Exploratory Data Analysis with R

Dylan Childs

2021-04-19

Contents

What you will learn

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

Aims

This book has three related aims:

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

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

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

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

Topics

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

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

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

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

Technologies

What is R?

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

- R is all about **data analysis**. We can carry out any standard statistical analysis in R, as well as access a huge array of more sophisticated tools with impressive names like "structural equation model", "random forests" and "penalized regression". These days, when statisticians and computer

scientists develop a new analysis tool, they often implement it in R first. This means a competent R user can always access the latest, cutting edge analysis tools. R also has the best graphics and plotting facilities of any platform. With sufficient expertise, we can make pretty much any type of figure we need (e.g. scatter plots, phylogenetic trees, spatial maps, or even volcanoes). In short, R is a very productive environment for doing data analysis.

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

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

What is RStudio?

R is essentially just a computer program that sits there and waits for instructions in the form of text. Those instructions can be typed in by a user or they can be sent to it from another program. R also runs in a variety of different environments. The job of RStudio is to provide an environment that makes R a more pleasant and productive tool.
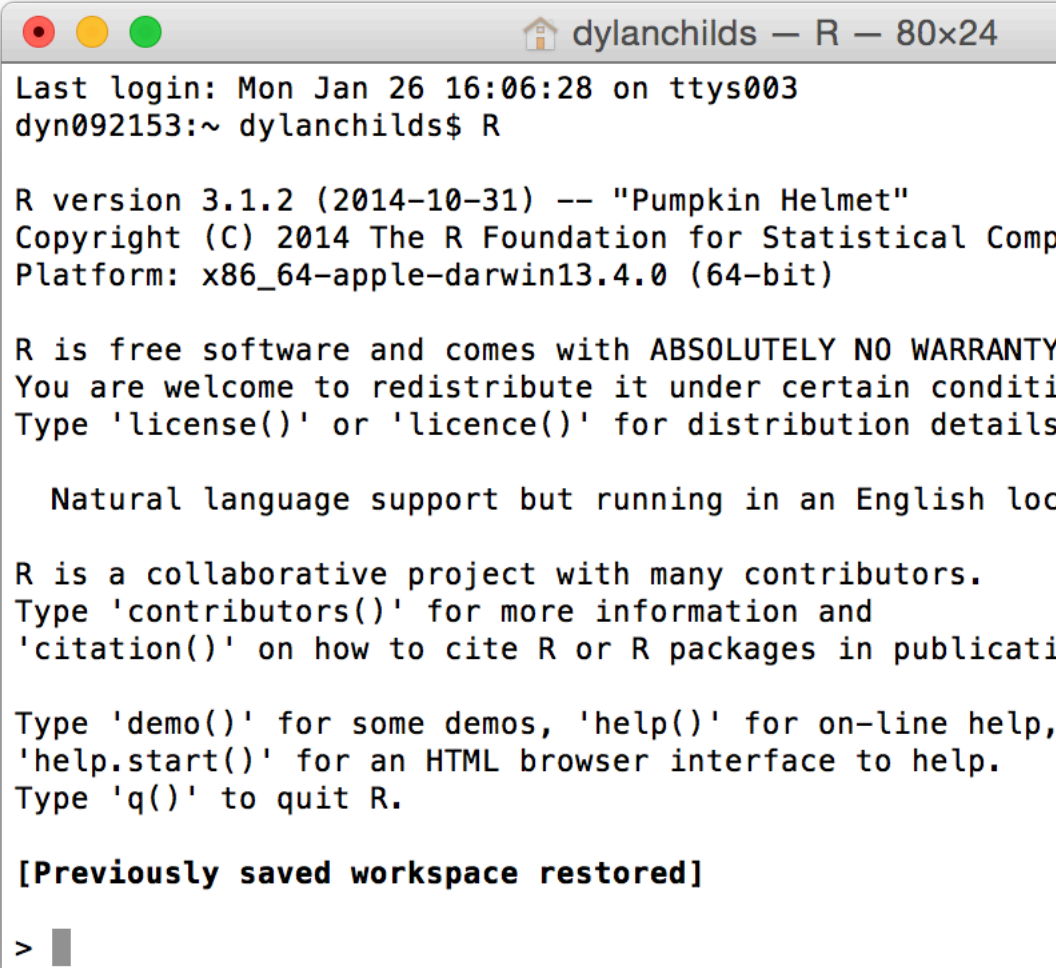
> ⚠ **R and RStudio are not the same thing.**
>
> RStudio is a different program from R—it is installed separately and occupies its own place in the Programs menu (Windows PC) or Applications folder (Mac). We can run R without RStudio if we need to, but we cannot run RStudio without R. Remember that!

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

---

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

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

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

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

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

- RStudio provides facilities for working with R programs using something called a Source Code Editor. An R program (also called a "script")" is just is a collection of instructions in the R language that have been saved to a text file. Nothing more! However, it is much easier to work with a script using a proper Source Code Editor than an ordinary text editor like Notepad.

- An good IDE like RStudio also gives you a visual, point-and-click means of accessing various language-specific features. This is a bit difficult to explain until we have have actually used some of these, but trust us, being able to do things like manage packages, set working directories, or inspect objects we've made simplifies day-to-day use of R. This especially true for new users.

- RStudio is cross-platform—it will run on a Windows PC, a Linux PC or a Mac. In terms of the appearance and the functionality it provides, RStudio is exactly the same on each of these platforms. If we learn to work with R via RStudio on a Windows PC, it's no problem migrating to a Mac or Linux PC later on if we need to. This is a big advantage for those of us who work on multiple platforms.

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

How to use this book

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

Text, instructions, and explanations

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

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

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

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

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

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

R code and output

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

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

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

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

Get up and running

Different ways to run RStudio

We can run RStudio in a variety of different ways—

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

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

Installing R and RStudio locally

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

Installing R

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

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

Installing RStudio

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

A quick look at RStudio

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

RStudio

Console ~/

R version 3.1.2 (2014-10-31) -- "Pumpkin Helmet"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)

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

  Natural language support but running in an English locale

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

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

[Workspace loaded from ~/.RData]

>

1

Environment

Global Enviro

Files   Plots

New Folder
Home
                        Nam
              .RData
              .Rhisto
              Applic
              backup
              Deskto
              Docum
              Downl
              Dropb
              Google
              Library
              Movies
              Music
              Picture
              Public
              Repos

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

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

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

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

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

Working at the Console in RStudio

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

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

```
1+3
```

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

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

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

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

> **What does that word 'expression' mean?**
>
> Why do we keep using that word *expression*? Here is what the Wikipedia page says:
>
>> An expression in a programming language is a combination of explicit values, constants, variables, operators, and functions that are interpreted according to the particular rules of precedence and of association for a particular programming language, which computes and then produces another value.
>
> That probably doesn't make much sense! In simple terms, an R expression is a small set of instructions that tell R to do something. That's it. We could write 'instructions' instead of 'expressions' throughout this book but we may as well use the correct word.

Part I

Introduction to R

Chapter 1

A quick introduction to R

1.1 Using R as a big calculator

1.1.1 Basic arithmetic

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

Addition–

```
3 + 2
```

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

Subtraction–

```
5 - 1
```

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

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

Multiplication–

```
7 * 2
```

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

Division–

```r
3 / 2
```

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

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

Exponentiation–

```r
4^2
```

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

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

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

1.1.2   Combining arithmetic operations

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

```r
2^3 - 6
```

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

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

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

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

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

If that's obvious, great. If not, it's time to learn about **order of precedence**. R uses a standard set of rules to decide the order in which calculations feed into

one another to unambiguously evaluate any expression. It uses the same order as every other computer language, which thankfully is the same one we all learn in mathematics classes at school. The order of precedence is:

1. exponents and roots (also, 'powers' or 'orders')

2. division and then multiplication

3. additional and then subtraction

> ℹ **BODMAS and friends**
>
> If you find it difficult to remember the standard order of precedence there are a load of mnemonics that can to help.

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

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

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

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

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

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

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

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

1.2   Problematic calculations

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

```
1 / 0
```

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

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

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

```
0 / 0
```

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

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

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

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

1.3   Storing and reusing results

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

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

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

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

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

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

Solution 1–

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

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

Solution 2–

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

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

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

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

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

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

```
a <- 1
```

```
b <- 6
```

```
c <- 5
```

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

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

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

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

```
a
```

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

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

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

Sum–

```
a + b + c
```

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

Product–

---

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

```
a * b * c
```

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

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

Solution 1–

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

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

Solution 2–

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

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

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

```
b <- 5
```

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

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

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

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

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

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

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

> ⚡ **RStudio shortcut**
>
> We use the assignment operator `<-` all the time when working with R. Because it's inefficient to type the `<` and `-` characters repeatedly, RStudio has a built-in shortcut for typing the assignment operator.
>
> The shortcut is 'Alt + -' . Try it now. Move the cursor to the Console, hold down the Alt key ('Option' on a Mac), and press the `-` sign key. RStudio will auto-magically add insert `<-`. **If you only learn one RStudio shortcut, learn this one!** It will save you a lot of time in the long run.

1.4   How does assignment work?

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

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

```
myvar <- 1
myvar
```

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

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

```
myvar <- 7
myvar
```

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

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

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

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

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

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

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

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

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

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

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

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

```r
myvar <- 10
```

```r
myvar
```

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

```r
mycopy
```

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

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

1.5   Global environment

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

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

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

1.6   Naming rules and conventions

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

- A legal name in R is any sequence of letters, numbers, `.`, or `_`, but the sequence of characters we use must begin with a letter. Both upper and lower case letters are allowed. For example, `num_1`, `num.1`, `num1`, `NUM1`, `myNum1` are all legal names, but `1num` and `_num1` are not because they begin with `1` and `_`.

- R is case sensitive—it treats upper and lower case letters as different characters. This means R treats `num` and `Num` as distinct names. Forgetting about case sensitivity is a good way to create errors when using R. Try to remember that.

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

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

Chapter 2

Using functions

2.1   Introduction

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

2.2   Functions and arguments

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

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

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

What about the bits inside the parentheses? These are called the **arguments** of the function. That's a horrible name, but it is the one that everyone uses, so

35

we have to get used to it. Depending on how it was defined, a function can take zero, one, or more arguments. We will discuss this idea in more detail later in this section. In simple terms, the arguments control the behaviour of a function.

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

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

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

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

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

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

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

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

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

---

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

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

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

2.3 Evaluating arguments and returning results

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

> **ⓘ Argument names vs variable names**
>
> Keeping in mind what we've just learned, take a careful look at this example:
>
> ```
> x <- 0
> round(x = 3.7, digits = x)
> ```
>
> ```
> ## [1] 4
> ```
>
> What is going on here? The key to understanding this is to realise that the symbol `x` is used in two different ways here. When it appears on the left-hand side of the `=` it represents an argument name. When it appears on the right-hand side, it is treated as a variable name, which must have a value associated with it for the above to be valid. That is a confusing way to use the `round` function, but it is perfectly valid.
>
> The message here is that what matters is where things appear relative to the `=`, not the symbols used to represent them.

2.4 Specifying function arguments

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

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

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

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

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

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

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

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

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

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

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

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

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

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

names:

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

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

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

> ⚠ **Be careful with your arguments**
>
> Here is some advice. Do not rely on partial matching of function names. It just leads to confusion and the odd error. If you use it a lot, you end up forgetting the true name of arguments, and if you abbreviate too much, you create name matching conflicts. For example, if a function has arguments `arg1` and `arg2` and you use the partial name `a` for an argument, there is no way to know which argument you meant. We are pointing out partial matching so that you are aware of the behaviour. It is not worth the hassle of getting it wrong to save on a little typing, so do not use it.
>
> What about position matching? This can also cause problems if we're not paying attention. For example, if you forget the order of the arguments to a function and then place your arguments in the wrong place, you will either generate an error or produce a nonsensical result. It is nice not to have to type out the `name = value` construct all the time though, so our advice is to rely on positional matching only for the first argument. This is a common convention in R that makes sense because it is often obvious what kind of information the first argument should carry.

2.5   Combining functions

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

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

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

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

Here is another way to replicate the same calculation:

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

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

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

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

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

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

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

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

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

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

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

2.6 Functions do not have 'side effects'

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

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

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

```
y
```

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

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

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

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

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

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

Chapter 3

Vectors

3.1 Introduction

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

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

3.2 Atomic vectors

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

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

---

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

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

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

```r
1 + 1
```

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

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

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

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

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

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

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

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

3.3 Numeric vectors

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

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

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

---

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

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

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

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

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

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

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

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

```
pi
```

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

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

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

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

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

---

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

3.4   Constructing numeric vectors

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

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

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

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

Now look at this example:

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

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

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

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

3.5   Named vectors

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

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

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

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

```
names(namedv)
```

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

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

3.6 Vectorised operations

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

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

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

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

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

This constructed two length-10 numeric vectors, called `x` and `y`, where `x` is a sequence from 1 to 10 and `y` is a sequence from 0.1 to 1.0. `x` and `y` are the same length. Now look at what happens when we add these using `+`:

```
x + y
```

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

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

Vectorisation is implemented in all the standard mathematical functions. For

example, the `round` function rounds each element of a numeric vector to the nearest integer by default:

```
round(y)
```

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

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

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

```
sum(y)
```

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

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

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

3.7   Other kinds of atomic vectors

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

3.7.1   Character vectors

Each element of a **character vectors** is what is known as a "character string" (or "string" if we are feeling lazy). That term "character string" refers to a sequence of characters, such as "Treatment 1", "University of Sheffield", "Pop-

ulation Density". A character vector is an atomic vector that stores an ordered collection of one or more character strings.

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

```r
"Dylan"
```

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

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

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

```r
Dylan
```

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

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

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

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

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

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

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

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

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

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

3.7.2   Logical vectors

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

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

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

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

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

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

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

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

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

```
y
```

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

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

```
x > y
```

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

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

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

```
x == y
```

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

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

⚠ **= and == are not the same**

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

Chapter 4

Data frames

4.1 Introduction

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

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

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

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

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

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

We're introducing these ideas now because we're going to consider a new type of data object in this chapter—the **data frame**. Real-world data analysis involves collections of related statistical variables. How should we keep a large collection

53

of variables organised? We could work with them individually, but this tends to be error prone. Instead, we need a way to keep related variables together. This is the problem that **data frames** are designed to manage.

4.2 Data frames

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

We'll think about the columns first.

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

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

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

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

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

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

```r
trt
```

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

```
bms
```

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

```
div
```

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3 Exploring data frames

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

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

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

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

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

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

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

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

it.

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

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

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

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

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

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

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

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

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

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

4.4   Extracting and adding a single variable

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
experim.data$Biomass
```

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

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

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

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

Chapter 5

Packages

5.1 The R package system

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

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

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

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

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

5.2   Task views

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

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

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

5.3   Using packages

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

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

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

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

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

5.3.1 Viewing installed packages

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

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

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

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

The `find.package` function either prints a "file path" showing us where the package is located, as above, or return an error if the package can't be found. Alternatively, a function called `installed.packages` will return a data frame containing a lot of information about the installed packages.

5.3.2 Installing packages

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

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

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

The quotes are necessary by the way. If everything is working—we have an active internet connection, the package name is valid, and so on—R will briefly pause while it communicates with the CRAN servers, we should see some red text reporting back what's happening, and then we're returned to the prompt. The red text is just letting us know what R is up to. As long as this text does not include the word "error", there is usually no need to worry about it.

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

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

5.3.3   Loading and attaching packages

Once we've installed a package or two, we'll probably want to use them. Two
things have to happen to access a package's facilities: the package has to be
loaded into memory, and then it has to attached to something called a search
path so that R can find it.

It is beyond the scope of this book to get in to "how" and "why" of these events.
Fortunately, there's no need to worry about these details, as both loading and
attaching can be done in a single step with a function called `library`. The
`library` function works as we might expect it to. If we want to start using the
`fortunes` package—which was just installed above—all we need is:

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

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

```
fortune()
```

```
##
## Friends don't let friends use Excel for statistics!
##     -- Jonathan D. Cryer (about problems with using Microsoft Excel for
##        statistics)
##        JSM 2001, Atlanta (August 2001)
```

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

> ⚠ **5.3.4 Don't use RStudio for loading packages!**
>
> As usual, if we don't like working in the Console RStudio can help us out. There is a small button next to each package listed in the **Packages** tab. Packages that have been loaded and attached have a blue check box next to them, whereas this is absent from those that have not. Clicking on an empty check box will load up the package. We mention this because at some point most people realise they can use RStudio to load and attach packages. **We don't recommend using this route**. It's much better to put `library` statements into your R script. Read the relevant appendix if you're not sure what a script is yet.

5.3.5 An analogy

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

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

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

5.4 Package data

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

vignettes. Use the `data` function to list the data sets hiding away in packages:

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

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

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

From the perspective of learning to use R, working with package data is really useful because it allows us to work with 'well-behaved' data sets without having to worry about getting them into R. Importing data is certainly an important skill but it's not necessarily something a new user wants to worry about. For this reason, we tend to use package data sets in this book.

5.5   The tidyverse ecosystem of packages

We're going to be using several packages that belong to a widely used, well-known ecosystem of packages known as the **tidyverse**. Here is the description of the tidyverse on its website:

> The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Using the tidyverse makes data analysis simpler, faster and a more entertaining experience (honestly).

Part II

Data Wrangling

Chapter 6

Getting ready to use dplyr

6.1 Introduction

Data wrangling is the process of cleaning and manipulating data to get it ready
for analysis, for example, by creating derived variables and subsets of the data.
Although not the most exciting part of a study—we want to answer questions,
not format data—data wrangling is critical. This step in a data analysis work-
flow can become very time-consuming if not tackled with the right tools.

The next few chapters will introduce the **dplyr** package. **dplyr** is an important
member of the tidyverse ecosystem. Its job is to provide a set of tools to address
common data manipulation tasks, such as selecting subsets of data, making new
variables, and summarising data in various ways.

However, before we can start to use these tools, we need a bit of a foundation.
That is the aim of this chapter. We will provide some background to the so-
called tidy data principles, introduce the data set we'll be using in our examples,
discuss the tidyverse version of data frames known as the tibble, and finally,
introduce the **dplyr** package.

6.2 Tidy data

dplyr will work with any data frame. However, it is most powerful when data
are organised according to tidy data conventions for rectangular data sets. Tidy
data has a specific structure that makes it easy to manipulate, model and visu-
alise. A tidy data set is one where each variable is only found in one column and
each row contains one unique observation (an imaged cell, a treated organism,
an experimental plot, and so on).

The basic principles of tidy data are not too difficult to understand. We'll
use an example to illustrate what the "one variable = one column" and "one

observation = one row" idea means. Let's return to the made-up experiment investigating the response of communities to fertilizer addition. This time, imagine we had only measured biomass, but that we had measured it at two time points throughout the experiment.

We'll look at two ways to organise some artificial data from this experimental setup. The first uses a separate column for each biomass measurement:

```
##      Treatment BiomassT1 BiomassT2
## 1    Control         284       324
## 2    Control         328       400
## 3    Control         291       355
## 4 Fertilser          956      1197
## 5 Fertilser          954      1012
## 6 Fertilser          685       859
```

This feels like a reasonable way to store such data, especially for an Excel user. However, this format is **not tidy**. Why? The biomass variable has been split across two columns (`BiomassT1` and `BiomassT2`), which means each row corresponds to two distinct observations. We won't go into the 'whys' here but take our word for it—adopting this format makes it difficult to use **dplyr** efficiently.

A tidy version of that example data set would still have three columns but now these would be: `Treatment`, denoting the experimental treatment applied; `Time`, denoting the sampling occasion; and `Biomass`, denoting the biomass measured:

```
##      Treatment Time Biomass
## 1     Control   T1     284
## 2     Control   T1     328
## 3     Control   T1     291
## 4   Fertilser   T1     956
## 5   Fertilser   T1     954
## 6   Fertilser   T1     685
## 7     Control   T2     324
## 8     Control   T2     400
## 9     Control   T2     355
## 10  Fertilser   T2    1197
## 11  Fertilser   T2    1012
## 12  Fertilser   T2     859
```

The change we made was to create an indicator variable called `Time` for the sampling occasion. In version one of the data, the time information was implicit—the time associated with a biomass measurement was encoded by column membership (`BiomassT1` vs `BiomassT2`). In the second version of the data set an indicator variable, `Time`, was created to label the sampling occasion explicitly. That simple change means each row corresponds to one distinct observation and each variable lives in one column. These data are now tidy and ideally set up for use with **dplyr**.

> ⚠ **Always try to start with tidy data**
>
> The best way to make sure your data set is tidy is to store in that format **when it's first collected and recorded**. Some packages can help convert non-tidy data into the tidy data format (e.g. the **tidyr** package), but life is much simpler if we ensure our data are tidy from the very beginning.

6.3 Penguins! + =

To make progress with **dplyr**, we are going to need some data to play around with. The data we'll use contains measurements taken from penguins on the Palmer Archipelago, off the north-western coast of the Antarctic Peninsula. Each row contains information about an individual penguin, including:

- the species it belongs to,
- the island it was found on,
- morphometric data (flipper length, body mass, bill dimensions)
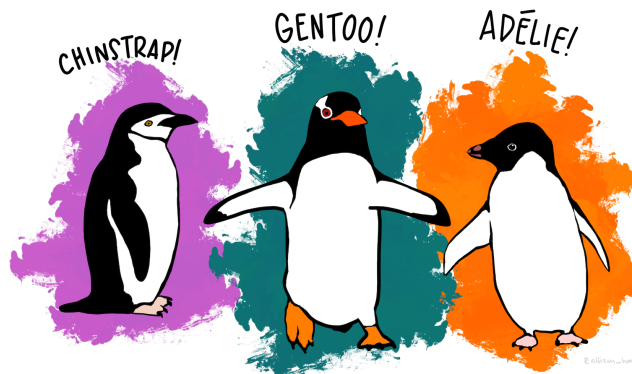- its body mass and sex,
- and finally, the year of capture.

Figure 6.1: Meet the Palmer penguins (artwork by Allison Horst)

Why use this data set? Apart from the fact that everyone likes penguins, obviously, these data are sufficiently complex to demonstrate everything we want to do, while remaining easily understandable. Here is the full data set shown as a table:

N.B. — the data will only show up in the HTML version of the book.

This shows the first ten rows along with the first few columns. The *Previous / Next* links at the bottom right can be used to navigate the rows; the arrows in

the top header row are used to view the different columns. Below each column name you can also see three-letter abbreviations like `<dbl>`, `<int>` and `<chr>`. These are telling us that each column is a vector of some kind:

- `<dbl>` = a numeric 'double' vector (real numbers, i.e. with decimals)

- `<int>` = a numeric integer vector (integer numbers)

- `<chr>` = character vector.

So… data frames are table-like objects with rows and columns. They can also be seen in even simpler terms—data frames are simply collections of vectors, where each one represents a different aspect of a multi-faceted data set.

We're going to be seeing a lot of this data set in the remaining chapters so we won't say anything more about it now.

> **Where can we get the penguins data?**
>
> The Palmer penguins data were collected and made available by Dr. Kristen Gorman and the Palmer Station. The data set exists mostly to support people learning and teaching R. It isn't part of base R, though. We have to import it into R somehow. There are two options:
>
> 1. The data are available in an R package called… palmerpenguins. Like other packages, **palmerpenguins** can be installed from CRAN using either `install.packages` or the usual RStudio point-and-click mechanism. Once installed, we can make it available by running `library(palmerpenguins)` in any R session. After that, just type the name of the data set and R will find it and use it. The version in **palmerpenguins** is called `penguins`, by the way.
>
> 2. We could get a copy of the data set and store it as a file on our hard drive, ideally using a standard data format. The most common and portable format for storing rectangular data is as a 'Comma Separated Value' (CSV) text file. Once you have a copy of the data as a CSV file, it's just a matter of using a function like read_csv from the readr package to import the data into R. This is explained in the Managing projects, scripts and data files appendix.

6.4 Um… tibbles?

To increase the scope of what the tidyverse can do, its makers created a special kind of data object known as a 'tibble'. This is meant to sound like the word 'table' pronounced by a New Zealander. We're not lying—the person who started the tidyverse is from New Zealand. Its name is a clue that a tibble is a table-like object, i.e. a rectangular data structure similar to a data frame.

In fact, the easiest way to conceptualise a tibble is as a special data frame with a few extra whistles and bells. More often than not, it's not necessary to pay attention to whether we're working with an ordinary data frame or a tidyverse tibble—we can often treat them as though they are interchangeable.

That said, there are exceptions to this rule of thumb, and we do occasionally need to work out which one we're using. A simple way to do this is by printing the data object. Imagine that we've imported the Palmer penguins into R and stored it in a tibble called `penguins`. This is what that would look like if we printed it at the Console:

