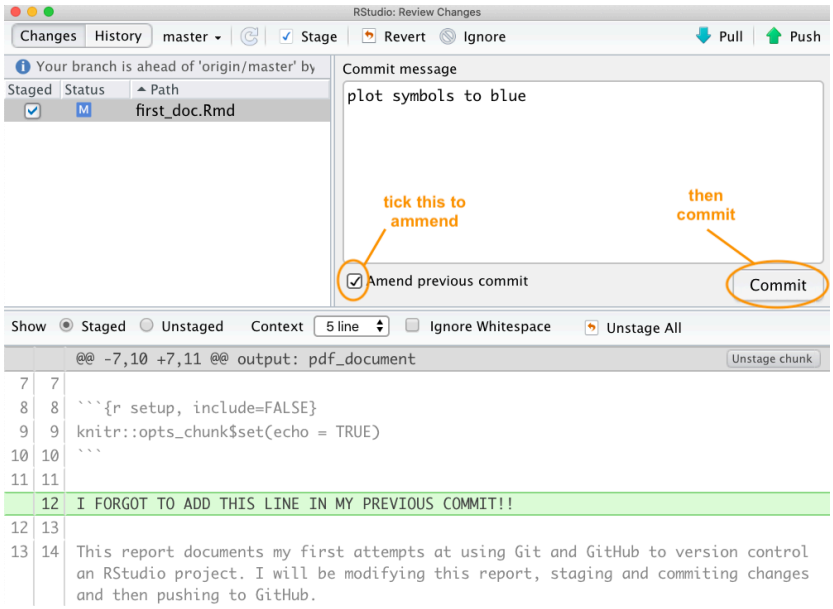


Staged but not committed and not pushed

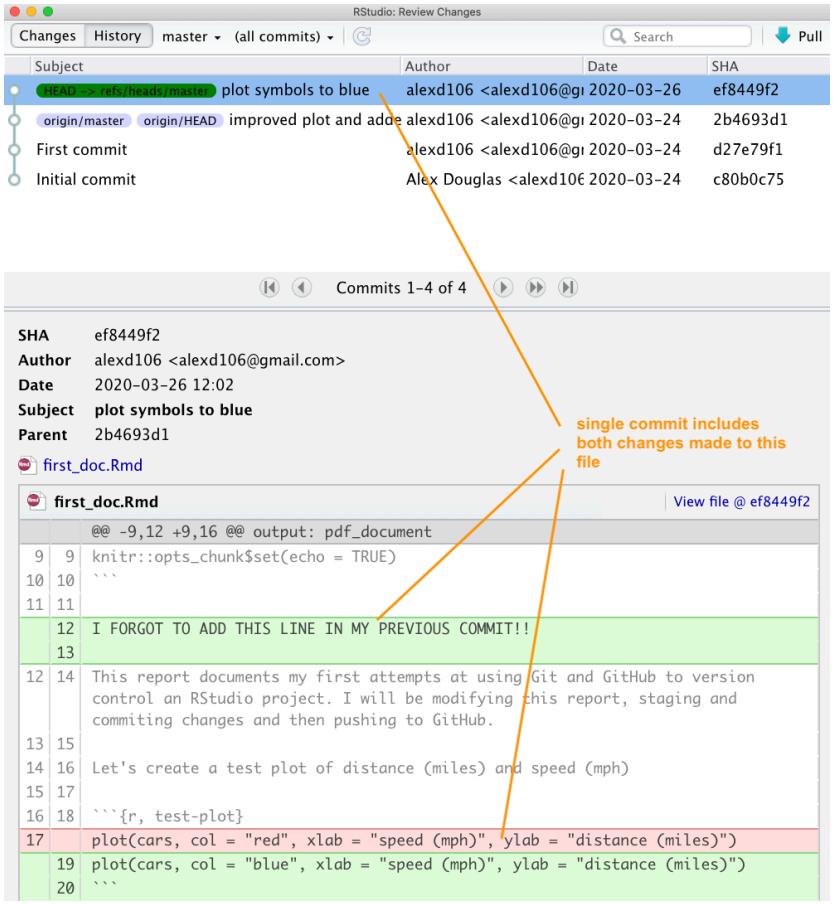
If you have staged your files, but not committed them then simply unstage them by clicking on the 'Staged' check box in the Git pane (or in the 'Review Changes' window) to remove the tick. You can then revert all or parts of the file as described in the section above.

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.