```
penguins
```

```
## # A tibble: 344 x 8
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <chr>   <chr>              <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen           39.1          18.7               181        3750
##  2 Adelie  Torgersen           39.5          17.4               186        3800
##  3 Adelie  Torgersen           40.3          18                 195        3250
##  4 Adelie  Torgersen           NA            NA                  NA          NA
##  5 Adelie  Torgersen           36.7          19.3               193        3450
##  6 Adelie  Torgersen           39.3          20.6               190        3650
##  7 Adelie  Torgersen           38.9          17.8               181        3625
##  8 Adelie  Torgersen           39.2          19.6               195        4675
##  9 Adelie  Torgersen           34.1          18.1               193        3475
## 10 Adelie  Torgersen           42            20.2               190        4250
## # ... with 334 more rows, and 2 more variables: sex <chr>, year <int>
```

The formatting applied to the output is telling us `penguins` is a tibble rather than an ordinary data frame (we only called it a data frame earlier because we had not introduced the concept of a tibble yet). There are a couple of clues about this in that printout. The very obvious one is the first line: `# A tibble: 344 x 8`. The printed output states `penguins` is a tibble! The output is also truncated—only the first ten lines were printed, and any columns that won't fit on one row are summarised at the bottom (i.e. `sex` and `year`).

6.5 Missing values

Take a close look at the values of the body mass data (`body_mass_g`) in `penguins` (we're using `$` to extract the whole column):

```
penguins$body_mass_g
```

```
##   [1] 3750 3800 3250   NA 3450 3650 3625 4675 3475 4250 3300 3700 3200 3800 4400
##  [16] 3700 3450 4500 3325 4200 3400 3600 3800 3950 3800 3800 3550 3200 3150 3950
##  [31] 3250 3900 3300 3900 3325 4150 3950 3550 3300 4650 3150 3900 3100 4400 3000
```

```
##   [46] 4600 3425 2975 3450 4150 3500 4300 3450 4050 2900 3700 3550 3800 2850 3750
##   [61] 3150 4400 3600 4050 2850 3950 3350 4100 3050 4450 3600 3900 3550 4150 3700
##   [76] 4250 3700 3900 3550 4000 3200 4700 3800 4200 3350 3550 3800 3500 3950 3600
##   [91] 3550 4300 3400 4450 3300 4300 3700 4350 2900 4100 3725 4725 3075 4250 2925
##  [106] 3550 3750 3900 3175 4775 3825 4600 3200 4275 3900 4075 2900 3775 3350 3325
##  [121] 3150 3500 3450 3875 3050 4000 3275 4300 3050 4000 3325 3500 3500 4475 3425
##  [136] 3900 3175 3975 3400 4250 3400 3475 3050 3725 3000 3650 4250 3475 3450 3750
##  [151] 3700 4000 4500 5700 4450 5700 5400 4550 4800 5200 4400 5150 4650 5550 4650
##  [166] 5850 4200 5850 4150 6300 4800 5350 5700 5000 4400 5050 5000 5100 4100 5650
##  [181] 4600 5550 5250 4700 5050 6050 5150 5400 4950 5250 4350 5350 3950 5700 4300
##  [196] 4750 5550 4900 4200 5400 5100 5300 4850 5300 4400 5000 4900 5050 4300 5000
##  [211] 4450 5550 4200 5300 4400 5650 4700 5700 4650 5800 4700 5550 4750 5000 5100
##  [226] 5200 4700 5800 4600 6000 4750 5950 4625 5450 4725 5350 4750 5600 4600 5300
##  [241] 4875 5550 4950 5400 4750 5650 4850 5200 4925 4875 4625 5250 4850 5600 4975
##  [256] 5500 4725 5500 4700 5500 4575 5500 5000 5950 4650 5500 4375 5850 4875 6000
##  [271] 4925   NA 4850 5750 5200 5400 3500 3900 3650 3525 3725 3950 3250 3750 4150
##  [286] 3700 3800 3775 3700 4050 3575 4050 3300 3700 3450 4400 3600 3400 2900 3800
##  [301] 3300 4150 3400 3800 3700 4550 3200 4300 3350 4100 3600 3900 3850 4800 2700
##  [316] 4500 3950 3650 3550 3500 3675 4450 3400 4300 3250 3675 3325 3950 3600 4050
##  [331] 3350 3450 3250 4050 3800 3525 3950 3650 3650 4000 3400 3775 4100 3775
```

The body mass information lives in a numeric (integer) vector, but not every element in that vector is a number. Two values are `NA`. That stands for 'Not Available'—the `NA`'s job is to label cases where a value is missing or unknown. If you scroll around the table view of `penguins` above you'll find `NA`'s in several of the columns.

Missing data crop up all the time. They are just one of those facts of life—maybe the recording machine broke one afternoon, perhaps one of our organisms was infected by a pathogen, or maybe a cow ate one of our experimental plots. One of the nice things about R is that it knows how to represent missing data. It does this using the `NA` symbol.

You need to be aware of missing values when they are present. Why? Because the behaviour of many functions is affected by the presence of `NA`s. Things get confusing if we don't understand that behaviour. We will see some examples of this in the next few chapters.

6.6   Introducing dplyr

The **dplyr** package has been carefully designed to make it easy to manipulate 'rectangular data', such as data frames. **dplyr** is very consistent in the way its functions work. For example, the first argument of the main **dplyr** functions is always an object containing our data. This consistency makes it very easy to get to grips with each of the main **dplyr** functions—it's often possible to understand how one works by seeing one or two examples of its use.

Another reason for using **dplyr** is that it is orientated around a few core functions, each designed to do one thing well. These **dplyr** functions are sometimes referred to as its 'verbs' to reflect the fact that they 'do something' to data. For example:

- `select` is obtains a subset of variables,
- `mutate` is constructs new variables,
- `filter` is obtains a subset of rows,
- `arrange` is reorders rows, and
- `summarise` is calculates information about groups.

Notice that the names are chosen to reflect what each verb/function does to the input data. We'll cover each of these verbs in detail in later chapters, as well as a few additional ones, such as `rename` and `group_by`.

Apart from being easy to use, **dplyr** is also fast. This doesn't matter for small data sets but can be important when working with data sets with hundreds of thousands of rows. The **dplyr** package also allows us to work with data stored in different ways, for example, by interacting directly with several database systems. We're going to focus on using it with data frames and tibbles. But remember—once you know how to use **dplyr** with data frames its easy to use it for work with other kinds of data sources.

> **A dplyr cheat sheet**
>
> The developers of RStudio have produced a very usable cheat sheat that summarises the main data wrangling tools provided by **dplyr**.

Our advice is to download this, print out a copy, and refer to it often as you start working with **dplyr**.

6.6.1 A first look at dplyr

Let's take a preliminary look at the **dplyr** package. The package is not part of the base R installation, so we have to have installed it first. Remember, once installed, there is no need to install the package every time we need to use it. We do have to use `library` to load and attach the package every time we want to use it:

```
library("dplyr")
```

Let's look at one handy **dplyr** function now. Sometimes we just need a quick, compact summary of a data frame or tibble. This is the job of the `glimpse` function from **dplyr**:

```
glimpse(penguins)
```

```
## Rows: 344
## Columns: 8
## $ species            <chr> "Adelie", "Adelie", "Adelie", "Adelie", "Adelie", "A~
```

```
## $ island           <chr> "Torgersen", "Torgersen", "Torgersen", "Torgersen", ~
## $ bill_length_mm    <dbl> 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, 34.1, ~
## $ bill_depth_mm     <dbl> 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, 18.1, ~
## $ flipper_length_mm <int> 181, 186, 195, NA, 193, 190, 181, 195, 193, 190, 186~
## $ body_mass_g       <int> 3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, ~
## $ sex               <chr> "male", "female", "female", NA, "female", "male", "f~
## $ year              <int> 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007~
```

The function takes one argument: the name of a data frame or tibble. It then
tells us how many rows it has, how many columns there are, what these columns
are called, and what type of data is associated with them. This function is useful
when we need a quick overview of what's inside our data set. Some advice—use
this function on any new data set before trying to do anything with it.

6.6.2   dplyr pseudocode

We'll wrap-up this chapter with a quick mention of 'pseudocode'. Pseudocode
uses structural conventions of normal R code but is intended for us (humans)
to read rather than the computer. We use it summarise how **dplyr** functions
work. Here's an example you will encounter in the next chapter:

```
select(<data>, <variable-1>, <variable-2>, ...)
```

This is not an example we can run. It is pseudocode that serves as a
template for using the **dplyr select** function. The words surrounded by 'angle
brackets' (`< >`) are placeholders. If we want to turn this pseudocode into real
R code we have to **replace those placeholders**. For example, `<data>` is a
placeholder for the name of a data set. We would replace that with the word
`penguins` if we want to use select on our example data set (we'd also have to do
something with the other placeholders (e.g. `<variable-1>`)—we'll get to that
later).

That's enough about pseudocode and **dplyr** templates. We'll see plenty of these
in the next few chapters.

Chapter 7

Working with variables

7.1 Introduction

This chapter will explore the **dplyr** `select` and `mutate` verbs, and the closely related `rename` and `transmute` verbs. We consider these functions together because they operate on the variables (i.e. the columns) of a data frame or tibble:

- The `select` function selects a subset of variables to retain and (optionally) renames them in the process.
- The `mutate` function creates new variables from pre-existing ones and retains the original variables.
- The `rename` function renames one or more variables while keeping the remaining variable names unchanged.
- The `transmute` function creates new variables from pre-existing ones and drops the original variables.

7.1.1 Getting ready

Obviously, we need to have first installed **dplyr** package to use it. Assuming that's been done, we need to load and attach the package in the current session:

```
library("dplyr")
```

We will use the Palmer penguins data to illustrate the ideas in this chapter. Remember—the previous chapter described this data set and explained where to find it. The examples below assume it was read into R as a tibble with the name `penguins`.

7.2   Subset variables with `select`

We use `select` to **select variables** from a data frame or tibble. This is used when we have a data set with many variables but only need to work with a subset of these. Basic usage of `select` looks like this:

```
select(<data>, <variable-1>, <variable-2>, ...)
```

Remember—this is not an example we can run. This is a pseudocode designed to provide a generic description of how we use `select`. Let's look at the arguments of select:

- The first argument, `<data>`, must be the name of the object containing our data (usually a data frame or tibble). This is not optional—**dplyr** functions only exist to manipulate data.
- We then include a series of one or more additional arguments, where each one is the name of a variable in `<data>`. We've expressed this as `<variable-1>, <variable-2>, ...,` where `<variable-1>` and `<variable-2>` are names of the first two variables. The `...` is acting as a placeholder for the remaining variables. There could be any number of these.

It's easiest to understand how a function like `select` works by seeing it in action. We select the `species`, `bill_length_mm` and `bill_depth_mm` variables from **penguins** like this:

```
select(penguins, species, bill_length_mm, bill_depth_mm)
```

```
## # A tibble: 344 x 3
##    species bill_length_mm bill_depth_mm
##    <chr>            <dbl>         <dbl>
##  1 Adelie            39.1          18.7
##  2 Adelie            39.5          17.4
##  3 Adelie            40.3          18
##  4 Adelie            NA            NA
##  5 Adelie            36.7          19.3
##  6 Adelie            39.3          20.6
##  7 Adelie            38.9          17.8
##  8 Adelie            39.2          19.6
##  9 Adelie            34.1          18.1
## 10 Adelie            42            20.2
## # ... with 334 more rows
```

Hopefully, nothing about this example is too surprising. However, there are a few subtleties buried in that example:

- The `select` function is designed to work in a non-standard way which means variable names should **not** be surrounded by quotes. The one

exception is when a name has a space in it. Under those circumstances, it has to be quoted with backticks, e.g. `` `variable 1` ``.
- The `select` function does not have 'side effects'. This means is that it does not change the original `penguins` object. We printed the result produced by `select` to the Console, so we can't access the modified data set. If we need to use the result, we have to assign it a name using `<-`.
- The order of variables (i.e. the column order) in the resulting object is the same as the order in which they were supplied to the argument list. This means we can reorder variables at the same time as selecting them if we need to.
- The `select` function will return the same kind of data object we give it to work on. It returns a data frame if our data was in a data frame and a tibble if it was a tibble. In this example, R prints a tibble because `penguins` was a tibble.

That second point is important—we have to remember to assign the result a name using `<-` if we want to keep it and use it later. For example, we might call the result of that last example `penguins_bill`:

```
penguins_bill <- select(penguins, species, bill_length_mm, bill_depth_mm)
```

Now that we've named the new data set created by `select` we can refer to it by that name whenever we need it:

```
penguins_bill
```

```
## # A tibble: 344 x 3
##    species bill_length_mm bill_depth_mm
##    <chr>            <dbl>         <dbl>
##  1 Adelie            39.1          18.7
##  2 Adelie            39.5          17.4
##  3 Adelie            40.3          18
##  4 Adelie            NA            NA
##  5 Adelie            36.7          19.3
##  6 Adelie            39.3          20.6
##  7 Adelie            38.9          17.8
##  8 Adelie            39.2          19.6
##  9 Adelie            34.1          18.1
## 10 Adelie            42            20.2
## # ... with 334 more rows
```

Remember—the original `penguins` data is completely unchanged:

```
penguins
```

```
## # A tibble: 344 x 8
```

```
##    species island   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <chr>   <chr>             <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen          39.1          18.7               181        3750
##  2 Adelie  Torgersen          39.5          17.4               186        3800
##  3 Adelie  Torgersen          40.3          18                 195        3250
##  4 Adelie  Torgersen          NA            NA                 NA          NA
##  5 Adelie  Torgersen          36.7          19.3               193        3450
##  6 Adelie  Torgersen          39.3          20.6               190        3650
##  7 Adelie  Torgersen          38.9          17.8               181        3625
##  8 Adelie  Torgersen          39.2          19.6               195        4675
##  9 Adelie  Torgersen          34.1          18.1               193        3475
## 10 Adelie  Torgersen          42            20.2               190        4250
## # ... with 334 more rows, and 2 more variables: sex <chr>, year <int>
```

7.2.1 Alternative ways to identify variables with `select`

It's sometimes more convenient to use `select` to subset variables by specifying
those we do **not** need, rather than specifying of the ones to keep. We can use the
`!` operator to indicate that certain variables should be dropped. For example,
to get rid of the `bill_depth_mm` and `bill_length_mm` columns, we could use:

```
select(penguins, !bill_depth_mm, !bill_length_mm)
```

```
## # A tibble: 344 x 8
##    species island   bill_length_mm flipper_length_mm body_mass_g sex     year
##    <chr>   <chr>             <dbl>             <int>       <int> <chr>  <int>
##  1 Adelie  Torgersen          39.1               181        3750 male    2007
##  2 Adelie  Torgersen          39.5               186        3800 female  2007
##  3 Adelie  Torgersen          40.3               195        3250 female  2007
##  4 Adelie  Torgersen          NA                 NA          NA <NA>     2007
##  5 Adelie  Torgersen          36.7               193        3450 female  2007
##  6 Adelie  Torgersen          39.3               190        3650 male    2007
##  7 Adelie  Torgersen          38.9               181        3625 female  2007
##  8 Adelie  Torgersen          39.2               195        4675 male    2007
##  9 Adelie  Torgersen          34.1               193        3475 <NA>     2007
## 10 Adelie  Torgersen          42                 190        4250 <NA>     2007
## # ... with 334 more rows, and 1 more variable: bill_depth_mm <dbl>
```

This returns a tibble with all the other variables: `species`, `island`,
`flipper_length_mm`, `body_mass_g`, `sex` and `year`.

The `select` function can also be used to grab (or drop) a set of variables that
occur in a sequence next to one another. We specify a series of adjacent variables
using the `:` operator. We use this with two variable names, one on the left-hand
side and one on the right. When we use `:` like this, `select` will subset both
those two variables along with any others that fall in between them.

For example, if we want only the morphometric variables (`bill_length_mm`, `bill_depth_mm`, `flipper_length_mm` and `body_mass_g`) we could use:

```
select(penguins, bill_length_mm:body_mass_g)
```

```
## # A tibble: 344 x 4
##    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##             <dbl>         <dbl>             <int>       <int>
##  1           39.1          18.7               181        3750
##  2           39.5          17.4               186        3800
##  3           40.3          18                 195        3250
##  4           NA            NA                  NA          NA
##  5           36.7          19.3               193        3450
##  6           39.3          20.6               190        3650
##  7           38.9          17.8               181        3625
##  8           39.2          19.6               195        4675
##  9           34.1          18.1               193        3475
## 10           42            20.2               190        4250
## # ... with 334 more rows
```

The `:` operator can also be combined with `!` if we need to drop a series of variables according to their position in a data frame or tibble. For example, we can use this trick to get the complement of the previous example, i.e. throw away the morphometric variables:

```
select(penguins, !bill_length_mm:body_mass_g)
```

```
## # A tibble: 344 x 4
##    species island    sex    year
##    <chr>   <chr>     <chr> <int>
##  1 Adelie  Torgersen male   2007
##  2 Adelie  Torgersen female 2007
##  3 Adelie  Torgersen female 2007
##  4 Adelie  Torgersen <NA>   2007
##  5 Adelie  Torgersen female 2007
##  6 Adelie  Torgersen male   2007
##  7 Adelie  Torgersen female 2007
##  8 Adelie  Torgersen male   2007
##  9 Adelie  Torgersen <NA>   2007
## 10 Adelie  Torgersen <NA>   2007
## # ... with 334 more rows
```

7.2.2 Renaming variables with `select` and `rename`

The `select` function can also rename variables at the same time as selecting them. To do this, we name the arguments using the `name = value` construct,

where the name of the selected variable is placed on the right-hand side (`value`), and the new name goes on the left-hand side (`name`).

For example, to select the`species`, `bill_length_mm` and `bill_depth_mm` variables from `penguins`, and in the process, rename `bill_length_mm` and `bill_depth_mm` to `BillLength` and `BillDepth`, use:

```
select(penguins, species, BillLength = bill_length_mm, BillDepth = bill_depth_mm)
```

```
## # A tibble: 344 x 3
##    species BillLength BillDepth
##    <chr>        <dbl>     <dbl>
##  1 Adelie        39.1      18.7
##  2 Adelie        39.5      17.4
##  3 Adelie        40.3      18
##  4 Adelie        NA        NA
##  5 Adelie        36.7      19.3
##  6 Adelie        39.3      20.6
##  7 Adelie        38.9      17.8
##  8 Adelie        39.2      19.6
##  9 Adelie        34.1      18.1
## 10 Adelie        42        20.2
## # ... with 334 more rows
```

Renaming the variables is a common task. What should we do if the only thing we want to achieve is to rename variables, rather than rename *and* select them? **dplyr** provides an additional function called `rename` for exactly this purpose. This function renames some variables while retaining all others. It works like `select`. For example, to rename `bill_length_mm` and `bill_depth_mm` to `BillLength` and `BillDepth` but keep all the variables, use:

```
rename(penguins, BillLength = bill_length_mm, BillDepth = bill_depth_mm)
```

```
## # A tibble: 344 x 8
##    species island  BillLength BillDepth flipper_length_~ body_mass_g sex    year
##    <chr>   <chr>        <dbl>     <dbl>            <int>       <int> <chr> <int>
##  1 Adelie  Torger~       39.1      18.7              181        3750 male   2007
##  2 Adelie  Torger~       39.5      17.4              186        3800 fema~  2007
##  3 Adelie  Torger~       40.3      18                195        3250 fema~  2007
##  4 Adelie  Torger~       NA        NA                 NA          NA <NA>   2007
##  5 Adelie  Torger~       36.7      19.3              193        3450 fema~  2007
##  6 Adelie  Torger~       39.3      20.6              190        3650 male   2007
##  7 Adelie  Torger~       38.9      17.8              181        3625 fema~  2007
##  8 Adelie  Torger~       39.2      19.6              195        4675 male   2007
##  9 Adelie  Torger~       34.1      18.1              193        3475 <NA>   2007
## 10 Adelie  Torger~       42        20.2              190        4250 <NA>   2007
```

```
## # ... with 334 more rows
```

Notice `rename` also preserves the order of the variables found in the original data.

7.3 Creating variables with `mutate`

We use `mutate` to **add new variables** to a data frame or tibble. This is useful if we need to construct one or more derived variables to support an analysis. Basic usage of `mutate` looks like this:

```
mutate(<data>, <expression-1>, <expression-2>, ...)
```

Again, this is not an example we can run—it's pseudocode highlighting how to use `mutate` in abstract terms.

The first argument, `<data>`, must be the name of the object containing our data. We then include a series of one or more additional arguments, where each of these is a valid R expression involving one or more variables in `<data>`. We've have expressed these as `<expression-1>`, `<expression-2>`, where `<expression-1>` and `<expression-2>` represent the first two expressions, and the `...` is acting as a placeholder for the remaining expressions. These can be any valid R code that refers to variables in `<data>`. This is often a simple calculation (e.g. involving arithmetic), but it can be arbitrarily complex.

To see `mutate` in action, let's construct a new version of `penguins` that contains one extra variable—body mass measured in kilograms:

```
mutate(penguins, body_mass_g / 1000)
```

```
## # A tibble: 344 x 9
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <chr>   <chr>              <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen           39.1          18.7               181        3750
##  2 Adelie  Torgersen           39.5          17.4               186        3800
##  3 Adelie  Torgersen           40.3          18                 195        3250
##  4 Adelie  Torgersen           NA            NA                  NA          NA
##  5 Adelie  Torgersen           36.7          19.3               193        3450
##  6 Adelie  Torgersen           39.3          20.6               190        3650
##  7 Adelie  Torgersen           38.9          17.8               181        3625
##  8 Adelie  Torgersen           39.2          19.6               195        4675
##  9 Adelie  Torgersen           34.1          18.1               193        3475
## 10 Adelie  Torgersen           42            20.2               190        4250
## # ... with 334 more rows, and 3 more variables: sex <chr>, year <int>,
## #   body_mass_g/1000 <dbl>
```

This creates a copy of `penguins` with a new column called `body_mass_g/1000`

(look at the bottom of the printed output). That is not a very good name but do not worry—we will improve on it in a moment. Most of the rules that apply to `select` also apply to `mutate`:

- Quotes must not be placed around an expression that performs a calculation. This makes sense because the expression is meant to be evaluated so that it "does something". It is not a value.
- The `mutate` function does not have side effects, meaning it does not change the original `penguins` in any way. In the example, we printed the result produced by `mutate` rather than assigning it a name using `<-`, which means we have no way to access the result.
- The `mutate` function returns the same kind of object as the one it is working on: a data frame if our data was originally in a data frame, a tibble if it was a tibble.

Creating a variable called something like `body_mass_g/1000` is not ideal because that is a difficult name to work with. Fortunately, the `mutate` function can name new variables at the same time as it creates them. We just name the arguments using `=`, placing the name on the left-hand side. Look at how to use this construct to name the new area variable `body_mass_kg`:

```
mutate(penguins, body_mass_kg = body_mass_g / 1000)
```

```
## # A tibble: 344 x 9
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <chr>   <chr>              <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen           39.1          18.7               181        3750
##  2 Adelie  Torgersen           39.5          17.4               186        3800
##  3 Adelie  Torgersen           40.3          18                 195        3250
##  4 Adelie  Torgersen           NA            NA                  NA          NA
##  5 Adelie  Torgersen           36.7          19.3               193        3450
##  6 Adelie  Torgersen           39.3          20.6               190        3650
##  7 Adelie  Torgersen           38.9          17.8               181        3625
##  8 Adelie  Torgersen           39.2          19.6               195        4675
##  9 Adelie  Torgersen           34.1          18.1               193        3475
## 10 Adelie  Torgersen           42            20.2               190        4250
## # ... with 334 more rows, and 3 more variables: sex <chr>, year <int>,
## #   body_mass_kg <dbl>
```

We can create more than one variable by supplying `mutate` multiple (named) arguments:

```
mutate(penguins,
       bill_size = bill_depth_mm * bill_length_mm,
       scaled_bill_size = bill_size / body_mass_g)
```

```
## # A tibble: 344 x 10
```

```
##     species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##     <chr>   <chr>               <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen            39.1          18.7               181        3750
##  2 Adelie  Torgersen            39.5          17.4               186        3800
##  3 Adelie  Torgersen            40.3          18                 195        3250
##  4 Adelie  Torgersen            NA            NA                  NA          NA
##  5 Adelie  Torgersen            36.7          19.3               193        3450
##  6 Adelie  Torgersen            39.3          20.6               190        3650
##  7 Adelie  Torgersen            38.9          17.8               181        3625
##  8 Adelie  Torgersen            39.2          19.6               195        4675
##  9 Adelie  Torgersen            34.1          18.1               193        3475
## 10 Adelie  Torgersen            42            20.2               190        4250
## # ... with 334 more rows, and 4 more variables: sex <chr>, year <int>,
## #   bill_size <dbl>, scaled_bill_size <dbl>
```

Notice that we placed each calculation on a new line, remembering to use a comma to separate arguments. We can do this because R ignores white space. Splitting long a function call across multiple lines in this way is helpful because it makes it easier to read and understand the sequence of calculations.

This last example reveals a nice feature of `mutate`—we can use newly created variables in further calculations. Here we constructed a synthetic bill size variable, and used that to calculate a second variable representing the ratio of bill size to body mass.

7.3.1 Transforming and dropping variables

Occasionally we need to construct one or more new variables and then drop all the other ones in the original dataset. The `transmute` function is designed to do this. It works exactly like `mutate`, but it has a slightly different behaviour:

```
transmute(penguins,
          bill_size = bill_depth_mm * bill_length_mm,
          scaled_bill_size = bill_size / body_mass_g)
```

```
## # A tibble: 344 x 2
##    bill_size scaled_bill_size
##        <dbl>            <dbl>
##  1      731.            0.195
##  2      687.            0.181
##  3      725.            0.223
##  4       NA            NA
##  5      708.            0.205
##  6      810.            0.222
##  7      692.            0.191
##  8      768.            0.164
##  9      617.            0.178
```

```
## 10      848.             0.200
## # ... with 334 more rows
```

Here we repeated the previous example, but now only the new variables were retained in the resulting tibble. If we also want to retain additional variables without altering them, we can pass them as unnamed arguments. For example, to retain `species` identity in the output, use:

```
transmute(penguins,
          species,
          bill_size = bill_depth_mm * bill_length_mm,
          scaled_bill_size = bill_size / body_mass_g)
```

```
## # A tibble: 344 x 3
##    species bill_size scaled_bill_size
##    <chr>       <dbl>            <dbl>
##  1 Adelie       731.            0.195
##  2 Adelie       687.            0.181
##  3 Adelie       725.            0.223
##  4 Adelie        NA            NA
##  5 Adelie       708.            0.205
##  6 Adelie       810.            0.222
##  7 Adelie       692.            0.191
##  8 Adelie       768.            0.164
##  9 Adelie       617.            0.178
## 10 Adelie       848.            0.200
## # ... with 334 more rows
```

Chapter 8

Working with observations

8.1 Introduction

This chapter will explore the `filter` and `arrange` verbs. We discuss these functions together because they manipulate observations (i.e. rows) of a data frame or tibble:

- The `filter` function extracts a subset of observations based on supplied criteria.
- The `arrange` function reorders the rows according to the values in one or more variables.

8.1.1 Getting ready

We'll be using the **dplyr** package, so we need to remember to load and attach the package in the current session:

```
library("dplyr")
```

We'll use the Palmer penguins data again to illustrate the ideas in this chapter. The examples below assume those data been read into R as a tibble with the name `penguins`.

8.2 Relational and logical operators

Most `filter` operations rely on some combination of **relational and logical operators**. Relational operators allow us to ask questions like, "are the values of 'x' greater than those of 'y': `x > y`". These sorts of comparisons are used by R to express whether or not a particular condition is met (because they generate

a logical vector of TRUE/FALSE values). Logical operators allow us to combine such conditions, thereby building up complex conditions from simpler ones.

This is best understood by example. We'll do that in a moment. For now, simply make a mental note of the different relational and logical operators:

1. Use **relational operators** to make comparisons between a pair of variables on the basis of conditions like 'less than' or 'equal to':

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

2. Use **logical operators** to connect two or more comparisons to arrive at a single overall criterion:

   - `x & y`: are both x AND y true?
   - `x | y`: is x OR y true?

> ⚠ **Double == or single =?**
>
> Remember to use 'double equals' `==` when testing for equivalence between `x` and `y`. We all forget this from time to time and use 'single equals' `=` instead. This will lead to an error. **dplyr** is pretty good at spotting this mistake these days and will warn you in its error message that you used `=` when you meant to use `==`. Of course, if you don't read the error messages, you won't benefit from this helpful behaviour.

8.3 Subset observations with `filter`

We use `filter` to **subset observations** in a data frame or tibble containing our data. This is useful when we want to limit an analysis to a particular group of observations. Basic usage of `filter` looks something like this:

```
filter(<data>, <expression-1>, <expression-2>, ...)
```

Yes, this is pseudocode again. Let's review the arguments:

- The first argument, `<data>`, must be the name of the object (usually a data frame or tibble) containing our data. As with all **dplyr** verbs, this is not optional.
- We then include one or more additional arguments. Each of these is a valid R expression involving one or more variables in `<data>` that returns a logical vector. We've expressed these as `<expression-1>, <expression-2>,`

..., where `<expression-1>` and `<expression-2>` represent the first two expressions, and the `...` is acting as placeholder for the remaining expressions.

To see `filter` in action, we'll use it to subset observations in the `penguins` dataset, based on two relational criteria:

```
filter(penguins, bill_length_mm > 45, bill_depth_mm > 18)
```

```
## # A tibble: 44 x 8
##    species   island    bill_length_mm bill_depth_mm flipper_length_~ body_mass_g
##    <chr>     <chr>              <dbl>         <dbl>            <int>       <int>
##  1 Adelie    Torgersen          46            21.5              194        4200
##  2 Adelie    Torgersen          45.8          18.9              197        4150
##  3 Adelie    Biscoe             45.6          20.3              191        4600
##  4 Chinstrap Dream              50            19.5              196        3900
##  5 Chinstrap Dream              51.3          19.2              193        3650
##  6 Chinstrap Dream              45.4          18.7              188        3525
##  7 Chinstrap Dream              52.7          19.8              197        3725
##  8 Chinstrap Dream              46.1          18.2              178        3250
##  9 Chinstrap Dream              51.3          18.2              197        3750
## 10 Chinstrap Dream              46            18.9              195        4150
## # ... with 34 more rows, and 2 more variables: sex <chr>, year <int>
```

In this example, we've created a subset of `penguins` that only includes observations where the `bill_length_mm` variable is greater than 45 **and** the `bill_depth_mm` variable is greater than 45, i.e. both conditions must be met for an observation to be retained. This is probably starting to feel repetitious, but there are a few features of `filter` that we should be aware of:

- We do not surround each expression with quotes. The expression is meant to be evaluated—it is not 'a value.
- The result produced by `filter` was printed to the Console in the example. The `filter` function did not change the original `penguins` in any way (no side effects!).
- The `filter` function will return the same kind of data object it is working on: it returns a data frame if our data was originally in a data frame, and a tibble if it was a tibble.

Notice that including are two conditions separated by a comma means both conditions have to be met. There is another way to achieve the exact same result:

```
filter(penguins, bill_length_mm > 45 & bill_depth_mm > 18)
```

```
## # A tibble: 44 x 8
##    species   island    bill_length_mm bill_depth_mm flipper_length_~ body_mass_g
```

```
##    <chr>     <chr>              <dbl>       <dbl>            <int>          <int>
##  1 Adelie    Torgersen          46          21.5             194           4200
##  2 Adelie    Torgersen          45.8        18.9             197           4150
##  3 Adelie    Biscoe             45.6        20.3             191           4600
##  4 Chinstrap Dream              50          19.5             196           3900
##  5 Chinstrap Dream              51.3        19.2             193           3650
##  6 Chinstrap Dream              45.4        18.7             188           3525
##  7 Chinstrap Dream              52.7        19.8             197           3725
##  8 Chinstrap Dream              46.1        18.2             178           3250
##  9 Chinstrap Dream              51.3        18.2             197           3750
## 10 Chinstrap Dream              46          18.9             195           4150
## # ... with 34 more rows, and 2 more variables: sex <chr>, year <int>
```

This version links the two parts with the logical `&` operator. That is, rather than supplying `bill_length_mm > 45` and `bill_depth_mm > 18` as two arguments, we used a single R expression, combining them with the `&`.

We're pointing this out because we sometimes need to create filtering criteria that cannot be expressed as 'condition 1' **and** 'condition 2' **and** 'condition 3'... etc. Under those conditions we have to use logical operators to connect conditions. A simple instance of this situation is where we need to subset on an **either/or** basis. For example:

```
filter(penguins, bill_length_mm < 36 | bill_length_mm > 54)
```

```
## # A tibble: 29 x 8
##    species island     bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <chr>   <chr>               <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen            34.1          18.1               193        3475
##  2 Adelie  Torgersen            34.6          21.1               198        4400
##  3 Adelie  Torgersen            34.4          18.4               184        3325
##  4 Adelie  Biscoe               35.9          19.2               189        3800
##  5 Adelie  Biscoe               35.3          18.9               187        3800
##  6 Adelie  Biscoe               35            17.9               190        3450
##  7 Adelie  Biscoe               34.5          18.1               187        2900
##  8 Adelie  Biscoe               35.7          16.9               185        3150
##  9 Adelie  Biscoe               35.5          16.2               195        3350
## 10 Adelie  Torgersen            35.9          16.6               190        3050
## # ... with 19 more rows, and 2 more variables: sex <chr>, year <int>
```

This creates a subset of `penguins` that only includes observation where `bill_length_mm` is less than 36 **or** (`|`) greater than 54. This creates a subset of the data associated with the more 'extreme' values of bill length (unusually small or large).

We're not limited to using relational and logical operators when working with `filter`. The conditions specified in the `filter` function can be any expression

that returns a logical vector. The only constraint is that the output vector's length has to equal its input's length, or be a single logical values (`TRUE` or `FALSE`).

Here's an example. The **dplyr `between`** function is used to determine whether the values of a numeric vector fall in a specified range. It has three arguments: the numeric vector to filter on and the lower and upper and boundary values. For example:

```
filter(penguins, between(bill_length_mm, 36, 54))
```

```
## # A tibble: 313 x 8
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <chr>   <chr>              <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen           39.1          18.7               181        3750
##  2 Adelie  Torgersen           39.5          17.4               186        3800
##  3 Adelie  Torgersen           40.3          18                 195        3250
##  4 Adelie  Torgersen           36.7          19.3               193        3450
##  5 Adelie  Torgersen           39.3          20.6               190        3650
##  6 Adelie  Torgersen           38.9          17.8               181        3625
##  7 Adelie  Torgersen           39.2          19.6               195        4675
##  8 Adelie  Torgersen           42            20.2               190        4250
##  9 Adelie  Torgersen           37.8          17.1               186        3300
## 10 Adelie  Torgersen           37.8          17.3               180        3700
## # ... with 303 more rows, and 2 more variables: sex <chr>, year <int>
```

8.4 Reordering observations with `arrange`

We use **arrange** to **reorder the rows** of a data frame or tibble. Basic usage of **arrange** looks like this:

```
arrange(<data>, <variable-1>, <variable-2>, ...)
```

Yes, this is pseudocode. As always, the first argument, `<data>`, is the name of the object containing our data. We then include a series of one or more additional arguments, where each of these is the name of a variable in `<data>`: `<variable-1>` and `<variable-2>` are names of the first two ordering variables, and the `...` is acting as a placeholder for the remaining variables.

To see **arrange** in action, let's construct a new version of `penguins` where the rows have been reordered first by `flipper_length_mm`, and then by `body_mass_g`:

```
arrange(penguins, flipper_length_mm, body_mass_g)
```

```
## # A tibble: 344 x 8
```

```
##     species  island    bill_length_mm bill_depth_mm flipper_length_~ body_mass_g
##     <chr>    <chr>              <dbl>         <dbl>            <int>       <int>
##  1 Adelie    Biscoe              37.9          18.6              172        3150
##  2 Adelie    Biscoe              37.8          18.3              174        3400
##  3 Adelie    Torgersen           40.2          17                176        3450
##  4 Adelie    Dream               33.1          16.1              178        2900
##  5 Adelie    Dream               39.5          16.7              178        3250
##  6 Chinstrap Dream               46.1          18.2              178        3250
##  7 Adelie    Dream               37.2          18.1              178        3900
##  8 Adelie    Dream               37.5          18.9              179        2975
##  9 Adelie    Dream               42.2          18.5              180        3550
## 10 Adelie    Biscoe              37.7          18.7              180        3600
## # ... with 334 more rows, and 2 more variables: sex <chr>, year <int>
```

This creates a new version of `penguins` where the rows are sorted according to the values of by `flipper_length_mm` and `body_mass_g` in ascending order – i.e. from smallest to largest. Look at the cases where flipper length is 178 mm. What do these show? Since `flipper_length_mm` was placed before `body_mass_g` in the arguments, the values of `body_mass_g` are only used to break ties within any particular value of `flipper_length_mm`.

For the sake of avoiding doubt about how `arrange` works, we will quickly review its behaviour. It works the same as every other **dplyr** verb we have looked at:

- The variable names used as arguments of `arrange` are not surrounded by quotes.
- The `arrange` function did not change the original `penguins` in any way.
- The `arrange` function will return the same kind of data object it is working on.

`arrange` sorts variables in ascending order by default. If we need it to sort a variable in descending order, we wrap the variable name in the **dplyr** `desc` function:

```
arrange(penguins, flipper_length_mm, desc(body_mass_g))
```

```
## # A tibble: 344 x 8
##     species  island    bill_length_mm bill_depth_mm flipper_length_~ body_mass_g
##     <chr>    <chr>              <dbl>         <dbl>            <int>       <int>
##  1 Adelie    Biscoe              37.9          18.6              172        3150
##  2 Adelie    Biscoe              37.8          18.3              174        3400
##  3 Adelie    Torgersen           40.2          17                176        3450
##  4 Adelie    Dream               37.2          18.1              178        3900
##  5 Adelie    Dream               39.5          16.7              178        3250
##  6 Chinstrap Dream               46.1          18.2              178        3250
##  7 Adelie    Dream               33.1          16.1              178        2900
##  8 Adelie    Dream               37.5          18.9              179        2975
```

```
##  9 Adelie    Biscoe               40.5         18.9             180        3950
## 10 Adelie    Biscoe               38.8         17.2             180        3800
## # ... with 334 more rows, and 2 more variables: sex <chr>, year <int>
```

This creates a new version of `penguins` where the rows are sorted according to the values of `flipper_length_mm` and `body_mass_g`, in ascending and descending order, respectively. Look carefully at the values in the `flipper_length_mm` and `body_mass_g` columns to see the difference between this example and the previous one.

Chapter 9

Summarising and grouping

This chapter will explore the `summarise` and `group_by` verbs. We consider together because they are often used in combination. Their usage is also a bit different from the other **dplyr** verbs we've encountered. Here's a quick summary of what they do:

- The `group_by` function adds information to its input (a data frame or tibble), which makes subsequent calculations happen on a group-specific basis.

- The `summarise` function is a data reduction function that calculates single-number summaries of one or more variables, respecting the group structure if present.

We illustrate these ideas using the Paler penguins data set, which we assume has been read into a tibble called `peguins`.

9.1 Summarising variables with `summarise`

We use `summarise` to **calculate summaries of variables** in an object containing our data. We do this kind of calculation all the time when analysing data. In terms of pseudo-code, usage of `summarise` looks like this:

```
summarise(<data>, <expression-1>, <expression-2>, ...)
```

The first argument, `<data>`, must be the name of the data frame or tibble containing our data. We then include a series of one or more additional arguments; each of these is a valid R expression involving at least one variable in `<data>`. These are given by the pseudo-code placeholder `<expression-1>`, `<expression-2>`, ..., where `<expression-1>` and `<expression-2>` represent the first two expressions, and the `...` is acting as placeholder for the remaining

97

expressions. These expressions can be any calculation involving R functions that returns a vector of some kind.

The `summarise` function seems to work a lot like `mutate`. The main difference is that the expressions `mutate` uses have to all return a vector of the same length as their inputs. In contrast, `summarise` expressions used all have to produce the same length output, but those outputs can be any length. They often return a single value because they are **summarising** the data in some way, but they don't have to.

The `summarise` verb is best understood by example. The **dplyr** function `n_distinct` takes a calculates the number of distinct (i.e. unique) cases in a vector. We can use `n_distinct` with `summarise` to calculate the number of unique vales of the `bill_length_mm` and `bill_depth_mm` variables like this:

```
summarise(penguins, n_distinct(bill_length_mm), n_distinct(bill_depth_mm))
```

```
## # A tibble: 1 x 2
##   `n_distinct(bill_length_mm)` `n_distinct(bill_depth_mm)`
##                          <int>                       <int>
## 1                          165                          81
```

Notice what kind of object `summarise` returns—it's a tibble with one row and two columns: two columns because we calculated two counts, and one row containing because we only one set of counts. There are a few other things to note about how `summarise` works:

- The expression that performs each calculation is not surrounded by quotes because it's an expression that it 'does a calculation'.
- The order of the columns in the output is the same as the order in which they were created in the `<expression-1>, <expression-2>, ...` list.
- `summarise` returns the same kind of data object as its input—it returns a data frame if our data was originally in a data frame, or a tibble if it was in a tibble.
- If we don't specify a name `summarise` uses the actual R expression to name the columns of its output (e.g. `n_distinct(bill_length_mm)`)

Variable names based on the calculation (e.g. `n_distinct(bill_length_mm)`) are not ideal because they are long and contain special reserved characters like `(`. This makes it difficult refer to columns in the output because we have to remember to place back ticks (`` ` ``) around their name whenever we want to refer to them.

Fortunately, the `summarise` function can name the new variables at the same time as they are created (just like `mutate`). We do this by naming the arguments using `=`, placing the name we require on the left hand side. For example:

```
summarise(penguins,
          n_bill_length = n_distinct(bill_length_mm),
          n_bill_depth  = n_distinct(bill_depth_mm))
```

```
## # A tibble: 1 x 2
##   n_bill_length n_bill_depth
##           <int>        <int>
## 1           165           81
```

This time we end up with summary data set that has reasonable column names. Notice how we organised that example—we placed each calculation on a new line. We don't have to do this, but since R doesn't care about white space, we can use newlines and spaces to keep everything more human-readable. It pays to organise `summarise` calculations like this when they become longer.

9.1.1 More complicated calculations with `summarise`

Many useful base R functions can be used with `summarise`. Of particular value are those that calculate various summaries of numeric variables are, such as:

- `min` and `max` calculate the minimum and maximum values,
- `mean` and `median` calculate the mean and median, and
- `sd` and `var` calculate the standard deviation and variance.

We do need to pay attention when using base R functions with **dplyr**. Take a look at this attempt to use summarise to calculate the mean of `bill_length_mm` and `bill_length_mm`:

```
summarise(penguins,
          n_bill_length = mean(bill_length_mm),
          n_bill_depth  = mean(bill_depth_mm))
```

```
## # A tibble: 1 x 2
##   n_bill_length n_bill_depth
##           <dbl>        <dbl>
## 1            NA           NA
```

No numbers—just a pair of `NA`s. We forgot about the presence of missing values in the `penguins` data. Both `bill_length_mm` and `bill_depth_mm` contain missing values. When the `mean` function encounters even one missing value in its input its default behaviour is to spit out `NA`. It is possible to change that behaviour by setting the `na.rm` argument of `mean`:

```
summarise(penguins,
          n_bill_length = mean(bill_length_mm, na.rm = TRUE),
          n_bill_depth  = mean(bill_depth_mm,  na.rm = TRUE))
```

```
## # A tibble: 1 x 2
##   n_bill_length n_bill_depth
##           <dbl>        <dbl>
## 1          43.9         17.2
```

This example demonstrates something important—the functions we use within `summarise` often have their own arguments, and we sometimes need to set those arguments to perform the calculation we want.

Almost any R code can be used as `summarise` expressions. This means we can combine more than one function to build up arbitrarily complicated calculations. For example, if we need to know the ratio of the mean bill length and mean bill width in `penguins`, we would use:

```
summarise(penguins,
  ratio = mean(bill_length_mm, na.rm = TRUE) / mean(bill_depth_mm,  na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   ratio
##   <dbl>
## 1  2.56
```

The ability to work with arbitrary expressions makes `summarise` (and `mutate`) very powerful.

9.2   Grouped operations using `group_by`

Performing a calculation with one or more variables using the whole data set can be useful. However, we often need to carry out calculations on different subsets of our data. For example, it's more useful to know how the mean bill length and depth vary among the different species in the `penguins` data set, rather than knowing the overall mean of these traits. We could calculate separate means by using `filter` to create different subsets of `penguins`, and then use `summary` on each of these to calculate the means. This would get the job done, but it's inefficient and quickly becomes tiresome if we have to work with many groups.

The `group_by` function provides an elegant solution to this kind of problem. All the `group_by` function does is add a bit of information to a tibble or data frame. In effect, it defines subsets of data based on one or more **grouping variables**. That's all it does.

The magic happens when the grouped object is used with a **dplyr** verb like `summarise` or `mutate`. Once a the data has been tagged with grouping information, operations that involve **dplyr** verbs are carried out on separate subsets of the data—defined by the values of the grouping variable(s)—and then combined.

Basic usage of `group_by` looks like this:

```
group_by(<data>, <variable-1>, <variable-2>, ...)
```

The first argument, `<data>`, must be the name of the object containing our data. We then have to include one or more additional arguments, where each one is the name of a variable in `<data>`. We have expressed this as `<variable-1>`, `<variable-2>`, `...`, where `<variable-1>` and `<variable-2>` are names of the first two variables, and the `...` is acting as a placeholder for the remaining variables.

We'll illustrate `group_by` by using it alongside `summarise`. We're aiming to calculate the mean bill length for each species in `penguins`. This is a two-step process. The first step uses `group_by` to add grouping information to `penguins`. Take a look at what we end up with when we do that:

```
group_by(penguins, species)
```

```
## # A tibble: 344 x 8
## # Groups:   species [3]
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <chr>   <chr>              <dbl>         <dbl>             <int>        <int>
##  1 Adelie  Torgersen           39.1          18.7               181         3750
##  2 Adelie  Torgersen           39.5          17.4               186         3800
##  3 Adelie  Torgersen           40.3          18                 195         3250
##  4 Adelie  Torgersen           NA            NA                  NA           NA
##  5 Adelie  Torgersen           36.7          19.3               193         3450
##  6 Adelie  Torgersen           39.3          20.6               190         3650
##  7 Adelie  Torgersen           38.9          17.8               181         3625
##  8 Adelie  Torgersen           39.2          19.6               195         4675
##  9 Adelie  Torgersen           34.1          18.1               193         3475
## 10 Adelie  Torgersen           42            20.2               190         4250
## # ... with 334 more rows, and 2 more variables: sex <chr>, year <int>
```

Compare this to the output produced when we print the original `penguins` data set:

```
penguins
```

```
## # A tibble: 344 x 8
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <chr>   <chr>              <dbl>         <dbl>             <int>        <int>
##  1 Adelie  Torgersen           39.1          18.7               181         3750
##  2 Adelie  Torgersen           39.5          17.4               186         3800
##  3 Adelie  Torgersen           40.3          18                 195         3250
##  4 Adelie  Torgersen           NA            NA                  NA           NA
##  5 Adelie  Torgersen           36.7          19.3               193         3450
```

```
##  6 Adelie  Torgersen          39.3          20.6           190         3650
##  7 Adelie  Torgersen          38.9          17.8           181         3625
##  8 Adelie  Torgersen          39.2          19.6           195         4675
##  9 Adelie  Torgersen          34.1          18.1           193         3475
## 10 Adelie  Torgersen          42            20.2           190         4250
## # ... with 334 more rows, and 2 more variables: sex <chr>, year <int>
```

There is very little difference—**group_by** really doesn't do much on its own. The main change is that printing the tibble resulting from the **group_by** operation shows a bit of additional information at the top: **Groups: species [3]**. This tells us that the tibble is now grouped by the **species** variable. The **[3]** part tells us that there are three different groups (i.e. species of penguin). The **only** thing **group_by** did was add this grouping information to a copy of **penguins**.

The original **penguins** object was not altered in any way. If we want to do anything useful with the grouped tibble we need to assign it a name so that we can work with it:

```
penguins_by_species <- group_by(penguins, species)
```

Now we have a grouped tibble called **penguins_by_species** in which the value of **species** define the different groups—any row where **species** is equal to 'Adelie' is assigned to the first group, any row where **species** is equal to 'Chinstrap' is assigned to a second group, and any row where **species** is equal to 'Gentoo' is assigned to a third group.

dplyr operations on this tibble will now be performed on a 'by group' basis. To see this in action, we use **summarise** to calculate the mean bill length:

```
summarise(penguins_by_species,
          mean_bill_length = mean(bill_length_mm, na.rm = TRUE))
```

```
## # A tibble: 3 x 2
##   species    mean_bill_length
##   <chr>                 <dbl>
## 1 Adelie                 38.8
## 2 Chinstrap              48.8
## 3 Gentoo                 47.5
```

This is part two of the two-step process mentioned above. When we used **summarise** on an ungrouped object, the result was a tibble with one row— the overall global mean. Now the resulting tibble has three rows, one for each species in the data set. The **mean_bill_length** column shows the mean bill lengths for each species. The **species** column tells us what species each mean belongs to. Notice that **summarise** also printed an (un)helpful message:

```
`summarise()` ungrouping output (override with `.groups` argument)
```

There's no need to worry about this. It is simply saying that `summarise` has removed the grouping information from the resulting tibble.

We can also carry out multiple calculations with grouped data if we need to. For example, if we need to calculate the mean bill length and mean bill depth for each species, we would use the grouped version of `penguins` like this:

```
summarise(penguins_by_species,
          mean_bill_length = mean(bill_length_mm, na.rm = TRUE),
          mean_bill_depth  = mean(bill_depth_mm,  na.rm = TRUE))
```

```
## # A tibble: 3 x 3
##   species   mean_bill_length mean_bill_depth
##   <chr>                <dbl>           <dbl>
## 1 Adelie                38.8            18.3
## 2 Chinstrap             48.8            18.4
## 3 Gentoo                47.5            15.0
```

9.2.1 More than one grouping variable

What if we need to calculate summaries using more than one grouping variable? The workflow is unchanged. Assume we need to know the mean body mass of males and females of each penguin species. First, we make a grouped copy of `penguins` using the appropriate grouping variables:

```
penguins_by_species_sex <- group_by(penguins, species, sex)
```

We called the grouped tibble `penguins_by_species_sex`. Look at what happens when we print this:

```
penguins_by_species_sex
```

```
## # A tibble: 344 x 8
## # Groups:   species, sex [8]
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <chr>   <chr>              <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen           39.1          18.7               181        3750
##  2 Adelie  Torgersen           39.5          17.4               186        3800
##  3 Adelie  Torgersen           40.3          18                 195        3250
##  4 Adelie  Torgersen           NA            NA                 NA          NA
##  5 Adelie  Torgersen           36.7          19.3               193        3450
##  6 Adelie  Torgersen           39.3          20.6               190        3650
##  7 Adelie  Torgersen           38.9          17.8               181        3625
##  8 Adelie  Torgersen           39.2          19.6               195        4675
##  9 Adelie  Torgersen           34.1          18.1               193        3475
## 10 Adelie  Torgersen           42            20.2               190        4250
```

```
## # ... with 334 more rows, and 2 more variables: sex <chr>, year <int>
```

We see `Groups: species, sex [8]` near the top, which tells us that the tibble is grouped by two variables (`species` and `sex`) with eight unique combinations of values. That seems odd at first—there are three species and two sexes represented in this dataset, which gives six possible combinations at most.

The reason for the discrepancy becomes clear when we move on to calculate the mean body mass for each sex-species combination:

```
summarise(penguins_by_species_sex,
          body_mass_g = mean(body_mass_g, na.rm = TRUE))
```

```
## `summarise()` has grouped output by 'species'. You can override using the `.groups`

## # A tibble: 8 x 3
## # Groups:   species [3]
##    species   sex     body_mass_g
##    <chr>     <chr>         <dbl>
## 1 Adelie    female        3369.
## 2 Adelie    male          4043.
## 3 Adelie    <NA>          3540
## 4 Chinstrap female        3527.
## 5 Chinstrap male          3939.
## 6 Gentoo    female        4680.
## 7 Gentoo    male          5485.
## 8 Gentoo    <NA>          4588.
```

This shows mean body mass for each unique combination of `species` and `sex`. The first line shows that the mean body mass associated with female Adelie penguins is 3369; the second line shows us the mean body mass associated with male Adelie penguins is 4043. The third line shows us the mean body mass of Adelie penguins where **sex is missing** (`NA`). That explains why we ended up with more groups than unique combinations of `species` and `sex` — missing values create extra groups.

9.2.2   Using `group_by` with other verbs

The `summarise` function is the **dplyr** verb that is most often used with grouped data. However, all the main **dplyr** verbs will alter their behaviour to respect group information when it is present. For example, when `mutate` or `transmute` are used with a grouped object the calculation of new variables occur "by group". Here's an example:

```
# create a data set 'mean centred' bill length variable
transmute(penguins_by_species_sex,
          body_mass_cen = body_mass_g - mean(body_mass_g, na.rm = TRUE))
```

```
## # A tibble: 344 x 3
## # Groups:   species, sex [8]
##    species sex     body_mass_cen
##    <chr>   <chr>           <dbl>
##  1 Adelie  male           -293.
##  2 Adelie  female          431.
##  3 Adelie  female         -119.
##  4 Adelie  <NA>             NA
##  5 Adelie  female           81.2
##  6 Adelie  male           -393.
##  7 Adelie  female          256.
##  8 Adelie  male            632.
##  9 Adelie  <NA>            -65
## 10 Adelie  <NA>            710
## # ... with 334 more rows
```

This calculated a standardised measure of body mass. The new `body_mass_cen` variable contains the difference between the original body mass and its mean in the appropriate species-sex group (rather than the overall mean).

9.3   Removing grouping information

On occasion, it's necessary to remove grouping information and revert to operating on the whole data set. The `ungroup` function removes grouping information:

```
ungroup(penguins_by_species)
```

```
## # A tibble: 344 x 8
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <chr>   <chr>              <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen           39.1          18.7               181        3750
##  2 Adelie  Torgersen           39.5          17.4               186        3800
##  3 Adelie  Torgersen           40.3          18                 195        3250
##  4 Adelie  Torgersen           NA            NA                 NA          NA
##  5 Adelie  Torgersen           36.7          19.3               193        3450
##  6 Adelie  Torgersen           39.3          20.6               190        3650
##  7 Adelie  Torgersen           38.9          17.8               181        3625
##  8 Adelie  Torgersen           39.2          19.6               195        4675
##  9 Adelie  Torgersen           34.1          18.1               193        3475
## 10 Adelie  Torgersen           42            20.2               190        4250
## # ... with 334 more rows, and 2 more variables: sex <chr>, year <int>
```

Looking at the top right of the printed summary, we can see that the `Group:` part is now gone—the `ungroup` function effectively recreated the original `penguins` tibble.

Chapter 10

Building pipelines

We don't often use the various **dplyr** verbs in isolation. Instead, they are combined in a sequence to prepare the data for further analysis. For example, we might create a new variable or two with `mutate` and then use `group_by` and `summarise` to calculate some numerical summaries. This chapter will introduce something called the pipe operator: `%>%`. The pipe operator's job is to allow us to represent a sequence of such steps in a transparent, readable manner.

10.1   Why do we need 'pipes'?

We've seen that carrying out calculations on a per-group basis can be achieved by grouping a tibble, assigning this a name, and then applying the `summarise` function to the new tibble. For example, in the previous chapter we saw that to calculate the mean bill length for each species in the Palmer penguins data set, **penguins**, we first create a grouped version of it:

```
penguins_by_species <- group_by(penguins, species)
```

Then we use `summarise` with the grouped version to calculate the mean bill length in each group:

```
summarise(penguins_by_species,
          mean_bill_length = mean(bill_length_mm, na.rm = TRUE))
```

```
## # A tibble: 3 x 2
##   species   mean_bill_length
##   <chr>                <dbl>
## 1 Adelie                38.8
## 2 Chinstrap             48.8
```

```
## 3 Gentoo                 47.5
```

There's nothing wrong with this way of doing things. However, building up an analysis this was is quite lengthy because we have to keep storing intermediate steps. This is especially true if an analysis involves more than a couple of steps. It also tends to clutter the global environment with many intermediate objects we don't need to keep.

One way to make things more concise is to use function nesting, like this:

```
summarise(group_by(penguins, species),
          mean_bill_length = mean(bill_length_mm, na.rm = TRUE))
```

```
## # A tibble: 3 x 2
##   species    mean_bill_length
##   <chr>                 <dbl>
## 1 Adelie                 38.8
## 2 Chinstrap              48.8
## 3 Gentoo                 47.5
```

In this version, we placed the `group_by` function call inside the list of arguments to `summarise`. Remember—we have to read nested function calls from the inside out to understand what they are doing. This is exactly equivalent to the previous example, but now we get the result without having to store intermediate data.

However, there are a couple of very good reasons why this approach is not advised:

- Experienced R users might not mind this approach because they're used to it. Nonetheless, no reasonable person would argue that nesting functions inside one another is intuitive. Reading outward from the inside of a large number of nested functions is hard work.

- Using function nesting is an error-prone approach. For example, it's very easy to accidentally put an argument on the wrong side of a closing `)`. If we're lucky, this will produce an error and we'll catch the problem. If we're not, we may end up with complete nonsense in the output.

10.2   Using pipes (%>%)

There is a better way to combing **dplyr** functions, which has the dual benefit of keeping our code concise and readable while avoiding the need to clutter the global environment with intermediate objects. This third approach involves the 'pipe operator': `%>%`. Notice there are no spaces between the three characters that make up the pipe—spaces are not allowed (e.g. `% > %` will not be recognised as the pipe).

The pipe operator isn't part of base R. Instead, **dplyr** imports it from another package and makes it available for us to use.

The pipe has become very popular in recent years. The main reason for this is because it allows us to specify a chain of function calls in a (reasonably) human readable format. Here's how we write the previous example using the pipe operator `%>%`:

```
penguins %>% group_by(., species) %>% summarise(., mean_bill_length = mean(bill_length_mm, na.rm
```

```
## # A tibble: 3 x 2
##   species    mean_bill_length
##   <chr>                 <dbl>
## 1 Adelie                 38.8
## 2 Chinstrap              48.8
## 3 Gentoo                 47.5
```

How do we make sense of this? Every time we see the `%>%` operator it means the following: take whatever is produced by the left-hand expression and use it as an argument in the function on the right-hand side. The `.` serves as a placeholder for the location of the corresponding argument. A sequence of calculations can then be read from left to right, just as we would read the words in a book. This example says, take the `penguins` data, group it by `species`, then take that grouped tibble and apply the `summarise` function to it to calculate the mean of `mean_bill_length`.

This is exactly the same calculation we did above.

When using the pipe operator we can often leave out the `.` placeholder. Remember, this signifies the argument of the function on the right of `%>%` that is associated with the result from the left of `%>%`. If we choose to leave out the `.`, the pipe operator assumes we meant to slot it into the first argument. This means we can simplify our example even more:

```
penguins %>% group_by(species) %>% summarise(mean_bill_length = mean(bill_length_mm, na.rm = TRUE
```

```
## # A tibble: 3 x 2
##   species    mean_bill_length
##   <chr>                 <dbl>
## 1 Adelie                 38.8
## 2 Chinstrap              48.8
## 3 Gentoo                 47.5
```

This is why the first argument of a dplyr verb is always the data. Adopting this convention ensures we can use `%>%` without explicitly specifying the argument to pipe into. Data goes into the pipe; data comes out of the pipe.

Remember, R does not care about white space, which means we can break a

piped set of functions over several lines to make our code more readable:

```
penguins %>%
  group_by(species) %>%
  summarise(mean_bill_length = mean(bill_length_mm, na.rm = TRUE))
```

```
## # A tibble: 3 x 2
##   species   mean_bill_length
##   <chr>                <dbl>
## 1 Adelie                38.8
## 2 Chinstrap             48.8
## 3 Gentoo                47.5
```

Now each step in the pipeline is on its own line. Most **dplyr** users use this formatting convention to improve the readability of their R code.

Finally, we do have to remember to assign the result of a chain of piped functions a name if we want to capture the result and use it later. We have to break the left to right rule a bit to do this, placing the assignment at the beginning[1]:

```
bill_length_means <-
  penguins %>%
  group_by(species) %>%
  summarise(mean_bill_length = mean(bill_length_mm, na.rm = TRUE))
```

> **ⓘ Why is %>% called the 'pipe' operator?**
>
> The %>% operator takes the output from one function and "pipes it" to another as the input. It's called 'the pipe' for the simple reason that it allows us to create an analysis 'pipeline' from a series of function calls. Incidentally, if you Google the phrase "magrittr pipe" you'll see why **magrittr** is a clever name for an R package.

One final piece of advice—make an effort to learn how to use the %>% method of piping together functions. Why? Because it's the simplest and cleanest method for doing this, many of the examples in the **dplyr** help files and on the web use it, and the majority of people carrying out real-world data wrangling with **dplyr** rely on piping.

---

[1] Actually, there is a rightward assignment operator, `->`, but let's pretend that does not exist.

Chapter 11

Helper functions

11.1 Introduction

In addition to the main **dplyr** verbs, the package provides quite a few **helper functions**. Helper functions are used in conjunction with the main verbs to make specific tasks and calculations a bit easier. Many of these are summarised in the **dplyr** cheat sheet (under *Manipulate Variables*, *Vector Functions* and *Summary Functions*). We are not going to review every single one of them in this chapter. Instead, we aim to point out where helper functions tend to be used and highlight a few of the more useful ones.

11.2 Working with `select`

There are a few helper functions that can be used with `select`. Their job is to make it easier to match variable names according to various criteria. We'll look at the three simplest of these—look at the examples in the help file for `select` and the cheat sheet to see what else is available.

We can select variables according to the sequence of characters used at the start of their name with the `starts_with` function. For example, to select all the variables in **penguins** that begin with the word "bill", we use:

```
select(penguins, starts_with("Bill"))
```

```
## # A tibble: 344 x 2
##    bill_length_mm bill_depth_mm
##             <dbl>         <dbl>
## 1            39.1          18.7
## 2            39.5          17.4
## 3            40.3          18
```

```
## 4              NA           NA
## 5              36.7         19.3
## 6              39.3         20.6
## 7              38.9         17.8
## 8              39.2         19.6
## 9              34.1         18.1
## 10             42           20.2
## # ... with 334 more rows
```

This returns a tibble containing just `bill_length_mm` and `bill_depth_mm`. There is also a helper function to select variables according to characters used at the end of their name—the `ends_with` function (no surprises there).

Notice that we quote the name we want to match against because `starts_with` expects a literal character value. This is not optional. Unusually, `starts_with` and `ends_with` are not case sensitive by default. For example, we passed `starts_with` the argument `"Bill"` instead of `"bill"`, yet it still selected variables beginning with the character string `"bill"`. If we want to select variables on a case-sensitive basis, we need to set an argument `ignore.case` to `FALSE` in `starts_with` and `ends_with`.

The last `select` helper function we'll look at is called `contains`. This one allows us to select variables based on a partial match **anywhere** in their name. Look at what happens if we pass `contains` the argument `"length"`:

```
select(penguins, contains("length"))
```

```
## # A tibble: 344 x 2
##    bill_length_mm flipper_length_mm
##             <dbl>             <int>
## 1            39.1               181
## 2            39.5               186
## 3            40.3               195
## 4            NA                  NA
## 5            36.7               193
## 6            39.3               190
## 7            38.9               181
## 8            39.2               195
## 9            34.1               193
## 10           42                 190
## # ... with 334 more rows
```

This selects all the variables with the word 'length' in their name.

There is nothing to stop us combining the different variable selection methods. For example, we can use this approach to select all the variables that have some units at the end of their names (millimetres or grams):

```
select(penguins, ends_with("_mm"), ends_with("_g"))
```

```
## # A tibble: 344 x 4
##    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##             <dbl>         <dbl>             <int>       <int>
## 1            39.1          18.7               181        3750
## 2            39.5          17.4               186        3800
## 3            40.3          18                 195        3250
## 4            NA            NA                 NA          NA
## 5            36.7          19.3               193        3450
## 6            39.3          20.6               190        3650
## 7            38.9          17.8               181        3625
## 8            39.2          19.6               195        4675
## 9            34.1          18.1               193        3475
## 10           42            20.2               190        4250
## # ... with 334 more rows
```

When we apply more than one selection criteria like this, the `select` function returns the variables that match any criteria, rather than the set that meets all of them.

11.3 Working with `mutate` and `transmute`

There are quite a few **helper functions** that can be used with `mutate`. These make it easier to carry out certain calculations that aren't easy to do with base R. We won't explore these here as they tend to be needed only in quite specific circumstances. However, in situations where we need to construct an unusual variable, it's worth looking at that handy cheat sheat to see what options might be available.

We will look at one particularly useful helper function that's used a lot when we need to recode a particular variable using `mutate`. The function is called `case_when`. It works by setting up a series of paired matching criteria and replacement values. For example, imagine that we want to replace the names in `species` with three-letter shortcodes for each species. This is how to achieve that using `case_when` with `mutate`:

```
penguins %>%
  mutate(species = case_when(
          species == "Adelie"    ~ "ADL",
          species == "Gentoo"    ~ "GEN",
          species == "Chinstrap" ~ "CHN",
          TRUE ~ "UNKNOWN"
        ))
```

```
## # A tibble: 344 x 8
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <chr>   <chr>              <dbl>         <dbl>             <int>       <int>
##  1 ADL     Torgersen           39.1          18.7               181        3750
##  2 ADL     Torgersen           39.5          17.4               186        3800
##  3 ADL     Torgersen           40.3          18                 195        3250
##  4 ADL     Torgersen           NA            NA                  NA          NA
##  5 ADL     Torgersen           36.7          19.3               193        3450
##  6 ADL     Torgersen           39.3          20.6               190        3650
##  7 ADL     Torgersen           38.9          17.8               181        3625
##  8 ADL     Torgersen           39.2          19.6               195        4675
##  9 ADL     Torgersen           34.1          18.1               193        3475
## 10 ADL     Torgersen           42            20.2               190        4250
## # ... with 334 more rows, and 2 more variables: sex <chr>, year <int>
```

The `mutate` bit of this is not new. Look at the `case_when` component—there
are four criteria. The first of these is `species == "Adelie"    ~ "ADL"`. The
way to read this is, "look for cases where the value of `species` is equal to
`"Adelie"`, and where that is true spit out the value `"ADL"`". `case_when` steps
through each criterion like this in turn, trying to find a match. The last one
`TRUE ~ "UNKNOWN"` acts as a catch-all for the non-matches.

This looks confusing at first but it does make sense with a bit of practise, and
recoding variables using `case_when` is a lot easier than going through a spread-
sheet by hand.

11.4   Working with `filter`

There aren't that many **dplyr** helper function that works with `filter`. In fact,
we've already looked at the most useful one: the `between` function. This is used
to identify cases where the values of a numeric variable lie inside a defined range.
For example, if we want all the individuals that had a body mass in the 4-5kg
range, we could use:

```
filter(penguins, between(body_mass_g, 4000, 5000))
```

```
## # A tibble: 116 x 8
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <chr>   <chr>              <dbl>         <dbl>             <int>       <int>
##  1 Adelie  Torgersen           39.2          19.6               195        4675
##  2 Adelie  Torgersen           42            20.2               190        4250
##  3 Adelie  Torgersen           34.6          21.1               198        4400
##  4 Adelie  Torgersen           42.5          20.7               197        4500
##  5 Adelie  Torgersen           46            21.5               194        4200
##  6 Adelie  Dream               39.2          21.1               196        4150
##  7 Adelie  Dream               39.8          19.1               184        4650
```

```
##  8 Adelie  Dream              44.1       19.7           196       4400
##  9 Adelie  Dream              39.6       18.8           190       4600
## 10 Adelie  Dream              42.3       21.2           191       4150
## # ... with 106 more rows, and 2 more variables: sex <chr>, year <int>
```

11.5 Working with `summarise`

There are a small number **dplyr** helper functions that can be used with
`summarise`. These generally provide summaries that aren't available directly
using base R functions. For example, we've already seen the `n_distinct`
function in action. This can be used to calculate the number of distinct values
of a variable:

```
summarise(penguins,
          num_species = n_distinct(species),
          num_island  = n_distinct(island ))
```

```
## # A tibble: 1 x 2
##   num_species num_island
##         <int>      <int>
## 1           3          3
```

This confirms what we already knew—that there are three unique species and
three unique islands in the `penguins` data set.

Part III

Exporing Data

Chapter 12

Exploratory data analysis

12.1   Introduction

Exploratory data analysis (EDA) was promoted by the statistician John Tukey in his 1977 book, "Exploratory Data Analysis". The broad goal of EDA is to help us formulate and refine hypotheses that lead to informative analyses or further data collection. The core objectives of EDA are:

- to suggest hypotheses about the causes of observed phenomena,
- to guide the selection of appropriate statistical tools and techniques,
- to assess the assumptions on which statistical analysis will be based,
- to provide a foundation for further data collection.

EDA involves a mix of numerical and visual methods of analysis. Statistical methods are sometimes used to supplement EDA. However, the main purpose of EDA is to facilitate understanding before carrying out formal statistical modelling. Even if we already know what kind of analysis we plan to pursue, it's always a good idea to **explore a data set before diving into that analysis**. At the very least, this will help determine whether or not our plans are sensible. Very often, it uncovers new patterns and insights.

In this chapter, we're going to examine some basic concepts that underpin EDA. We will:

1. see how to classify different **types of variables**,
2. distinguish between **populations and samples**, and
3. review some key **descriptive statistics**.

This will provide a conceptual foundation and vocabulary for learning how to explore data in later chapters.

12.2   Statistical variables and data

In the Data frames chapter, we pointed out that the word 'variable' can mean one of two things. In programming, a variable is a name-value association that we create when we run some code. Statisticians use the word differently. To them, a variable is any characteristic or quantity that can be measured, classified or experimentally controlled. Much of statistics is about quantifying and explaining the variation in such quantities as best we can.

Species richness, relative abundance, infection status, enzyme activity, gene frequency, and blood glucose concentration are examples of statistical variables we encounter in the biological sciences. These as statistical variables because their values vary between different observations. For instance, 'annual fitness'—measured as the number of offspring produced—is a variable that differs both among the organisms in a population and over the life cycle of a given individual.

There are different ways to describe statistical variables according to how they can be analysed, measured, or presented. It's important to be clear about what kind of variables we're dealing with because this determines how we should visualise the data, and later, how we might analyse it statistically. There are many different ways to go about classifying variables. However, we only need to be aware of two fairly simple classification schemes in this book: numeric vs categorical variables and ratio vs interval scales.

12.2.1   Numeric vs categorical variables

Numeric variables have values that describe a measurable quantity like 'how many' or 'how much' as a number. Numeric variables are also called quantitative variables; the data collected containing numeric variables are called quantitative data. Numeric variables may be classified as either continuous or discrete:

- **Continuous numeric variable**: Observations can take any value in a set of real numbers, i.e. numbers represented with decimals. Examples of continuous variables include concentration, mass, age, time, and temperature. The set of numbers a continuous variable takes is typically either 'every possible number' or 'just positive numbers'. For example, a concentration may be very large or very small, but it is strictly positive, whereas a change in concentration can be positive or negative.

- **Discrete numeric variable**: Observations can take a value based on a count from a set of whole values, e.g. 1, 2, 3, 4, 5, and so on. A discrete variable cannot take the value of a fraction between one value and the next closest value. Examples of discrete variables include the number of individuals in a population, the number of offspring produced ('reproductive fitness'), and the number of infected individuals in an experiment. All of these are measured as whole units.

Categorical variables take values that describe a characteristic of a data unit,

like 'what type' or 'which category'. Categorical variables fall into mutually exclusive (in one category or in another) and exhaustive (include all possible options) categories. Categorical variables are qualitative variables and tend to be represented by a non-numeric value; the data collected for a categorical variable are called qualitative data. Categorical variables may be further described as ordinal or nominal:

- **Ordinal variable**: Categories can be logically ordered or ranked. The categories associated with ordinal variables can be ranked higher or lower than another but do not necessarily establish a numeric difference between each category. Examples of ordinal categorical variables include academic grades (e.g. A, B, C) and size classes (e.g. small, medium, large).

- **Nominal variable**: Categories cannot be organised into a logical sequence. Examples of nominal categorical variables include sex (see *C. elegans* example), human blood group (A, B, AB and O), genotype (e.g. AA, Aa, aa), experimental conditions (e.g. control vs enhanced nutrition), and mortality status (alive vs dead).

⚠ **Do not use numbers to classify categorical variables**

Be careful when classifying variables. It can be dangerous to assume that just because a numerical scheme has been used to describe a variable, it must not be categorical. There is nothing to stop someone from using numbers to describe a categorical variable (e.g. *C. elegans* sex: Male = 1, Hermaphrodite = 2). That said, although we can use numbers to describe categories, it does not mean we should. Using numbers gets confusing and can lead to mistakes. It is much clearer to use a non-numeric recording scheme based on words or acronyms to record categorical variables (e.g. *C. elegans* sex: Male = "Male", Hermaphrodite = "Herm").

12.2.2 Ratio vs interval scales

A second way of classifying numeric variables (**not** categorical variables) relates to the scale they are measured on. The measurement scale is important because it determines how we interpret things like differences, ratios, and variability.

- **Ratio scale**: This scale does possess a meaningful zero value. It gets its name from the fact that a measurement on this scale represents a ratio between a measure of the magnitude of a quantity and a unit of the same kind. What this means in simple terms is that it is meaningful to say that something is "twice as …" as something else when working with a variable measured on a ratio scales. Ratio scales most often appear when we work with physical quantities. For example, we can say that one tree is twice as big as another or that one elephant has twice the mass of another because length and mass are measured on ratio scales.

- **Interval scale**: This allows for the degree of difference between measurements, but not the ratio between them. This kind of scale does not have a unique, non-arbitrary zero value. A good example of an interval scale is the date, which we measure relative to an arbitrary epoch (e.g. AD). It makes no sense to say that 2000 AD is twice as long as 1000 AD. However, we can compare ratios of differences on an interval scale. For example, it does make sense to talk about the amount of time between two dates, i.e. we can to say that twice as much time has passed since the epoch in 2000 AD versus 1000 AD.

Keep in mind that the distinction between ratio and interval scales is a property of the measurement scale, not the thing being measured. For example, when we measure temperature in º C we're working on an interval scale defined relative to the freezing and boiling temperatures of water under standard conditions. It doesn't make any sense to say that 30º C is twice as hot as 15º C. However, if we measured the same two temperatures on the Kelvin scale, it is meaningful to say that 303.2K is 1.05 times hotter than 288.2K. This is because the Kelvin scale is relative to a true zero (absolute zero).

12.3   Populations and samples

Whenever we collect data, we are almost always working with a sample drawn from a wider population. We might want to know something about that population, but since it is impossible to study the whole population, we study the properties of one or more samples instead. For example, a physiologist might want to know how exercise affects lung function. Since they obviously can't study every person on the planet, they have to study a small sample of people.

We mention the distinction between populations and samples because EDA is concerned with exploring the properties of samples. EDA aims to characterise a sample without trying to infer too much about the wider population from which it is derived. Learning about populations is the basis of much of statistics. That topic is best dealt with in a dedicated statistics book—not this one.

12.3.1   Sample distributions

When we say that 'EDA is concerned with exploring the properties samples,' we actually mean that EDA is concerned with the properties of **variables** in one or more samples. We can be even more precise—the property we are alluding to is the variable's **sample distribution**. The distribution of a variable describes the relative frequency with which different values occur. This is best understood by example...

Imagine we took a sample of undergraduates and measured their height. The majority of students would be around about 1.7m tall, even though there would be plenty of variation among students. Men would tend to be slightly taller than women, and very small or very tall people would be rare. We know from

experience that no one in this sample would be over 3 meters tall. These are all statements about a hypothetical sample distribution of undergraduate heights.

12.3.2 Associations

We've been talking about statistical variables as though we study them one at a time. However, most interesting samples involve more than one variable, and the goal of the ensuing data analysis is to understand associations among those variables. These associations might involve the same (e.g. numeric vs numeric) or different (e.g. numeric vs categorical) types of variable. Whatever the details, the aim of EDA is often to understand how each variable in a sample relates to the others.

12.4 Types of EDA

Exploratory data analysis involves questions such as:

- *What are the most common values of the variable*?
- *How much do observations differ from one another*?
- *Is one variable associated with another*?

Rather than describe the answers to such questions in purely verbal terms, as we did above, EDA relies on descriptive statistics and graphical summaries:

1. **Descriptive statistics**. Descriptive statistics are used to quantify the basic features of a sample distribution. They provide numerical summaries about the sample that can be used to make comparisons and draw preliminary conclusions. For example, we often use the mean to summarise the 'most likely' values of a variable.

2. **Graphical summaries**. Descriptive statistics are not much use on their own—a few numbers can't capture every aspect of a distribution. Graphical summaries are a powerful complement to descriptive statistics because they capture a lot of information about a sample in a way that is easy for people to understand.

Descriptive statistics are important, but they are not the main focus of this book. The next few chapters will set out how to construct a range of useful graphical data summaries. Indeed, this book's overarching goal is to demonstrate a workflow that starts with raw data and ends in one or more exploratory plots. We do need to know a bit about descriptive statistics to understand some of those plots. To that end, we'll finish this chapter with a quick survey of descriptive statistics that will pop up later.

12.5 A primer of descriptive statistics

12.5.1 Numeric variables

So far, we've only mentioned the properties of sample distributions in very general terms—using phrases like 'most common values' and 'the range of the data'—without really saying what we mean. Statisticians have devised a set of terms to describe distributions and various descriptive statistics to quantify these. The two that matter most for numeric variables are the **central tendency** and the **dispersion**:

- A measure of **central tendency** describes a typical value of a distribution. Most people know at least one measure of central tendency. The "average" that they calculated at school is the arithmetic mean of a sample. There are many different measures of central tendency, each with its own pros and cons. Take a look at the Wikipedia page to see the most common ones. Among these, the median is the one that is used most often in exploratory analyses.

- A measure of **dispersion** describes how spread out a distribution is. Dispersion measures quantify the variability or scatter of a variable. If one distribution is more dispersed than another, it means that it encompasses a wider range of values. What this means in practice depends on the kind of measure we're working with. We tend to focus on the variance, the standard deviation, and the interquartile range. There are many others, though.

> **Beyond central tendency and dispersion**
>
> Another important aspect of a distribution is its **skewness** (a.k.a. 'skew'). There are many different ways to quantify skewness. Unfortunately, these are quite difficult to make sense of. For now, we only need to understand what skewness means in qualitative terms. Skewness refers to the (a)symmetry of a distribution. When we talk about distributions with high skew, we mean they are very asymmetric.

12.5.1.1 Central tendency

The central tendency of a numeric variable's sample distribution is typically described using either the arithmetic mean or the median. The **arithmetic mean** of a sample is 'the mean' that everyone learns at school[1]. Most people have calculated the mean by hand at some point. As R users, we can use the `mean` function to calculate the arithmetic mean if we need it. For example, this will calculate the mean body mass in the `penguins` data set:

---

[1]People often just say 'the mean' when referring to the arithmetic sample mean. This is fine, but keep in mind that there are other kinds of mean , such as the harmonic mean and the geometric mean.