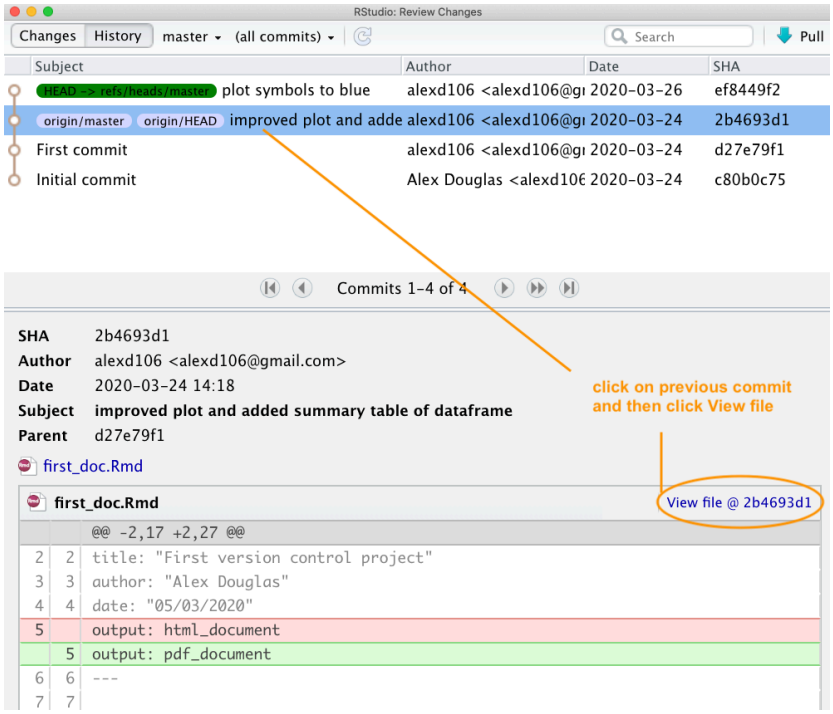


If we check out our commit history you can see that our latest commit contains both changes to the file rather than having two separate commits. We use the amend commit approach alot but it's important to understand that you should **not** do this if you have already pushed your last commit to GitHub as you are effectively rewriting history and all sorts bad things may happen!

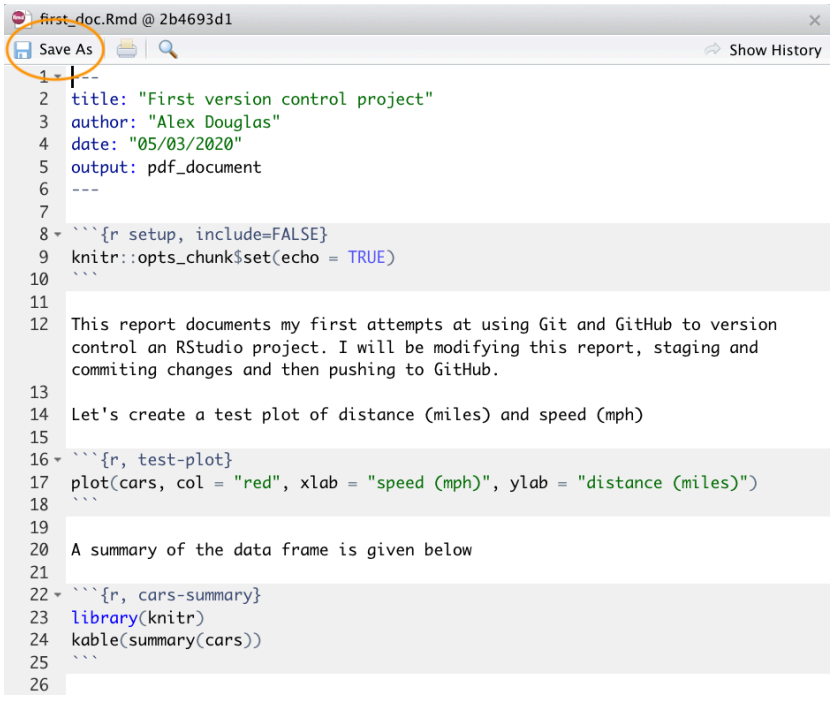


If you spot a mistake that has happened multiple commits back or you just want to revert to a previous version of a document you have a number of options.