```
mean(penguins$body_mass_g, na.rm = TRUE)
```

```
## [1] 4201.754
```

Remember—we used `na.rm = TRUE` here because `body_mass_g` contains a small number of missing (`NA`) values. This calculation tells us the arithmetic mean of body mass is 4202 grams, i.e. in some sense, the most common body mass is about 4202 grams.

One limitation of the arithmetic mean is that it is affected by the shape of a distribution. This is why, for example, it does not make much sense to look at the mean income of workers to get a sense of what a 'typical' person earns. Income distribution are highly skewed, such that a few people receive very large salaries compared to the vast majority of the population. Those few who earn very good salaries tend to shift the mean well past anything that is really 'typical'.

Because the sample mean is sensitive to the shape of a distribution in this way, we often prefer to use a more robust measure of central tendency—the **sample median**. The median of a sample is the value that separates the upper half from the lower half. We can find the sample median with the `median` function in R:

```
median(penguins$body_mass_g, na.rm = TRUE)
```

```
## [1] 4050
```

This tells us that the arithmetic mean of body mass in the `penguins` data is 4050 grams. This is less than the mean, reflecting the fact that the body mass distribution is somewhat asymmetric.

> **What about 'the mode'?**
>
> The **mode** of a variable's distribution is simply the value that is most likely to occur. This is a simple idea. Unfortunately, it is often difficult to estimate the mode from a sample. Nonetheless, it is important to know what the mode represents, because the concept is useful even when the actual value is hard to estimate.

12.5.1.2 Dispersion

There are many ways to quantify the dispersion of a sample distribution. The most important quantities from the standpoint of statistics are the sample **variance** and **standard deviation**. The sample variance (s^2) is the average squared deviations of each point from the mean. Variances are non-negative. The larger the variance, the more observations are spread out around the mean. A variance of zero only occurs if all values are identical.

We won't waste time showing the formula because we'll never actually need to use it directly. As usual, R can calculate the sample variance if we need it:

```
var(penguins$body_mass_g, na.rm = TRUE)
```

```
## [1] 643131.1
```

That's a big number. What does it mean? Is it 'big' or is it 'small'? No idea. That's the problem with variances—they are difficult to interpret because their calculation involves squared deviations. The variance is an important quantity in statistics because many common tools use changes in the variance as a basis for statistical tests. However, variances seldom feature in EDA because they are so hard to interpret.

A somewhat better descriptive statistic is to describe sample dispersion is a closely related quantity called the **standard deviation** of the sample, usually denoted s. The standard deviation is equal to the square root of the variance. We calculate it using the `sd` function:

```
sd(penguins$body_mass_g, na.rm = TRUE)
```

```
## [1] 801.9545
```

Why do we prefer the standard deviation over the variance? Because it is the square root of the variance, it operates on the same scale as the variable it summarises. This means it reflects the dispersion we perceive in the data. The sample standard deviation is not without problems, though. Like the sample mean, it is sensitive to the shape of a distribution and the presence of outliers.

A measure of dispersion that is robust to these kinds of problems is the **interquartile range**. The interquartile range (IQR) is defined as the difference between the third and first quartile (see box). The IQR contains the middle 50% of the values of a variable. The more spread out the data, the larger the IQR. People prefer IQR to measure dispersion for exploratory work because it only depends on the 'middle' of a distribution. This makes it robust to the presence of outliers.

We can use the `IQR` function to find the interquartile range of the body mass variable:

```
IQR(penguins$body_mass_g, na.rm = TRUE)
```

```
## [1] 1200
```

The IQR is used as the basis for a useful data summary plot called a 'box and whiskers' plot. We'll see how to construct this later.

> **What are quartiles?**
>
> We need to know what a quartile is to understand the interquartile range. Three quartiles are defined for any sample. These divide the data into four equal-sized groups, from the set of smallest numbers up to the set of largest numbers. The second quartile (Q_2) is the median, i.e. it divides the data into an upper and lower half. The first quartile (Q_1) is the number that divides the lower 50% of values into two equal-sized groups. The third quartile (Q_3) is the number that divides the upper 50% of values into two equal-sized groups.

12.5.2 Categorical variables

Descriptive statistics of categorical variables aim to quantify specific features of their sample distribution, just as with numeric variables. The general question we need to address is, what are the relative frequencies of different categories? Because categorical variables take a finite number of values, the simplest thing we can do is tabulate the number of occurrences of each type. We can use the **dplyr** `count` function to do this:

```
penguins %>% count(species)
```

```
## # A tibble: 3 x 2
##   species        n
##   <chr>      <int>
## 1 Adelie       152
## 2 Chinstrap     68
## 3 Gentoo       124
```

This prints that the number of observations associated with each species in `penguins`. The summary reveals that the most common species in the data set is the Adelie penguin, followed by the Gentoo and Chinstrap.

Can we quantify the central tendency of a categorical sample distribution? Various measures exist. We can certainly find the **sample mode** of a categorical variable easily enough. This is just the most common category. In the case of the above `species` variable, the mode is obviously Adelie. It is also possible to calculate a **sample median** of a categorical variable, but only when it is ordinal. Since the median value is the one that lies in the middle of an ordered set of values, it makes no sense to talk about the middle of a set of nominal values that have no inherent order.

What about dispersion? Well, measures of dispersion for categorical variables do exist, but they are not very easy to interpret. They seldom get used in exploratory data analysis so let's not worry about them here.

12.5.3   Associations

Statisticians have devised different ways to quantify an association between variables. The common measures calculate some kind of **correlation coefficient**. The terms 'association' and 'correlation' are closely related, so they are often used interchangeably. Strictly speaking, correlation has a narrower definition: a correlation is defined by a metric (the 'correlation coefficient') that quantifies the degree to which an association tends to a certain pattern.

12.5.3.1   Pairs of numeric variables

A widely used measure of correlation for pairs of numeric variables is **Pearson's correlation coefficient** (a.k.a. Pearson product-moment correlation coefficient, or Pearson's r). Remember, a correlation coefficient quantifies the degree to which an association tends to a certain pattern. Pearson's correlation coefficient is designed to summarise the strength of a **linear** (i.e. straight line) association.

Pearson's correlation coefficient takes a value of 0 if two variables are not linearly associated and a value of +1 or -1 if they are perfectly related and represent a straight line. A positive value indicates that high values in one variable are associated with high values of the second; a negative value indicates that high values of one variable is associated with low values of the second. High values are those that are greater than the mean; low values are those that are less than the mean.

We can use the `cor` function to calculate Pearson's correlation coefficient. For example, the Pearson correlation coefficient between flipper length and body mass is given by:

```r
cor(penguins$flipper_length_mm, penguins$body_mass_g, use = "complete.obs")
```

```
## [1] 0.8712018
```

This is positive, indicating flipper length tends to increase with body mass. It is also quite close to +1, indicating the association is strong. We should interpret Pearson's correlation coefficient with care. Because it is designed to summarise the strength of a **linear** relationship, Pearson's correlation coefficient will mislead when this relationship is curved. If that statement does not make immediate sense, take a look at the famous Anscombe's quartet.

⚠ **Other measures of correlation**

What should we do if we think the relationship between two variables is non-linear? Calculate something called a **rank correlation**. The idea is quite simple. Instead of working with the actual values of each variable, we rank them by value, i.e. sort each variable from lowest to highest and

the assign the labels 1st, 2nd, 3rd, ... to observations. Rank correlations are based on the association among the ranks of two variables.

The two most popular rank correlation coefficients are Spearman's ρ ('rho') and Kendall's τ ('tau'). The differences are minimal:

- Spearman's ρ is a bit more sensitive to outliers in the data.
- Kendall's τ can be slow to calculate for large data sets.

We can either coefficient using the `cor` function, setting the `method` argument to the appropriate value: `method = "kendall"` or `method = "spearman"`. A rank correlation coefficient is interpreted in the same way as Pearson's correlation coefficient. It takes a value of 0 if the ranks are uncorrelated and +/- 1 if they are perfectly associated (though not necessarily as a straight line). The sign tells us about the direction of the association.

12.5.3.2 Pairs of categorical variables

Quantifying associations between pairs of categorical variables is not as simple as the numeric case. The general question is, "do different **combinations** of categories seem to be under- or over-represented?" We need to understand which combinations are common and which are rare. The simplest thing we can do is 'cross-tabulate' the number of occurrences (i.e. the 'frequencies') of each combination. The resulting table is called a **contingency table**.

The `xtabs` function (= 'cross-tabulation') can do this. For example, the frequencies of each penguin species and island combination is given by:

```
xtabs(~ species + island, data = penguins)
```

```
##            island
## species     Biscoe Dream Torgersen
##    Adelie       44    56        52
##    Chinstrap     0    68         0
##    Gentoo      124     0         0
```

The first argument sets the variables to cross-tabulate. `xtabs` uses R's special formula language, which means we must include that ~ symbol at the beginning. After the ~, we provide the list of variables to cross-tabulate, separated by the + sign. The second argument tells the function which data set to use.

The table above shows us how many observations are associated with each combination of the `species` and `island` categories. This particular case represents a fairly extreme example of (dis)association; the Chinstrap and Gentoo species simply don't occur on certain islands (or perhaps they weren't sampled on those islands for some reason).

What about measures of association such as correlation coefficients? Spearman's ρ and Kendall's τ are designed for numeric variables, but these can also be used to measure the correlation between ordinal variables (Kendall's τ is best). Various measures of association have been constructed for pairs of nominal variables (e.g. Cramér's V). However, none of these are widely used in exploratory data analyses—people tend to stick with graphical tools for categorical data.

Chapter 13

Introduction to ggplot2

One of the main reasons data analysts turn to R is for its strong data visualisation capabilities. The R ecosystem includes many different packages that support data visualisation. The three most widely used are: 1) the base graphics system, which uses the **graphics** package; 2) the **lattice** package; and 3) the **ggplot2** package. Each system has its own strengths and weaknesses:

- Base graphics is part of base R, which means it's always available. It's very flexible and allows us to construct more or less any plot we like. This flexibility comes at a cost, though. While it is certainly easy to get up and running with base graphics—there are specialised functions making common plots—building complex figures quickly becomes time-consuming. We have to write a lot of code to prepare even moderately complex plots, there are many graphical parameters to learn, and many of the standard plotting functions are inconsistent in how they work.

- Deepayan Sarkar developed the **lattice** package to implement the ideas of Bill Cleveland in his 1993 book, Visualizing Data. The package implements something called Trellis graphics, a very useful approach for graphical exploratory data analysis. Trellis graphics are designed to help us visualise complicated, multiple variable relationships. The **lattice** package has many "high level" functions to make this process easy. The **lattice** package is very powerful, but it is hard to master.

- Hadley Wickham developed the **ggplot2** package to implement the ideas in a book called The Grammar of Graphics by Wilkinson (2005). It produces Trellis-like graphics but is quite different from **lattice** in the way it goes about this. It uses its own mini-language to define graphical objects, adopting the language of Wilkinson's book to define these. It takes a little while to learn the basics, but once these have been mastered, it's very easy to produce sophisticated plots with very little R code. The downside of working with **ggplot2** is that it isn't as flexible as base graphics.

We are not going to survey all these plotting systems. It's entirely possible to meet most data visualisation needs by becoming proficient with just one of them. This book focuses on the **ggplot2** package. In many ways, **ggplot2** hits the 'sweet spot' between base graphics and **lattice**. It enables complex visualisations without the need to write many lines of R code, but remains flexible enough to allow the kind of customisation required to produce publication-quality figures.

13.1   The anatomy of ggplot2

The easiest way to learn **ggplot2** is by using it. However, before we dive into **ggplot2** code, we need to review the essential features of its 'grammar'—the rules of how to specify a graph. This grammar is fairly abstract and won't make much sense on first reading. That is fine. Ideas like 'aesthetic mappings' and 'geoms' will start to make sense as we work through various examples.

The design of **ggplot2** reflects Wilkinson's grammar of graphics. A complete **ggplot2** object is defined by a combination of:

1) one or more **layers**,
2) a set of **scales** (one for each 'aesthetic mapping'),
3) a **coordinate system** (one per plot),
4) a **facet** specification (if using a multi-panel plot).

The underlying idea is that we construct a visualisation by defining one or more layers. Each layer is associated with some data and a set of rules for how to display the data. Let's review these components before moving onto the business of actually using **ggplot2**.

13.1.1   Layers

Each layer in a **ggplot2** plot has five different components, though we don't necessarily have to specify all of these because most have some kind of default setting:

- The **data**. At a minimum, every plot needs some data. Unlike base R graphics, **ggplot2** always accepts data in one format, an R data frame (or tibble). Each layer can be associated with its own data set. We don't have to add data to each layer explicitly. If we choose not to specify the data set for a layer, **ggplot2** will use the default data (if defined).

- A set of **aesthetic mappings**. These describe how variables in the data are associated with the aesthetic properties of the layer. Aesthetic properties include things we perceive, such as position, colour, and size of the points. Each layer can be associated with its own unique aesthetic mappings. When we choose not to specify these for a layer **ggplot2** will use the defaults (if defined).

- A **geometric object** (a.k.a. 'geom'). The geom part of layer tells **ggplot2** how to represent the information—i.e. it refers to the objects we see on a plot, such as points, lines, bars or even text. Each geom only works with a particular subset of aesthetic mappings. We always have to define a geom when specifying a layer.

- A **statistical transformation** (a.k.a. 'stat'). A stat takes the raw data and transforms it in some way. A stat allows us to plot useful summaries of our raw data. We won't explicitly use them in this book because we prefer to produce summary figures by first processing the data with **dplyr**. Nonetheless, the stat facility is often doing useful work in the background for some kinds of plots.

- A **position adjustment**. These apply small tweaks to the position of layer elements. These are typically used when we need to define how the information for different categories are separated. For example, when making a bar plot, we may need to specify whether bars should be stacked on top of one another (the default) or plotted side-by-side.

13.1.2 Scales

The scale part of a **ggplot2** object controls how the information in a variable is mapped to the aesthetic properties. A scale takes the data and converts it into variation we can perceive, such as an x/y location or the colour and size of points in a plot. The two most important things to understand about scales are:

1. A scale must be defined for every aesthetic in a plot. It doesn't make sense to define an aesthetic mapping without a scale because there is no way for **ggplot2** to know how to go from the data to the aesthetics without one.
2. When we include two or more layers, they all have to use the same scale for any shared aesthetic mappings. This behaviour is necessary to ensure information is displayed consistently.

If we choose not to explicitly define a scale for an aesthetic **ggplot2** will use a default. This will often be a 'sensible' choice, which means we can get quite a long way with **ggplot2** without ever really understanding scales. We will take a brief look at a few of the more common options, though.

13.1.3 Coordinate system

A **ggplot2** coordinate system takes the position of objects (e.g. points and lines) and maps them onto the 2d plane a plot lives on. Most people are already very familiar with the most common coordinate system (even if they didn't realise it). That's the Cartesian coordinate system. This is the one we've all been using since we first constructed a graph with paper and pencil at school. All the most common statistical plots use this coordinate system, so we won't consider any others in this book.

13.1.4 Faceting

The idea behind faceting is very simple. Faceting allows us to break a data set up into subsets according to the unique values of one or more variables and then produce a separate plot for each subset. The result is a multi-panel plot where each panel shares the same layers, scales, etc. The data is the only thing that varies from panel to panel. The result is a kind of 'Trellis plot', similar to those produced by the **lattice** package. Faceting is a very powerful tool that allows us to slice up our data in different ways and understand the relationship between different variables.

13.2   A quick introduction to ggplot2

Now that we've briefly reviewed the **ggplot2** grammar, we can start learning how to use it. The package uses this grammar as the basis of a sort of mini-language within R. It uses functions to specify components like aesthetic mappings and geoms, which are combined with data to define a **ggplot2** graphics object. Once we've constructed a suitable object, we can use it to display our graphic on the computer screen or save it using a common graphics format (e.g. PDF, PNG or JPEG).

Rather than orientating this introduction around each of the key functions, we're going to develop a simple example to help us see how **ggplot2** works. Many of the key ideas about how **ggplot2** works can be taken away from this one example. Hence, it's worth investing the time to understand it—i.e. use the example to understand how the different **ggplot2** functions are related to the grammar outlined above.

Our goal is to produce a simple scatter plot. The scatter plot is one of the most commonly used visualisation tools in the EDA toolbox. A scatter plot uses horizontal and vertical positions (the 'x' and 'y' axes) to visualise pairs of related observations as a series of points in two dimensions. It's designed to show how one numeric variable is associated with another. We'll use the `penguins` data to construct the scatter plot. The questions we want to explore are:

- what is the relationship between bill depth and bill length, and
- how does this vary in relation to other variables?

13.2.1 Making a start

We will begin our work with **ggplot2** by setting up a minimal **graphical object**. That's a job for the `ggplot` function. Using `ggplot` without any arguments builds an empty graphical object:

```
# construct empty ggplot2 graphical object
plot_obj <- ggplot()
```

This constructs the skeleton object and assigns it a name, `plot_obj`. We can use the `summary` function to inspect the object to find out a bit more about it:

```
# print summary of empty graphical object
summary(plot_obj)
```

```
## data: [x]
## faceting: <ggproto object: Class FacetNull, Facet, gg>
##      compute_layout: function
##      draw_back: function
##      draw_front: function
##      draw_labels: function
##      draw_panels: function
##      finish_data: function
##      init_scales: function
##      map_data: function
##      params: list
##      setup_data: function
##      setup_params: function
##      shrink: TRUE
##      train_scales: function
##      vars: function
##      super:  <ggproto object: Class FacetNull, Facet, gg>
```

The output of `summary` is quite verbose, but the important parts are near the top, just before the `faceting:` section. In this case, the 'important part' is basically empty. All we did was set up an empty graphical object—there are no data, aesthetic mapping, layers, etc associated with `plot_obj`.

It is not necessary to inspect the insides of every **ggplot2** object with `summary`. However, it can be instructive to do this when first learning about the package.

> ### ggplot2 vs. ggplot
>
> Notice that the package is called **ggplot2**, but the actual function that does the work of setting up the graphical object is called `ggplot`. Try not to mix the names up—this is a common source of errors.

How can we improve on this? We could add a default data set. This is easy. We do it by passing the name of a data frame or tibble to `ggplot`. Let's try doing this with `penguins`:

```
# construct ggplot2 graphical object with data
plot_obj <- ggplot(penguins)
```

Then print out the summary of the updated `plot_obj`:

```
# print summary of updated graphical object
summary(plot_obj)
```

```
## data: species, island, bill_length_mm, bill_depth_mm,
##   flipper_length_mm, body_mass_g, sex, year [344x8]
## faceting: <ggproto object: Class FacetNull, Facet, gg>
##   ...facet summary suppressed...
```

We suppressed the facet information this time because it takes up a lot of space and we're not interested in it. The important point is that the `plot_obj` summary now contains some information related to the data. The variables from `penguins` now comprise the data associated with the `plot_obj` object.

The next step is to add a default aesthetic mapping to the graphical object. Remember, this describes how variables in the data are mapped to the aesthetic properties of the layer(s).

One way to think about aesthetic mappings is they define what kind of relationships the plot will describe. Since we're making a scatter plot, we need to define mappings for positions on the 'x' and 'y' axes. We want to investigate how bill depth depends on bill length, so we need to associate `bill_length_mm` with the x position and `bill_depth_mm` with the y position.

We define an aesthetic mapping with the `aes` function. One way to use `aes` is like this:

```
# add aesthetic mappings to graphical object
plot_obj <- plot_obj + aes(x = bill_length_mm, y = bill_depth_mm)
```

This little snippet of R code looks odd at first glance. There are a couple of things to take away from this:

1. We can 'add' the aesthetic mapping to the `plot_obj` object using the `+` operator. This has nothing to do with arithmetic. The **ggplot2** package uses some clever programming tricks to redefine the way `+` works so that it can be used to combine graphical objects. It takes a bit of getting used to but this is useful because it makes building a plot from the components of the grammar very natural.

2. The second thing to notice is that an aesthetic mapping is defined by one or more name-value pairs, specified as arguments of `aes`. The names on the left-hand side of each `=` refer to the properties of our graphical object (e.g. the 'x' and 'y' positions). The values on right-hand side refer to variable names in the data we want to associate with these properties.

Notice that we overwrote the original `plot_obj` object with the updated version using the assignment operator. We could have created a distinct object, but there's usually no need to do this.

Once again, we can inspect the result using `summary`:

```
# print summary of updated graphical object
summary(plot_obj)
```

```
## data: species, island, bill_length_mm, bill_depth_mm,
##   flipper_length_mm, body_mass_g, sex, year [344x8]
## mapping:  x = ~bill_length_mm, y = ~bill_depth_mm
## faceting: <ggproto object: Class FacetNull, Facet, gg>
##   ...facet summary suppressed...
```

The data (`data:`) from the original `plot_obj` are still there, but now we can also see that two default mappings (`mapping:`) have been defined for the x- and y-axis positions. We have successfully used the `ggplot` and `aes` functions to set up a graphical object with both default data and aesthetic mappings.

Any layers that we now add will use these data and mappings unless we choose to override them by specifying different options.

We now need to specify a layer to tell **ggplot2** how to visualise the data. Remember, each layer has five components: data, aesthetic mappings, a geom, a stat and a position adjustment. Since we have already set up the default data and aesthetic mappings, there's no need to define these again—**ggplot2** will use the defaults if we leave them out of the layer definition. This leaves the geom, stat and position adjustment.

What kind of geom do we need? A scatter plot allows us to explore a relationship as a series of points. That means we need to add a layer that uses the 'point' geom. Simple.

What about the stat and position? Not so simple. These are difficult to explain without drilling down into how **ggplot2** works. A key insight is that both the stat and the position adjustment change the data somehow before plotting it. When we want to stop **ggplot2** from doing anything to our data, the keyword is 'identity'. We use this value whenever we want **ggplot2** to plot the data without modification.

We will examine the easy way to add a layer in a moment. However, we'll use the long-winded approach first because this reveals what happens whenever we build a **ggplot2** object. The general function for adding a layer is simply called `layer`. Here's how it works in its most basic usage:

```
# add layer to graphical object -- using points geom
plot_obj <- plot_obj + layer(geom = "point", stat = "identity", position = "identity")
```

This adds a layer to the existing `plot_obj` object with the `layer` function and overwrites the old version. Again, we add the new component using the `+` symbol. We set three arguments of the `layer` function:

1. define the **geom**: the name of this argument was `geom` and the value assigned to it was `"point"`.
2. define the **stat**: the name of this argument was `stat` and the value assigned to it was `"identity"`.
3. define the **position adjustment** : the name of this argument was `position` and the value assigned to it was `"identity"`.

Let's review the structure of the resulting graphical object one last time to see what we've achieved:

```
# print summary of updated graphical object
summary(plot_obj)
```

```
## data: species, island, bill_length_mm, bill_depth_mm,
##   flipper_length_mm, body_mass_g, sex, year [344x8]
## mapping:  x = ~bill_length_mm, y = ~bill_depth_mm
## faceting: <ggproto object: Class FacetNull, Facet, gg>
##   ...facet summary suppressed...
## ----------------------------------
## geom_point: na.rm = FALSE
## stat_identity: na.rm = FALSE
## position_identity
```

The text above the `-----` line is the same as before. It summarises the default data and the aesthetic mapping. The text below it summarises the layer we just added. This tells us that layer uses the points geom (`geom_point`), the identity stat (`stat_identity`), and the identity position adjustment (`position_identity`).

Now `plot_obj` has everything it needs to render a figure. How do we do this? Simply 'print' the graphical object:

```
# 'print' graphical object to show it
print(plot_obj)
```

```
## Warning: Removed 2 rows containing missing values (geom_point).
```
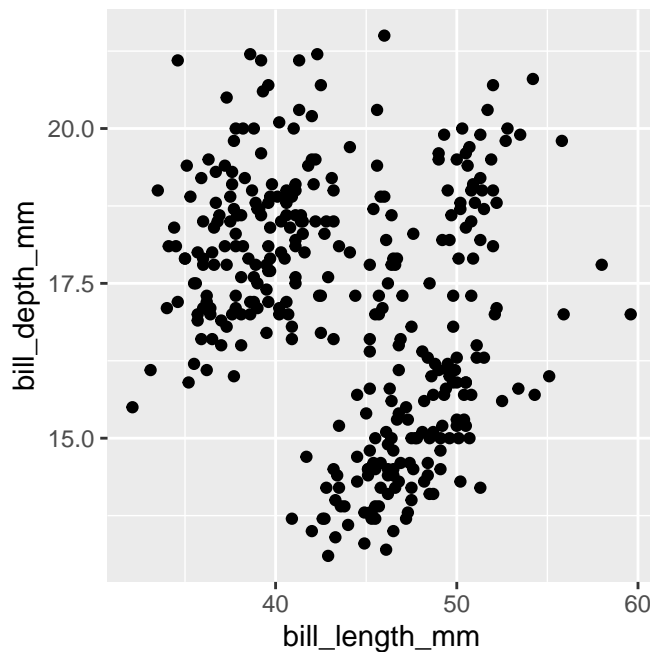
That's it! We have produced a scatter plot showing how bill depth is associated with bill length. It seems a little odd to show a plot by 'printing' it but that's just how **ggplot2** works. It soon starts to feel natural with practise.

Notice that **ggplot2** printed a warning. That is nothing to worry about. It is just letting us know a couple of rows in the data were ignored because they contained missing values. We will suppress that message from now on because it gets a bit irritating.

Here's a quick summary of what we did, all in one place:

```
# step 1. set up the skeleton graphical object with a default data set
plot_obj <- ggplot(penguins)
# step 2. add the default aesthetic mappings
plot_obj <- plot_obj + aes(x = bill_length_mm, y = bill_depth_mm)
# step 3. specify the layer we want to use
plot_obj <- plot_obj + layer(geom = "point", stat = "identity", position = "identity")
# step 4. show the plot
print(plot_obj)
```

⚠ **Don't use this workflow!**

It's possible to construct any **ggplot2** visualisation using the workflow outlined in this subsection, but **this is not the recommended approach**.

> The workflow we adopted here was used to reveal how the grammar works, rather than its efficiency. A more concise, standard approach to using **ggplot2** is outlined next. Use that for real-world analysis.

13.3   A standard way of using ggplot2

The **ggplot2** package is quite flexible, which means we can specify a visualisation in more than one way. To keep life simple, we're going to adopt a consistent workflow from now on. This won't reveal the full array of **ggplot2** tricks, but it is sufficient to construct a wide range of standard visualisations. Let's make the same bill depth vs. bill length scatter plot again to see the workflow in action.

We began building our **ggplot2** object by setting up a skeleton object with a default data set and then added the default aesthetic mappings. There is a more concise way to achieve the same result:

```
# create graphical object with data and aesthetic mappings
plot_obj <- ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm))
```

In this form, the `aes` function appears inside `ggplot` as a second argument. This sets up a graphical object with default data and aesthetic mappings in a single step. We will always use this approach from now on.

The next step adds a layer. We saw that the `layer` function could be used to construct one from its component parts. However, **ggplot2** provides many convenience functions that construct layers according to the type of geom they need. They all look like this: `geom_TYPE`, where `TYPE` stands for the name of the geom we want to use. For example, a point geom is specified using `geom_point`. Using this function, an alternative to the last line of the example is therefore:

```
# use geom_point to add a points layer to graphical object
plot_obj <- plot_obj + geom_point()
```

We didn't have to specify the stat or the position adjustment components of the layer because the `geom_TYPE` functions all use reasonable defaults. These can be overridden if needed, but most of the time, there's no need to do this. This way of defining a layer is much simpler and less error-prone than the manual `layer` method. We will always use the `geom_TYPE` method from now on.

There's one last trick we need to learn to use **ggplot2** efficiently. We've been building a plot object in several steps, giving the intermediates the name `plot_obj`, and then manually printing the object to display it when it's ready. This is useful if we want to make different versions of the same plot. However, we very often just want to build the plot and display it in one go. This is done by combining everything with `+` and printing the resulting object directly:

```r
# display bill morphology scatter plot **in one step**
ggplot(penguins,
       aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point()
```

That code builds the **ggplot2** graphical object and renders it in one go. We didn't even have to use `print` to generate the output. There is a lot is going on in the background, but this small snippet of R code contains everything **ggplot2** needs to construct and display a simple scatter plot.

What does this scatter plot actually tell us about penguin bill morphology? It seems to suggest there isn't much of a relationship between bill depth and bill length. Does that seem sensible? Maybe we need a more informative plot.

> **Plotting incomplete ggplot2 objects**
>
> It is instructive to see what happens when we render an incomplete graphical object. The four panels below show the output produced by a) an empty **ggplot2** object, b) a partially complete **ggplot2** object with only data, c) a partially complete **ggplot2** object with data and aesthetics, and d) a complete **ggplot2** object with data, aesthetics and a geom. The actual code to make each one is shown in the title area.

a) `ggplot()`

b) `ggplot(penguins)`

c) `ggplot(penguins,`
        `aes(x = bill_length_mm, y = bill_depth_mm))`

d) `ggplot(penguins,`
        `aes(x = bill_length_mm, y = bill_depth_mm)) +`
        `geom_point()`

We can see that **ggplot2** will try to plot something with the information
it has, but we only end up with an informative plot when we specify data,
aesthetic mappings, and a geom. That is the minimum we need to specify
to arrive at a useful plot.

13.3.1   How should we format ggplot2 code?

Take another look at that last example. We split the **ggplot2** definition over
two lines, placing each function on its own line. R doesn't care about those
extra newlines. As long as we put a + at the end of a line R will assume the
next line is part of the same plot definition. Splitting the different parts of a
graphical object definition across lines like this makes everything more readable
and helps us spot errors.

Once we split the definition across lines we can place comments between the
lines of **ggplot2** code. For example, we could add comments to state what the
`aes` and `geom_point` parts are doing:

```
# display bill morphology scatter plot
ggplot(penguins,
       # bill depth (y) vs bill length (x)
       aes(x = bill_length_mm, y = bill_depth_mm)) +
  # add points layer
  geom_point()
```

It's a very good idea to format and document **ggplot2** code in this way. That way, when we come back to it after a while, we can remember what we were trying to achieve! We will always use these conventions from now on.

13.4   Increasing the information density...

One of the great strengths of **ggplot2** is the ease with which it allows us to incorporate information from several variables into a single plot. So far, we have made a simple two-variable scatter plot to examine bill morphology—the bill depth-length relationship. There are clearly other variables in the `penguins` data set that might influence this relationship. In this section, we will highlight three approaches for including additional variables in a visualisation. That is, we will see how to increase the **information density** of a plot.

13.4.1   ...via aesthetic mappings

The most straightforward way to increase the information in a plot is by mapping a new variable to an unused aesthetic. For example, how might we learn whether the bill depth-length relationship varies by penguin species? We need to include information in the `species` variable in our scatter plot somehow. One option is to map the `species` to the colour aesthetic, so that the colour of the points correspond to different species. We do this by altering the `aes` part of the plot specification:

```
# display bill morphology scatter plot
ggplot(penguins,
       # bill depth (y) vs bill length (x) by species (colour)
       aes(x = bill_length_mm, y = bill_depth_mm, colour = species)) +
  # add points layer
  geom_point()
```
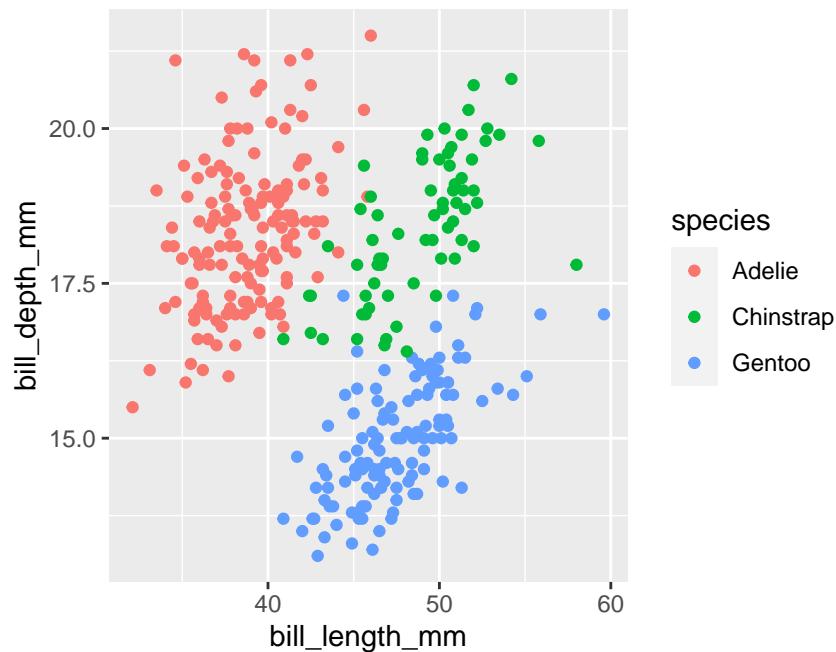
Individual points are now coloured according to the species they belong to. Notice **ggplot2** also adds a legend. This plot shows that bill morphology is reasonably species-specific. Separating things by species suggests a (mild) positive association between bill length and bill depth *within species*. This was not apparent before because it was hidden by the *among species* pattern.

We put the `aes` argument on a new line in this example because the first line was getting a bit long. This is only a readability thing—the `aes` part still belongs to `ggplot`.

We could certainly improve this visualisation. Nonetheless, it illustrates an important concept: we can add information to a plot by mapping additional variables to new aesthetics. There is nothing to stop us from using different aesthetics if we wanted to squeeze even more information into this plot. For example, we could map the sex variable (`sex`) to the point shape using `shape = sex` inside `aes`:

```
# display bill morphology scatter plot
ggplot(penguins,
       # bill depth (y) vs bill length (x) by species (colour) + sex (shape)
       aes(x = bill_length_mm, y = bill_depth_mm, colour = species, shape = sex)) +
  # add points layer
  geom_point()
```
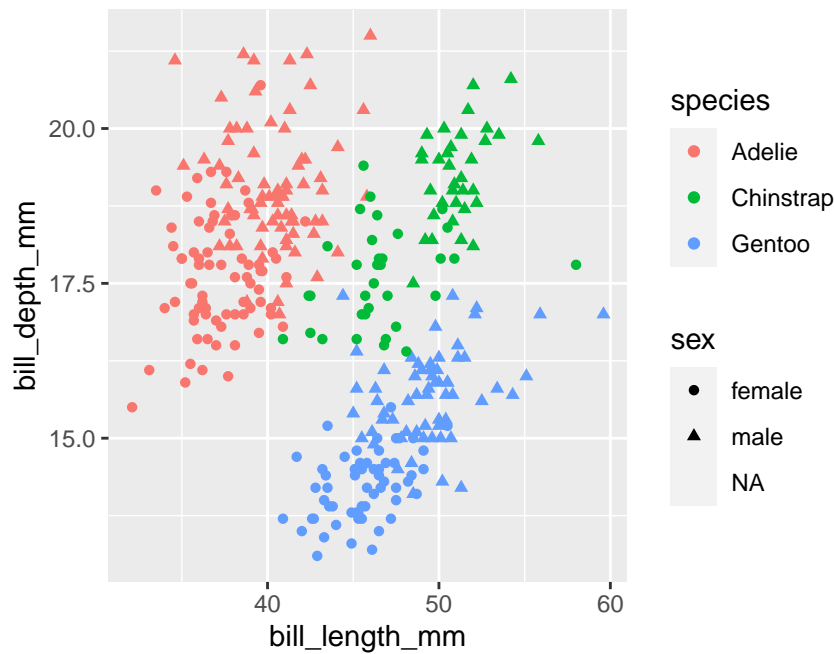
13.4.2 ...via facets

A second way to increase the amount of information on display is by making separate plots for meaningful subsets of the data. We can use the **faceting** facility of **ggplot2** to do this. Faceting allows us to define subsets of data according to the values of one or more variables and produce a separate plot for each subset, all without having to write much R code.

Faceting operates on the whole figure, which means we can't apply it by changing the properties of a layer. Instead, we have to use a new function to add the faceting information. There are two different ways to facet in **ggplot2**:

- `facet_wrap` forms a matrix of panels by wrapping the 1d sequence of panels into a 2d matrix with rows and columns. It is typically used with a single categorical faceting variable (though it works with two or more faceting variables).
- `facet_grid` forms a 2d matrix of panels defined by row and column variables. It is typically used when we have two or more categorical variables, and all combinations of the variables exist in the data.

This is best understood by example. What if we want to see how the bill morphology varies across islands? Here's how we split things up by `island` using the `facet_wrap` function:

```
# display facetted bill morphology scatter plot
ggplot(penguins,
       # species info via colour aesthetic
       aes(x = bill_length_mm, y = bill_depth_mm, colour = species)) +
  # add points layer
  geom_point() +
  # island info via facets
  facet_wrap(vars(island))
```
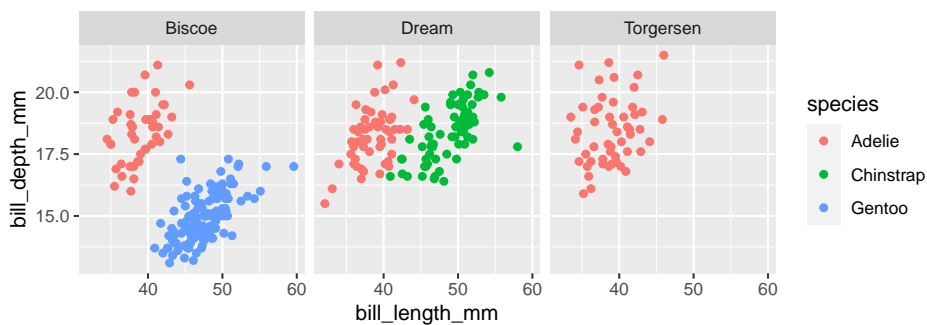

That first `vars(island)` argument of `facet_wrap` says to split up the data set according to the values of island. Simple. Notice that the panels share the same scales for the 'x' and 'y' axes. Using a common scale makes it easy to compare bill morphology across islands, but this can also be changed if necessary. For example, setting `scales = "free"` would ensure each panel uses its own x/y scale.

The plot indicates that bill morphology is roughly invariant across the three islands. It also inadvertently reveals something else about these penguins—the Gentoo and Chinstrap species are only found on a single island, which is different for each one, whereas Adelie penguins are found on all three islands. That's one reason exploratory analysis is so important—it throws up unexpected findings.

> ⚠ **Don't forget the `vars` part!**
>
> We have to wrap the name of the faceting variable with the `vars` function in the `facet_wrap` specification. Trust us, `facet_wrap` won't work without it.

13.4.3   ...via multiple layers

A third way to add additional information to a plot is by including multiple layers. So far, we have only seen examples with a single layer—only one `geom_` function is involved in creating the plot. There is no reason we can't add multiple

layers to a plot by 'adding' two or more `geom_` functions. We're not yet in a position to demonstrate how to make such multi-layer plots because they require additional skills covered in later chapters. However, as a taster, here is an example of the kind of thing we can do with this idea (we have deliberately hidden the code):
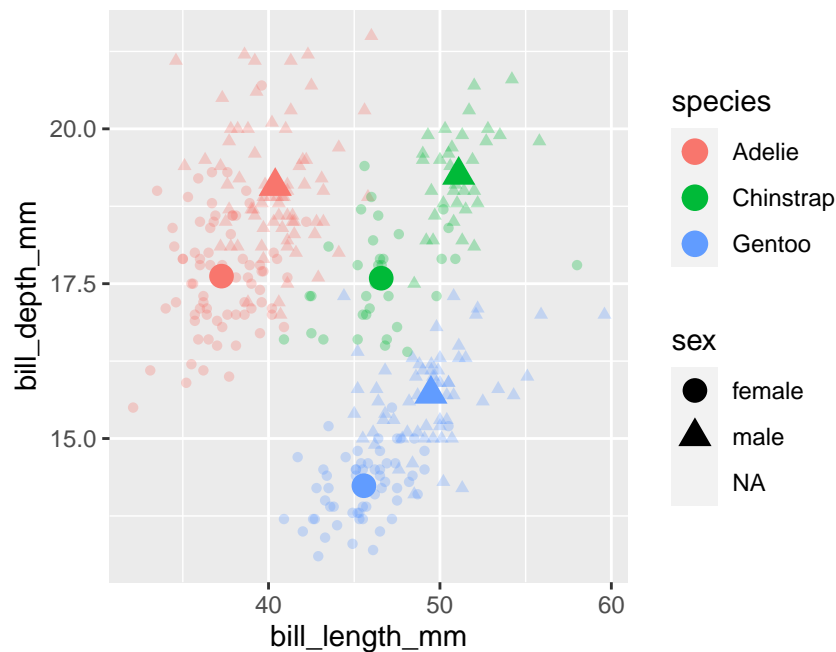
This shows how bill morphology varies by penguin species and sex, but now these relationships are summarised in two points layers. The first layer displays the raw data (small transparent points). The second layer adds the species/sex-specific means (large solid points). This places the average differences in the context of the overall variation in the raw data. Nice!

Chapter 14

Customising plots

The default formatting used by **ggplot2** has been carefully chosen to ensure the information in a plot is easy to discern. For example, by default, plots include a pale grey background and include grid lines. This is designed to emphasise the data while supporting comparisons. The grey highlights colour differences and ensures the grid lines have little visual impact beyond aiding in the assessment of position. Perfect for exploratory analysis!

Although we can justify these sorts of choices on the grounds they improve the readability of a plot, the fact remains they are somewhat unconventional and not much-loved by many users. For this reason, we often need to change the appearance of a plot before we include it in a document or presentation—e.g. most published figures use a white, rather than grey, background.

This chapter will demonstrate how to customise **ggplot2** plots. We are not going to cover every possible permutation. That would need its own book. Instead, we will explore the main routes to customisation so that we can use these as we review different visualisations later. Using the `penguins` data once again, we'll work on improving the following scatter plot from the Introduction to **ggplot2** chapter:

```r
# display bill morphology scatter plot
ggplot(penguins,
       # bill depth (y) vs bill length (x) by species (colour)
       aes(x = bill_length_mm, y = bill_depth_mm, colour = species)) +
  # add points layer
  geom_point()
```
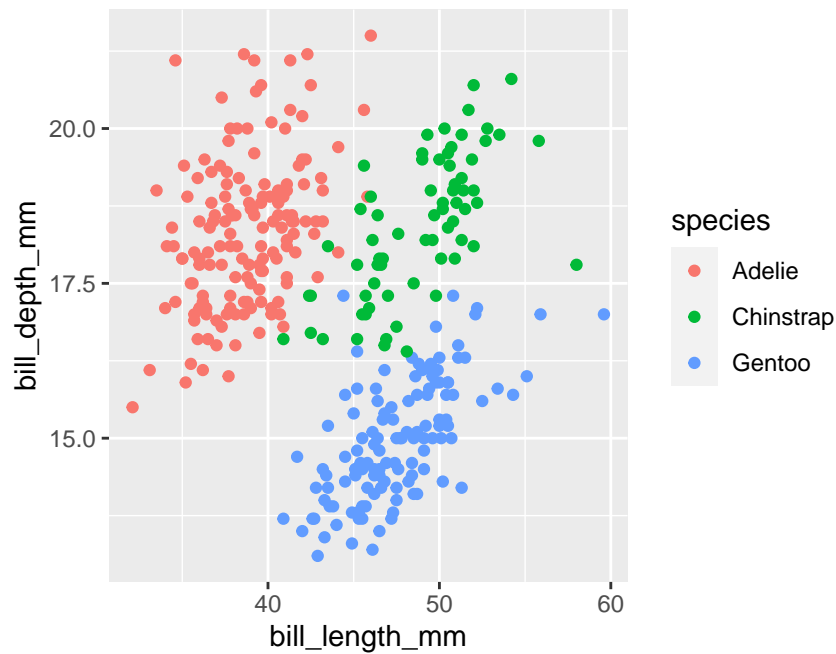
14.1   Geom properties

One common way to change a plot's appearance is by altering the properties of one or more geometric objects, such as points, lines, bars or polygons. How do we change the properties of the geom associated with a particular layer? We set those properties by specifying the appropriate arguments in the defining `geom_NAME` function.

For example, how might we change properties like the size or shape of points in our scatter plot? We used the `geom_point` function to add points to the scatter plot, which means we have to set the arguments of `geom_point` to change the point properties. We can use this idea to rebuild the example scatter plot, setting the shape, size and transparency of points:

```
# display bill morphology scatter plot
ggplot(penguins,
       # bill depth (y) vs bill length (x) by species (colour)
       aes(x = bill_length_mm, y = bill_depth_mm, colour = species)) +
  # add points layer and customise appearance of points
  geom_point(shape = 17, size = 1.5, alpha = 0.85)
```
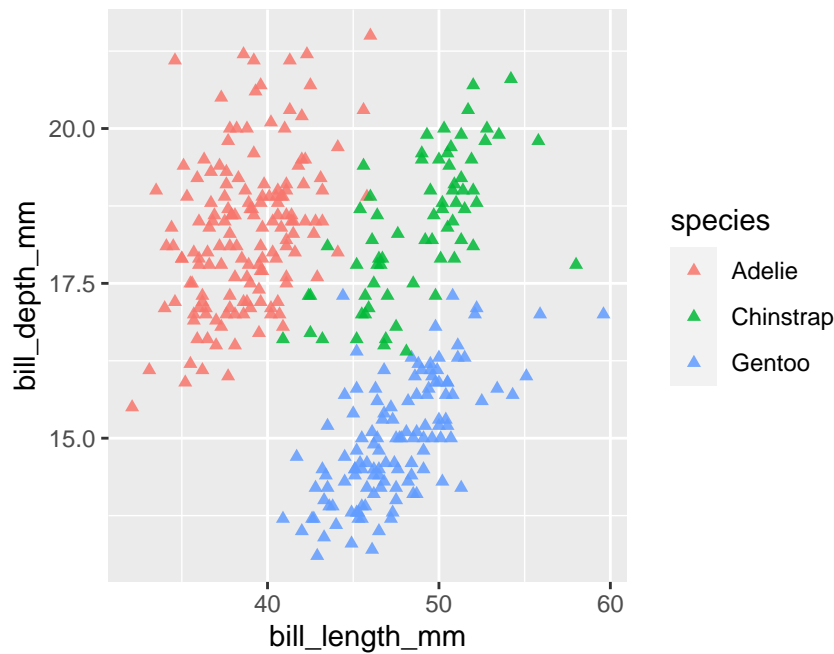
The point shape is set with the `shape` argument. There are a few different ways to specify point shapes in R. We used the numeric coding system in this example (17 = triangle). Unfortunately, the numeric codes aren't at all intuitive. Here are the more common ones:
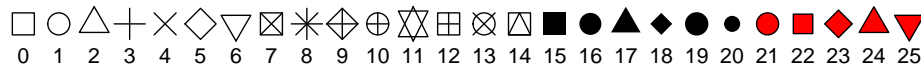
The point size is specified by the `size` argument. This has a baseline default value of 1. We assigned `size` a value of 1.5, thereby increasing the point size relative to the default. We could make the points smaller by using a value less than 1.

We made the points semi-transparent by setting the value of the `alpha` argument to be less than 1. In graphical systems, the 'alpha channel' specifies transparency—a value of 0 is taken to mean 'completely invisible', and a value of 1 means 'completely opaque'.

There are other arguments—such as `fill` and `colour`—that can also be used to adjust the way the points are rendered. We'll look at these later.

The key message to take away from this customisation example is this—if we need to alter the properties of a geometric object (points, lines, ect), we do so by specifying the appropriate arguments in the `geom_NAME` function that defines the layer it belongs to.

14.1.1   Relationship between aesthetic mappings and geom properties

In the previous chapter, we saw that we could introduce information into a plot by setting aesthetic mappings in the `aes` part of its specification. In the scatter plot example, we included information about species identity by mapping the `species` variable to the colour aesthetic. What happens if we also try to set the colour within the `geom_point` part?

```
# display bill morphology scatter plot
ggplot(penguins,
       # bill depth (y) vs bill length (x) by species (colour) + sex (shape)
       aes(x = bill_length_mm, y = bill_depth_mm, colour = species)) +
  # add points layer and customise appearance of points (including colour)
  geom_point(colour = "skyblue", shape = 17, size = 1.5, alpha = 0.85)
```
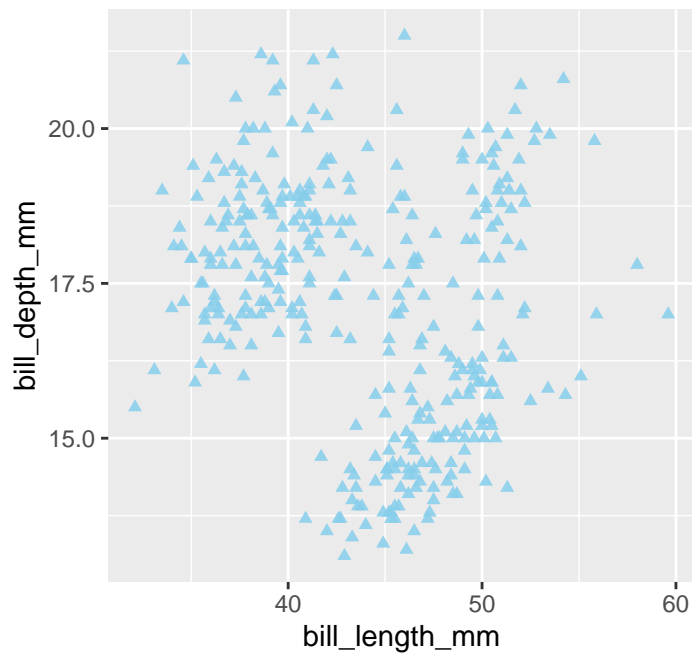

All the points are now one colour ('sky blue') which means we have lost the species information. What does this example demonstrate? When we set the properties of a geom, this will override any aesthetic mappings that conflict with our choice of customisation. Try to remember that—failing to do so is a good way to become frustrated.

> **ⓘ Built-in colours in R**
>
> There is nothing special about 'skyblue' other than the fact that it is colour name known to R. There are over 650 built-in colour names. To see them, use a function called `colours` to print them all to the Console. Here are 25 selected at random:
>
> ```
> sample(colours(), size = 25, replace = FALSE)
> ```
>
> ```
> ##  [1] "slateblue3"    "lavender"     "gray8"         "pink"
> ##  [5] "turquoise3"    "slategray3"   "hotpink"       "gray15"
> ##  [9] "tomato4"       "grey50"       "khaki1"        "grey65"
> ## [13] "darkslategray" "royalblue"    "gray29"        "grey64"
> ## [17] "grey6"         "palegreen3"   "lightyellow"   "firebrick1"
> ## [21] "gray81"        "darkorange1"  "darkorchid1"   "goldenrod1"
> ## [25] "violetred2"
> ```

14.2 Plot scales

Setting the arguments of a `geom_` function applies changes in a layer-specific manner. Other kinds of customisation affect every layer in a plot. Remember what we said about **scales** in the previous chapter:

> The scale part of a **ggplot2** object controls how the information in a variable is mapped to the aesthetic properties. A scale takes the data and converts it into variation we can perceive, such as an x/y location or the colour and size of points in a plot.

Every aesthetic mapping has a scale associated with it. We adjust 'how the information in a variable is mapped to the aesthetic properties' by changing its corresponding scale.

As always, this kind of thing is best understood by example. Let's adjust the scale associated with the y-axis aesthetic by increasing the number of the horizontal lines and their accompanying labels (the 'guides').
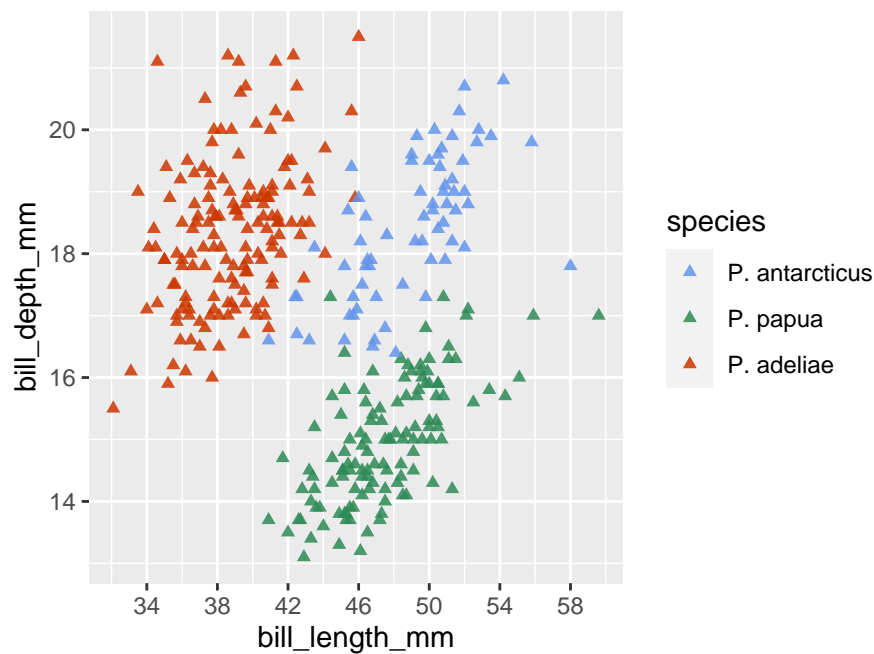
```
# display bill morphology scatter plot
ggplot(penguins,
       # bill depth (y) vs bill length (x) by species (colour)
       aes(x = bill_length_mm, y = bill_depth_mm, colour = species)) +
  # add points layer and customise appearance of points
  geom_point(shape = 17, size = 1.5, alpha = 0.85) +
  # customise y-axis grid line and label locations
  scale_y_continuous(breaks = seq(14, 20, by = 2))
```
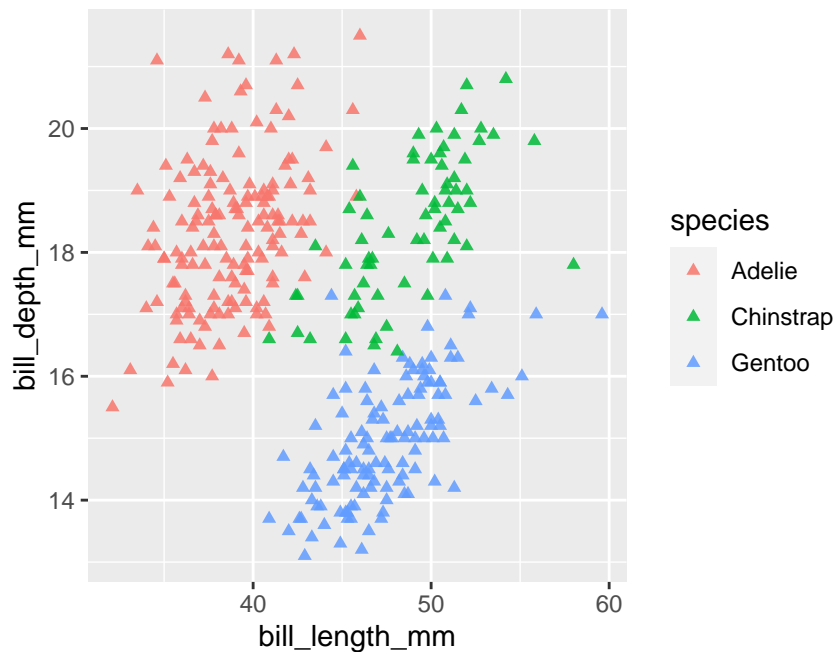
What's going on here?

- The aesthetic we wanted to alter was the y-axis position. The y-axis in this particular plot is associated with a continuous scale because `bill_depth_mm` is a numeric variable. This means we have to apply the `scale_y_continuous` function to tweak the position of the y-axis guides.

- The `breaks` argument of `scale_y_continuous` takes a numeric vector to specify where the guides should be drawn. We used the base R `seq` function to set up a numeric vector containing the sequence: 14, 16, 18 and 20. That tells **gglot2** to place the guides at 14, 16, 18 and 20 on the y-axis.

The functions that adjust a scale all have the general form `scale_AES_TYP`, where:

- the `AES` bit in the name refers to the relevant aesthetic, and
- the `TYP` part refers to the type of scale we want to apply.

Each kind of `scale_AES_TYP` function has its own set of arguments to control how the scale works. For example, the `breaks` argument of `scale_y_continuous` uses a numeric vector to specify where the guides should be drawn. We can do much fancier things if we want to. For example, the `trans` argument can be used to apply a particular transformation to an axis, such as a logarithm.

Predictably, there is also a `scale_x_continuous` function. We can use this in exactly the same way to control where the guides on the x-axis appear:

```
# display bill morphology scatter plot
ggplot(penguins,
       # bill depth (y) vs bill length (x) by species (colour)
       aes(x = bill_length_mm, y = bill_depth_mm, colour = species)) +
  # add points layer and customise appearance of points
  geom_point(shape = 17, size = 1.5, alpha = 0.85) +
  # customise x- and y-axis grid line and label locations
  scale_x_continuous(breaks = seq(34, 62, by = 4)) +
  scale_y_continuous(breaks = seq(14, 20, by = 2))
```
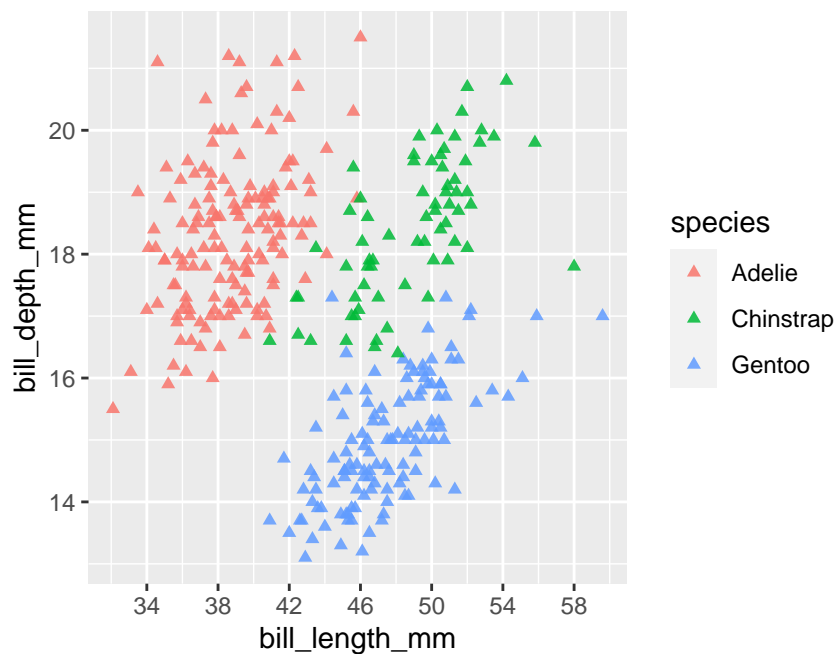

The key message to take away from this is that every aesthetic mapping has a scale associated with it. If we want to change how the information associated with an aesthetic mapping is displayed we have to change the corresponding scale. That suggests we can change the way point colours are associated with the `species` variable in our example by using one of the `scale_colour_YY` functions. That is correct:

```
# display bill morphology scatter plot
ggplot(penguins,
       # bill depth (y) vs bill length (x) by species (colour)
       aes(x = bill_length_mm, y = bill_depth_mm, colour = species)) +
  # add points layer and customise appearance of points
  geom_point(shape = 17, size = 1.5, alpha = 0.85) +
```

```
# customise x- and y-axis grid line and label locations
scale_x_continuous(breaks = seq(34, 62, by = 4)) +
scale_y_continuous(breaks = seq(14, 20, by = 2)) +
# customise the species colours ('value') and labels
scale_color_manual(
  limits = c("Chinstrap", "Gentoo", "Adelie"),
  values = c("cornflowerblue", "seagreen", "orangered3"),
  labels = c(Chinstrap = "P. antarcticus", Gentoo = "P. papua", Adelie = "P. adeliae"
)
```


This uses `scale_color_manual` to achieve three things:

- it defines the order to display the three species in the legend via `limits`,
- it sets up a custom colour palette via `values`, and
- it alters the category labels via `labels` (so that their scientific names are used).

Scales are one of the hardest aspects of **ggplot2** to master. There are many different scales, each controlled by its own unique set of arguments. Fortunately, the defaults used by **ggplot2** are often good enough to arrive at a reasonable plot without having to manipulate the scales.

14.3 Labels

What else might we like to tweak? Look at the x and y-axis labels. These are the names of the variables used in the x/y aesthetic mappings. These labels aren't too bad, but they could be nicer to look at. We know 'bill_depth_mm' stands for 'bill depth (measured in mm)', so why not use an axis label more like that?

The axes labels are a feature of the whole plot—i.e. they do not belong to a particular layer. This means we can't alter axis labels by passing arguments to the function that built a layer (`geom_point`). We have to use the labelling function: `labs`. This sets the label of any mapped aesthetics. For example, to set the labels of the x and y axes and the label associated with the colour mapping (i.e. the `species` legend title), use:

```r
# display bill morphology scatter plot
ggplot(penguins,
       # bill depth (y) vs bill length (x) by species (colour)
       aes(x = bill_length_mm, y = bill_depth_mm, colour = species)) +
  # add points layer and customise appearance of points
  geom_point(shape = 17, size = 1.5, alpha = 0.85) +
  # customise x- and y-axis grid line and label locations
  scale_x_continuous(breaks = seq(34, 62, by = 4)) +
  scale_y_continuous(breaks = seq(14, 20, by = 2)) +
  # customise the species colours ('value') and labels
  scale_color_manual(
    limits = c("Chinstrap", "Gentoo", "Adelie"),
    values = c("cornflowerblue", "seagreen", "orangered3"),
    labels = c(Chinstrap = "P. antarcticus", Gentoo = "P. papua", Adelie = "P. adeliae")
  ) +
  # set the the x/y-axis labels and legend label using `labs`
  labs(x = "Bill length (mm)", y = "Bill depth (mm)", colour = "Penguin Species")
```
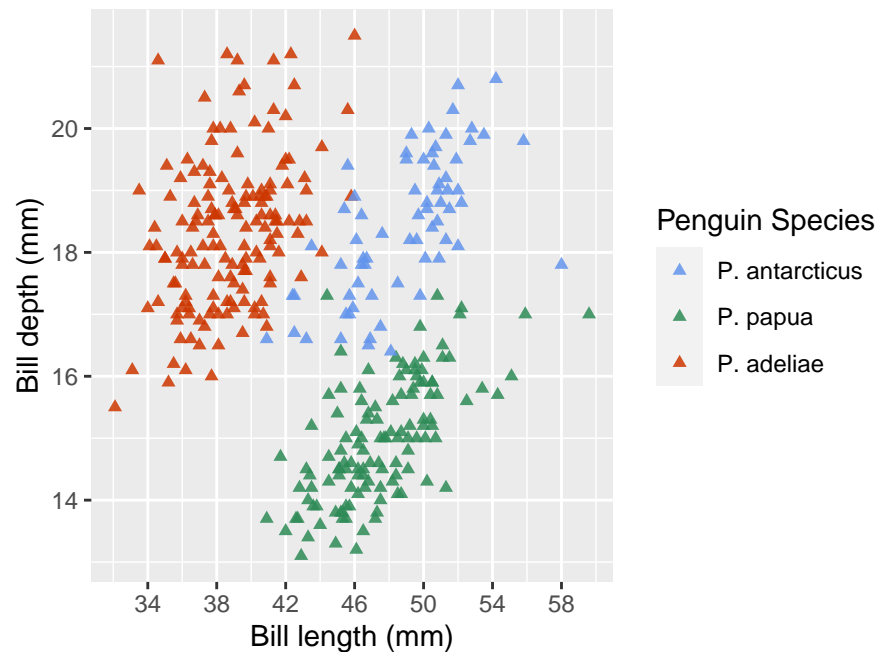
Using `labs` is easy—if a mapping was set up in `aes` we can change its label via a 'name = value' construct in `labs`. That's it.

Finally, another way to set the axis labels is via the `xlab` and `ylab`. We tend not to use these because `labs` deals with all our labelling needs on its own. Why waste effort remembering three things if you only need to remember one?

14.4   Themes

The final route to customisation happens via the 'theme' of a plot. We haven't considered the **ggplot2** theme system at all yet. A **ggplot2** theme deals with all the visual elements of a plot that aren't directly handled by adjusting geom properties or scales. These are essentially the 'non-data' parts of a plot—features such as the colour of the plotting region and the grid lines, whether or not those grid lines are even displayed, the position of labels, the font used in labels, and so on.

Here's a short example that adjusts the legend of the example plot:

```
# display bill morphology scatter plot
ggplot(penguins,
       # bill depth (y) vs bill length (x) by species (colour)
       aes(x = bill_length_mm, y = bill_depth_mm, colour = species)) +
  # add points layer and customise appearance of points
  geom_point(shape = 17, size = 1.5, alpha = 0.85) +
```

```r
# customise x- and y-axis grid line and label locations
scale_x_continuous(breaks = seq(34, 62, by = 4)) +
scale_y_continuous(breaks = seq(14, 20, by = 2)) +
# customise the species colours ('value') and labels
scale_color_manual(
  limits = c("Chinstrap", "Gentoo", "Adelie"),
  values = c("cornflowerblue", "seagreen", "orangered3"),
  labels = c(Chinstrap = "P. antarcticus", Gentoo = "P. papua", Adelie = "P. adeliae")
) +
# use `labs` to set x/y-axis labels and remove legend label
labs(x = "Bill length (mm)", y = "Bill depth (mm)", colour = "") +
# use theme to customise position and text formatting of the legend
theme(
  legend.position = "top",
  legend.text = element_text(face="italic")
)
```
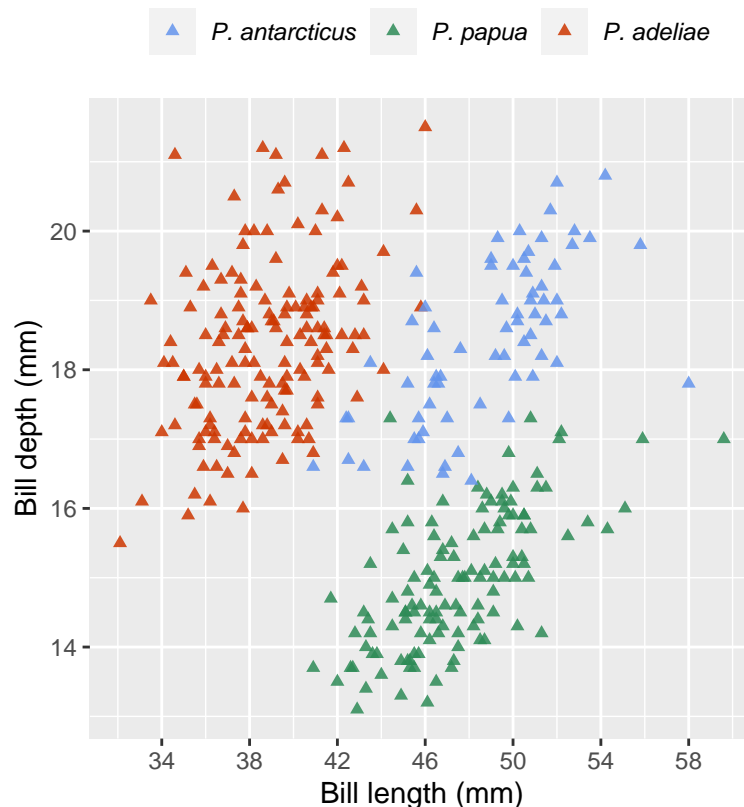

Changes to the theme are specified using... the `theme` function. We used `theme` to move the legend to the top of the plot (via `legend.position`) and show the

species labels in italics (via `legend.text`). We also got rid of the legend title
by setting the colour label to empty character string in `labs`. We could do that
via `theme` but using `labs` is simpler.

The **ggplot2** theme system is extremely powerful. Once we know how to use it,
we can set up a custom theme and apply it with very little effort. However, that's
not an entirely trivial thing to do because there are just so many components
of every plot. Most people just google whatever adjustment they want to make
when the time comes. Even then, tweaking each individual part of a theme
can be very time consuming. Fortunately, there are a range of standard themes
built into **ggplot2**. Here's how to apply a built-in theme:

```r
# display bill morphology scatter plot
ggplot(penguins,
       # bill depth (y) vs bill length (x) by species (colour)
       aes(x = bill_length_mm, y = bill_depth_mm, colour = species)) +
  # add points layer and customise appearance of points
  geom_point(shape = 17, size = 1.5, alpha = 0.85) +
  # customise x- and y-axis grid line and label locations
  scale_x_continuous(breaks = seq(34, 62, by = 4)) +
  scale_y_continuous(breaks = seq(14, 20, by = 2)) +
  # customise the species colours ('value') and labels
  scale_color_manual(
    values = c("cornflowerblue", "seagreen", "orangered3"),
    labels = c(Adelie = "P. adeliae", Chinstrap = "P. antarcticus", Gentoo = "P. papua
  ) +
  # use `labs` to set x/y-axis labels and remove legend label
  labs(x = "Bill length (mm)", y = "Bill depth (mm)", colour = "") +
  # apply 'black and white' theme
  theme_bw(base_size = 13) +
  # use theme to customise position and text formatting of the legend
  theme(
    legend.position = "top",
    legend.text = element_text(face="italic")
  )
```
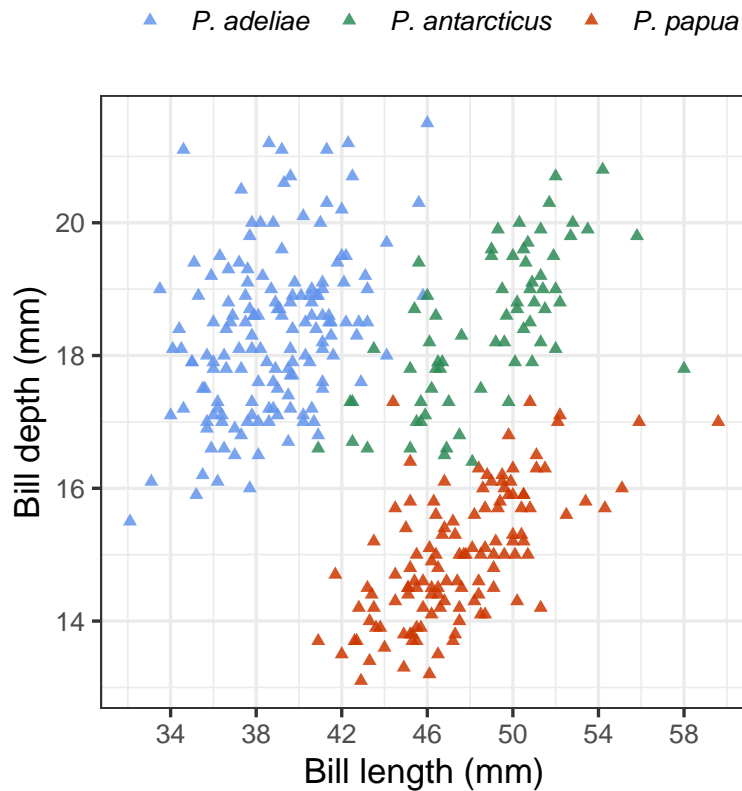
This uses + with the `theme_bw` function to add the built-in 'black and white' theme. This theme dispenses with the grey background that so many people dislike. There are a couple of things to note from this example:

- To apply the changes to the legend position, we had to place the `theme` part **after** we set the overall theme using `theme_bw`. The `theme_bw` function will override the `theme` changes if we try to do it the other way round.

- We set the `base_size` argument of `theme_bw`. This rescales the text in a plot, leaving the relative size of the different elements unchanged. We did not have to do this, but it is worth showing off because it is such a useful trick.

There aren't that many themes built into **ggplot2**. Here's a quick look at them:
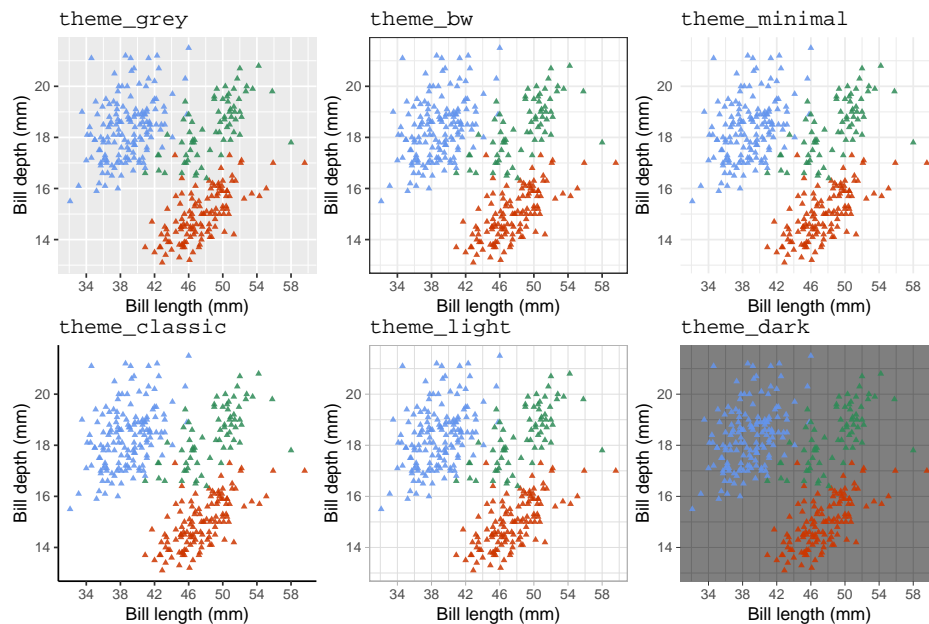
14.5   Advice for making plots

We'll end this section with a bit of advice about making your own plots with
ggplot2. Look at the code we ended up with for the final scatter plot above.
It's complicated! However, we didn't make that in one go. We built up to it
slowly, tweaking one thing at a time. That's the trick—**start with a basic
skeleton and gradually increase the level of customisation.** Begin by
getting the aesthetic mappings and the faceting correct so that all the relevant
variables are represented. Then introduce one bit of customisation and review
the plot to ensure it worked and everything looks right. Keep doing this, one
bit of customisation at a time, until the plot looks right.

Chapter 15

Exploring one variable

This chapter will consider how to go about exploring a single variable. Using the `penguins` data, we'll review some basic visualisations for exploring the sample distribution of numeric and categorical variables.

15.1 Exploring numerical variables

We'll work with flipper length (`flipper_length_mm`) and body mass (`body_mass_g`) to demonstrate how to explore numeric variables.

15.1.1 What kind of numeric variable?

The simplest way to explore data is to view it in its raw form. Take a quick look at the first 100 values of `flipper_length_mm` and `body_mass_g`. We can get these by extracting them with the `$` operator and using the `head` function:

flipper length —