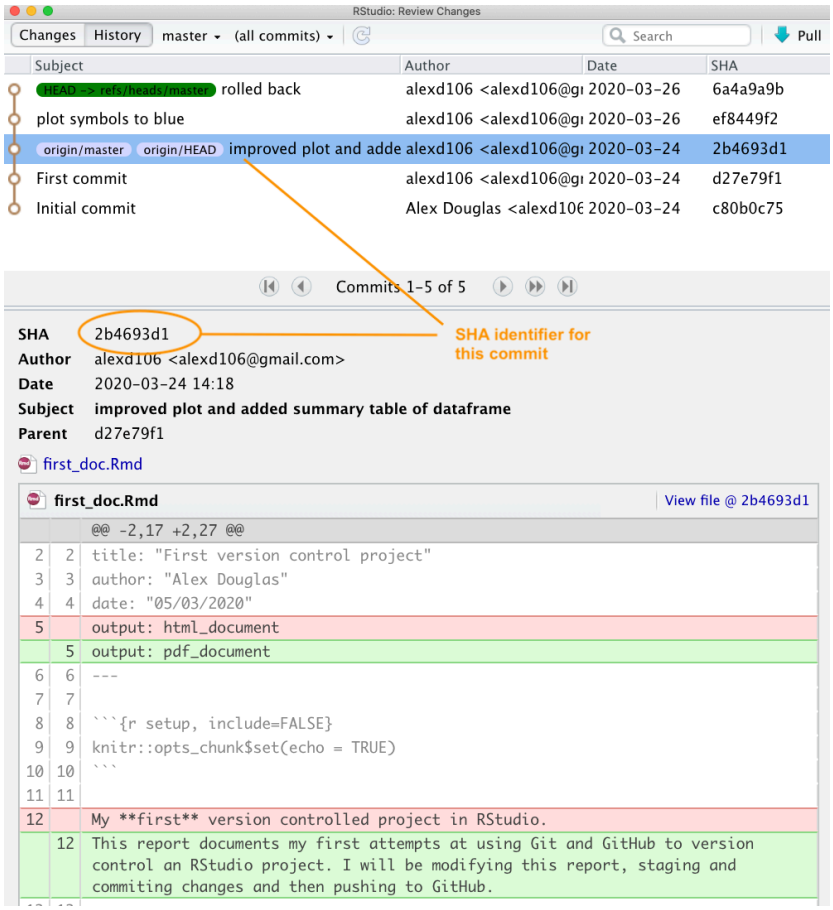
Option 1 - (probably the easiest but very unGit - but like, whatever!) is to look in your commit history in RStudio, find the commit that you would like to go back to and click on the 'View file @' button to show the file contents.



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!):

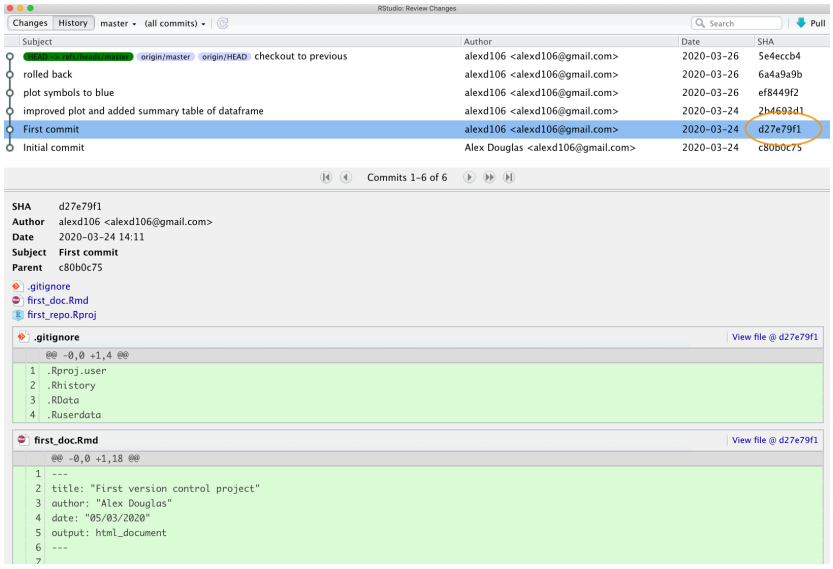
```
git rm -r .
git checkout 2b4693d1 .
```

Note that this will delete all files that you have created since you made this commit so

be careful!

Staged, committed and pushed

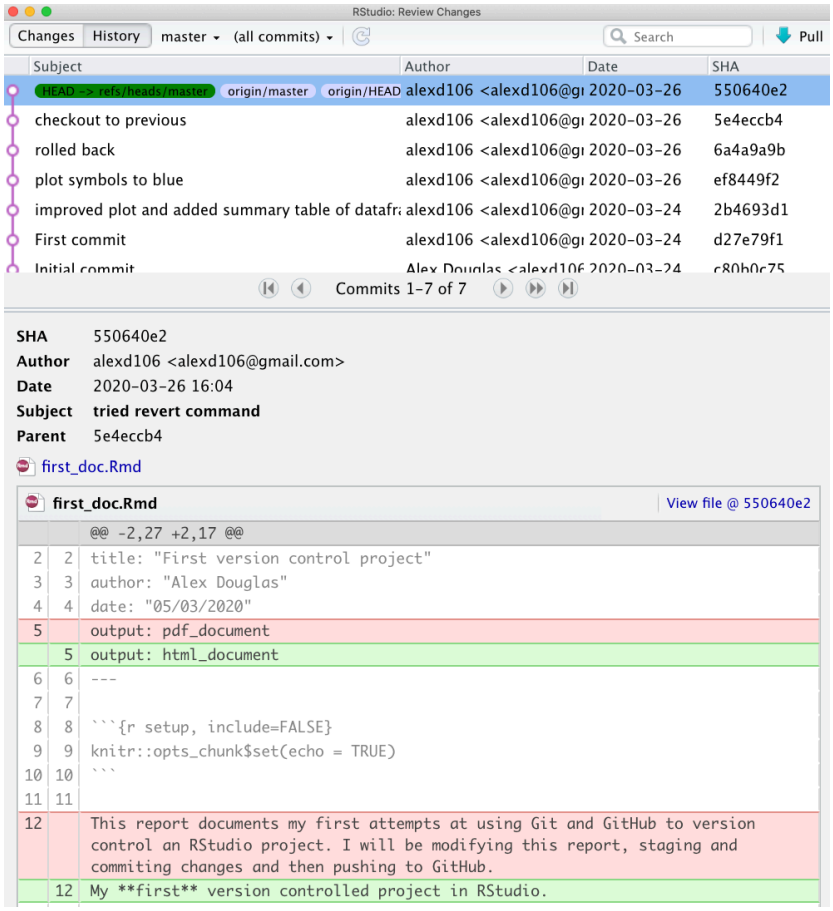
If you have already pushed your commits to GitHub you can use the `git checkout` strategy described above and then commit and push to update GitHub (although this is not really considered 'best' practice). Another approach would be to use `git revert` (Note: as far as we can tell `git revert` is not 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`.



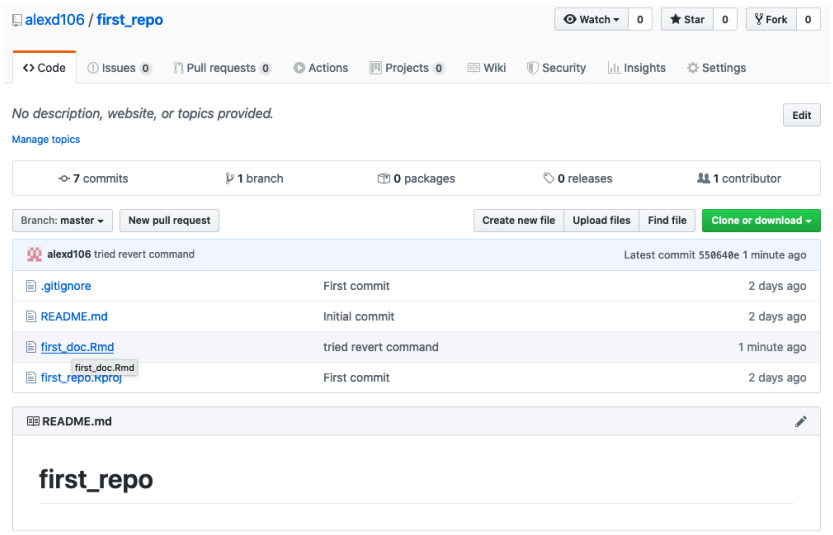
We can use the `revert` command as shown below in the Terminal. The `--no-commit` option is used to prevent us from having to deal with each intermediate commit.

```
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



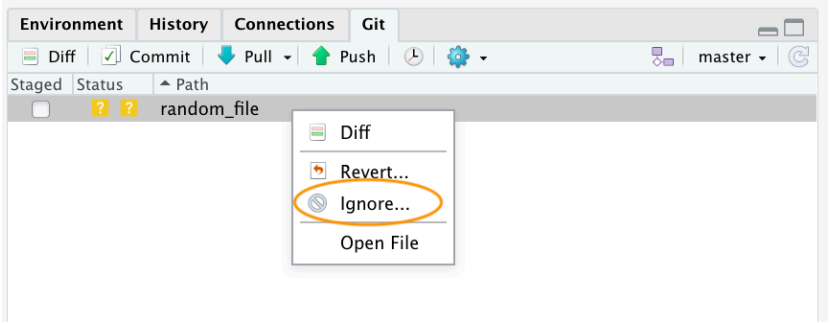
8.6.4 Collaborate with Git

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.

8.6.5 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.

8.7 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:

```
# 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:

```
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][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:

```
# 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:

```
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