```
head(penguins$flipper_length_mm, 50)
```

```
##  [1] 181 186 195  NA 193 190 181 195 193 190 186 180 182 191 198 185 195 197 184
## [20] 194 174 180 189 185 180 187 183 187 172 180 178 178 188 184 195 196 190 180
## [39] 181 184 182 195 186 196 185 190 182 179 190 191
```

body mass —

```
head(penguins$body_mass_g, 50)
```

```
##  [1] 3750 3800 3250   NA 3450 3650 3625 4675 3475 4250 3300 3700 3200 3800 4400
## [16] 3700 3450 4500 3325 4200 3400 3600 3800 3950 3800 3800 3550 3200 3150 3950
## [31] 3250 3900 3300 3900 3325 4150 3950 3550 3300 4650 3150 3900 3100 4400 3000
```

```
## [46] 4600 3425 2975 3450 4150
```

These are clearly both numeric variables, but we can say a bit more:

- They are both measured on a ratio scale. Zero really is zero, meaning that it makes sense to say that a length of 200 millimetres is twice as long as 100 millimetres or that 6000 grams is twice the mass of 3000 grams.
- They are continuous variables. Think about the possible values flipper length and body mass can take. A length of 200.52 millimetres and a mass of 3000.71 grams are both perfectly reasonable, so fundamentally, these are continuous variables.

Notice that even though body mass is a continuous variable `body_mass_g` looks discrete because body mass measurements have been taken to the nearest 25 grams. This reflects methodological limitations—presumably, weighing a live penguin to more than 25 grams precision in the field is challenging. This illustrates an important idea: we can't just look at the values a numeric variable takes to determine whether it is discrete or continuous.

Whether we treat a variable as continuous or discrete is sometimes an analysis decision. We have to decide based on knowledge of a variable's true nature and the measurement process. For example, if we were only able to measure body mass to the nearest 500 grams, we would only "see" a few different body mass categories. It might then be reasonable to treat the `body_mass_g` variable as an ordinal, categorical variable.

15.1.2   Histograms

We only looked at the first 50 values of the `flipper_length_mm` and `body_mass_g` variables because the `penguins` data set is a bit too big to view everything at once easily. It's hard to say much about the sample distribution of a numeric variable by looking at the raw data because there are so many unique values. Those first 50 are also not representative of the whole sample.

What else might we do? One useful tool is 'binning'. The idea behind binning a numeric variable is very simple. It involves two steps.

1. Divide the number line into equal-sized, non-overlapping intervals. These are the 'bins'. We pick the interval size to ensure bins (typically) include at least a few cases.
2. We then have to count up the number of times the variable falls inside each bin. The resulting set of counts summarises the sample distribution of the numeric variable.

Binning is very tedious to do by hand. Fortunately, R can do this for us. This shows how to use the `cut` function with the **dplyr transmute** and `count` verbs to bin the `flipper_length_mm` variable into 5 mm intervals:

```
penguins %>%
  transmute(bins = cut(flipper_length_mm, breaks = seq(170, 235, by = 5))) %>%
  count(bins)
```

```
## # A tibble: 14 x 2
##    bins          n
##    <fct>     <int>
##  1 (170,175]     2
##  2 (175,180]    11
##  3 (180,185]    28
##  4 (185,190]    58
##  5 (190,195]    57
##  6 (195,200]    38
##  7 (200,205]    18
##  8 (205,210]    30
##  9 (210,215]    33
## 10 (215,220]    32
## 11 (220,225]    20
## 12 (225,230]    14
## 13 (230,235]     1
## 14 <NA>          2
```

We won't explain how this works because we only need to understand the output. This is a small tibble where the `bins` column shows the intervals, and the `n` column gives the associated counts.

What does this tell us? It shows that the most common flipper length is around 190 mm. Values much below 190 mm are rare, but a range of values above this are possible, with higher values becoming less frequent. These binned data are telling us a lot about the sample distribution of `flipper_length_mm`.

It is still difficult to perceive distributional information when presented as a series of numbers. What we need is a visualisation based on the above numbers. This is what a **histogram** provides. Histograms summarise the sample distribution of a variable by showing the counts of binned data as a series of bars. The position and width of each bar correspond to an interval and the height shows the count. Here is a histogram based on the binned data we made:
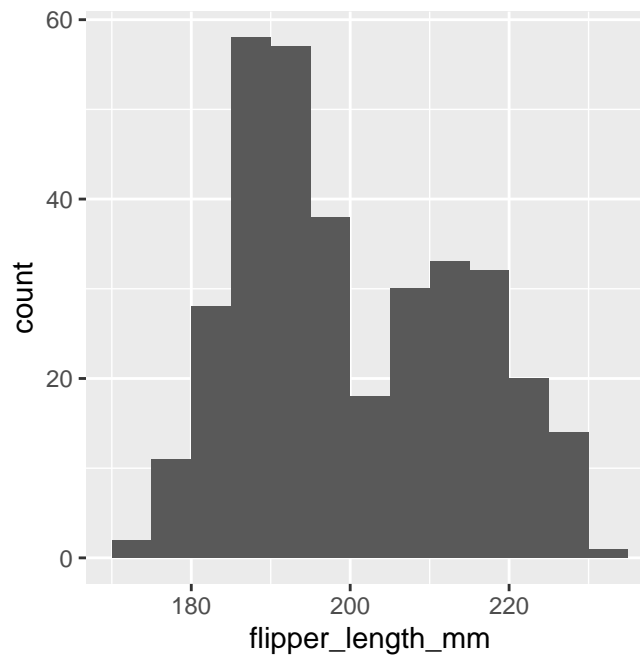
This gives a clear summary of the sample distribution of body mass. It reveals:

1. the most common values, which are round about 190 mm;
2. the range of the data, which is about 60 mm; and
3. the shape of the distribution, which appears to be **bimodal**—it has two peaks.

That last observation wasn't immediately obvious when looking at the raw counts in the binned summary.

We used **ggplot2** to make that histogram. We could have used **dplyr** to make the binned data set and then used this with **ggplot2** to construct the plot manually. However, there is a much easier way to make a histogram. We'll demonstrate this now. Rather than do it using one **ggplot2** expression, we will break the process up to demonstrate how it works.

The first step uses the `ggplot` function with `aes` to set up the default data and aesthetic mapping:

```
# create graphical object with data and aesthetic mapping
plot_hist <- ggplot(penguins, aes(x = flipper_length_mm))
```

This is no different than the scatter plot example we stepped through. The only difference is that a histogram requires only one aesthetic mapping. We supplied the argument `x = flipper_length_mm` to `aes` because we want to display the map intervals associated with `flipper_length_mm` to the x-axis. We don't need

to supply an aesthetic mapping for the y-axis because **ggplot2** is going to handle this for us.

The second step adds a layer to the `plot_hist` object. We need to find the right `geom_XX` function to do this. This is called `geom_histogram`:

```
# use geom_histogram to add histogram layer
plot_hist <- plot_hist + geom_histogram()
```

It is instructive to look at the summary of the **ggplot2** object at this stage:

```
# print summary of graphical object
summary(plot_hist)
```

```
## data: species, island, bill_length_mm, bill_depth_mm,
##   flipper_length_mm, body_mass_g, sex, year [344x8]
## mapping:  x = ~flipper_length_mm
## faceting: <ggproto object: Class FacetNull, Facet, gg>
##     compute_layout: function
##     draw_back: function
##     draw_front: function
##     draw_labels: function
##     draw_panels: function
##     finish_data: function
##     init_scales: function
##     map_data: function
##     params: list
##     setup_data: function
##     setup_params: function
##     shrink: TRUE
##     train_scales: function
##     vars: function
##     super:  <ggproto object: Class FacetNull, Facet, gg>
## ----------------------------------
## geom_bar: na.rm = FALSE, orientation = NA
## stat_bin: binwidth = NULL, bins = NULL, na.rm = FALSE, orientation = NA, pad = FALSE
## position_stack
```

Look at the text below the `----`. This shows that `geom_histogram` adds a 'stat' to the layer: `stat_bin`. This means **ggplot2** is going to take the raw `flipper_length_mm` data and bin it for us. Remember—the stat facility of **ggplot2** is its mechanism for creating summaries of data from the raw inputs. `stat_bin` deals with the binning process for us.

Everything we need to plot a histogram is now set up. Here is the resulting plot:

```
# show the histogram
print(plot_hist)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```
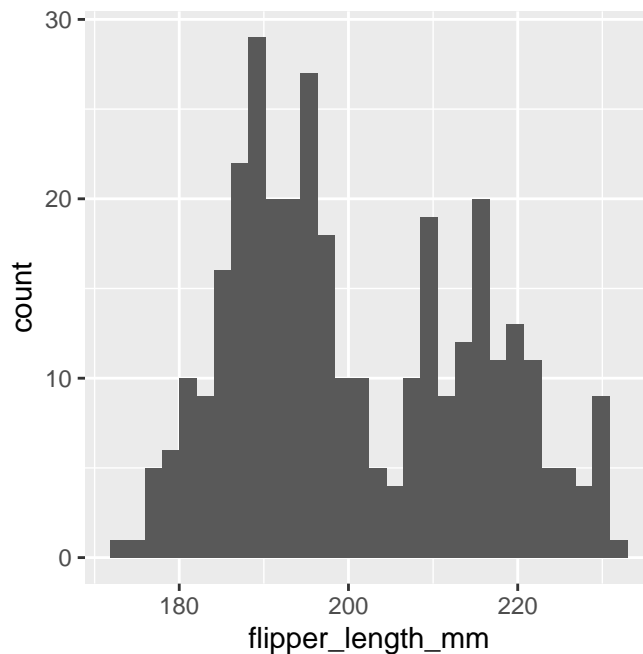

This is not quite the same as the example we saw above because it uses different bins. We can set the properties of the `geom_histogram` to tweak this kind of thing—the `binwidth` argument adjusts the width of the bins used. Let's construct the histogram again with 5 mm wide bins, as well as adjust the colour scheme and axis labels a bit:

```
# display flipper length histogram
ggplot(penguins, aes(x = flipper_length_mm)) +
  # use geom_histogram to add histogram layer + customisation
  geom_histogram(binwidth = 5, fill = "steelblue", colour="darkgrey") +
  # set the the x- and y-axis labels
  labs(x = "Flipper length (mm)", y = "Count")
```
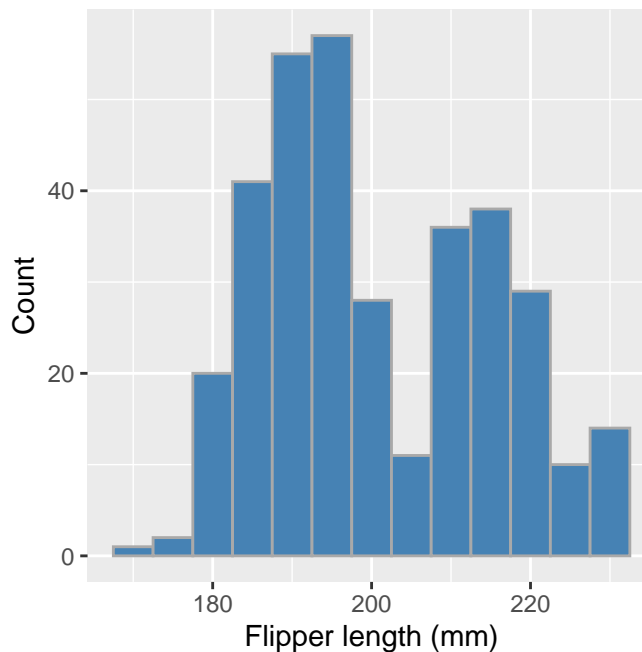
Notice that the effect of increasing the bin width is to 'smooth' the histogram, i.e. this version looks less jagged than the last. Whether or not that colour scheme is an improvement is a matter of taste. Mostly we wanted to demonstrate how the `fill` and `colour` arguments change the output— `fill` sets the fill colour of the bar and `colour` deals with the line around each bar.

⚠ **Choose your own bin widths**

It is good practice to experiment with `binwidth` because the value selected by **ggplot2** is seldom optimal. Finding a 'good' value is as much art as science. One way of doing this is to start with a value that is probably too large and then refine it down. For example, we could work out the approximate range of the data by eye and then use $1/10^{th}$ of that range as an initial value for `binwidth`. Unless the data set is very small, this will result in a histogram that aggregates too much information. Once we have that rough starting value, we then reduce `binwidth` in small steps, re-rendering the plot each time until we end up with something that summarises the distribution well.

We can use pretty much the same R code to produce a histogram summarising the body mass sample distribution:

```r
# display body mass histogram
ggplot(penguins, aes(x = body_mass_g)) +
```

```
# use geom_histogram to add histogram layer + customisation
geom_histogram(binwidth = 250, fill = "steelblue", colour="darkgrey") +
# set the the x- and y-axis labels
labs(x = "Flipper length (mm)", y = "Count")
```
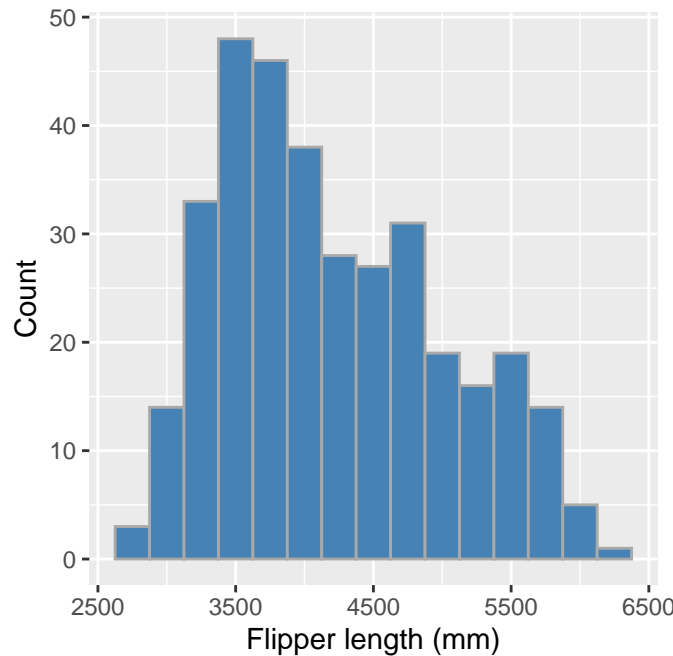

The only things that changed in this example were the aesthetic mapping and
the bin width, which we set to 250 to reflect the range of observed body masses.
The histogram reveals that the most common body mass (the mode) is around
3500 grams and the range of masses is also about 3500 grams. It also shows
that the distribution is **skewed**, i.e. it is asymmetric. We can say even more—
the distribution is right-skewed (a.k.a. positive-skewed) because the tail on the
right gets further away from the mode than the left tail.

We have to choose those bin widths carefully. Body mass is measured to the
nearest 25 grams. This means we should choose a bin width that is some multiple
of 25 to produce a meaningful histogram. Look what happens if we set the bin
width to 25:

```
# display body mass histogram
ggplot(penguins, aes(x = body_mass_g)) +
  # use geom_histogram to add histogram layer + customisation -- choose
  # a sensible bin width
  geom_histogram(binwidth = 25, fill = "steelblue", colour="darkgrey") +
  # set the the x- and y-axis labels
```

```
labs(x = "Flipper length (mm)", y = "Count")
```
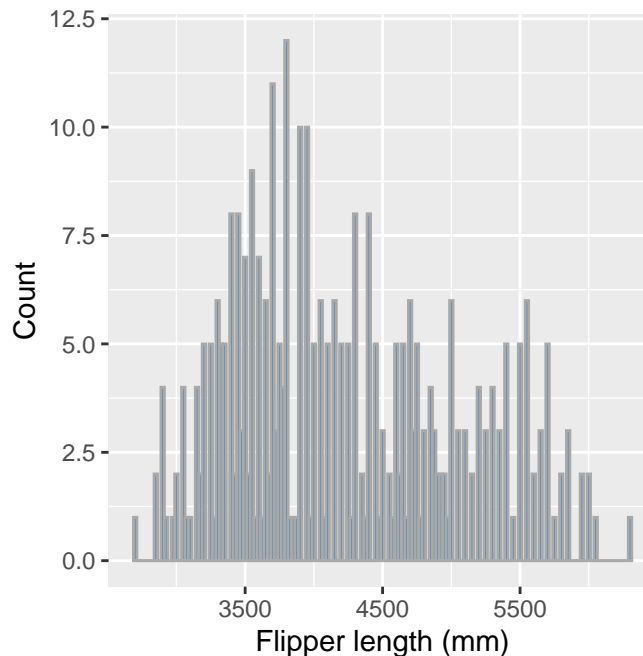

We end up with gaps in the histogram because many bins do not include any data. The take-home message is that we have to pay attention to our data to produce meaningful summaries.

15.1.3 Dot plots

Histograms are good for visualising sample distributions when we have a reasonably large sample size, e.g. >100 observations. They are less effective when a sample is small. In this 'small data' situation, it is better to use something called a **dot plot**[1].

We have created subset of **penguins** that only retains female Adelie penguin observations from 2007, called **penguins_small**. This is considerably smaller than the original:

```
## # A tibble: 22 x 8
##    species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##    <chr>   <chr>              <dbl>         <dbl>             <int>       <int>
## 1 Adelie  Torgersen           39.5          17.4               186        3800
```

---

[1]Not to be confused with the 'Cleveland dot plot'. A standard dot plot summarises a sample distribution. The Cleveland dot plot is something quite different, which summarises the frequencies of a categorical variable. It's meant to serve as a simple alternative to bar charts and pie charts.

```
##  2 Adelie  Torgersen         40.3        18              195          3250
##  3 Adelie  Torgersen         36.7        19.3            193          3450
##  4 Adelie  Torgersen         38.9        17.8            181          3625
##  5 Adelie  Torgersen         41.1        17.6            182          3200
##  6 Adelie  Torgersen         36.6        17.8            185          3700
##  7 Adelie  Torgersen         38.7        19              195          3450
##  8 Adelie  Torgersen         34.4        18.4            184          3325
##  9 Adelie  Biscoe            37.8        18.3            174          3400
## 10 Adelie  Biscoe            35.9        19.2            189          3800
## # ... with 12 more rows, and 2 more variables: sex <chr>, year <int>
```

This is about the right size for using a dot plot. Constructing a dot plot with **ggplot2** is easy once you understand the histogram case. The code is are very similar:

```
# display body mass dot plot
ggplot(penguins_small, aes(x = body_mass_g)) +
  # use geom_dotplot to add layer + choose a sensible bin width!
  geom_dotplot(binwidth = 100) +
  # set the the x-axis label and remove y-axis label
  labs(x = "Body mass (g)", y = "") +
  # remove y-axis guides
  theme(axis.text.y = element_blank(), axis.ticks.y = element_blank())
```
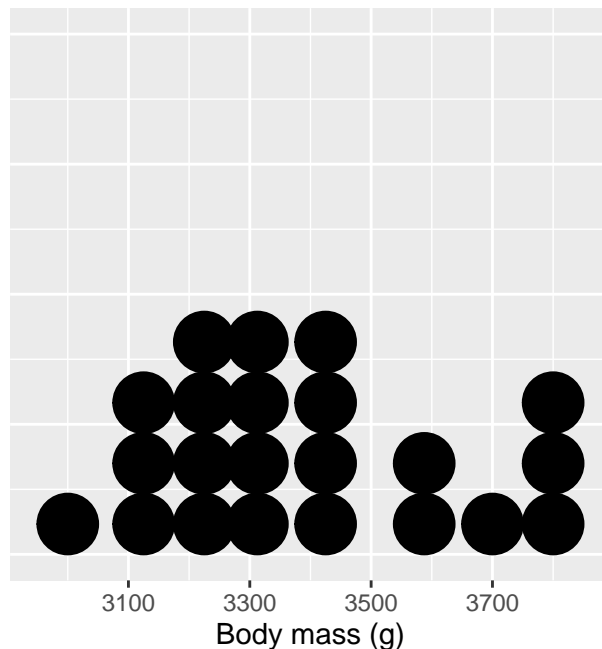

Each observation in the data adds one dot, and dots that fall into the same bin

are stacked up on top of one another. Notice the bins are not evenly spaced, however. The resulting plot displays the same information as a histogram but tends to be more informative when there are few observations.

The `theme` part of that example is entirely optional.It removes the grid lines and y-axis labels because these are not meaningful when the number of stacked dots represents ' height'. It is unfortunate that **ggplot2** displays them. But then, no package is perfect.

15.2 Exploring categorical variables

We will work with the penguin species (`species`) and observation year (`year`) variables to demonstrate how to explore categorical variables.

15.2.1 What kind of categorical variable?

Exploring categorical variables is generally simpler than the numeric case. The simplest thing we can do is examine the set of categories. For example, we can print these by extracting a variable with the `$` operator and using the `unique` function:

species —

```
unique(penguins$species)
```

```
## [1] "Adelie"    "Gentoo"    "Chinstrap"
```

observation year —

```
unique(penguins$year)
```

```
## [1] 2007 2008 2009
```

These are obviously categorical—there are no numeric values associated with either variable. The obvious question to ask about each variable is, is it ordinal or nominal?

It is probably safe to assume that `species` should be treated as a nominal variable. There is no reasonable way to order the three species—i.e. a statement such as 'Adelie > Gentoo' is nonsense.

What about `year`? That's a bit more tricky. If we were interested in how some aspect of the data changes over time, we might consider treating year as a numeric variable, or perhaps, as an ordinal categorical variable. Alternatively, if the question is simply, 'do the data vary from one year to the next' without any concern for trends, it's reasonable to treat year as a nominal categorical variable.

This once again illustrates an important idea—the classification of a variable sometimes depends on the analysis goals. The decision matters because it influences how we choose to summarise a variable, how we interpret its relationship with other variables, and whether a specific statistical model is appropriate for our data or not.

For now, let's assume it's fine to treat `year` as a nominal variable.

15.2.2   Bar plots

We can't say anything about the sample distribution of a categorical variable by just looking at the possible categories. We need to explore the relative frequency of those categories. There is no need to carry out anything as complicated as binning. Instead, we simply count up the number of times each category occurs in the data. The **dplyr count** verb will does this:

```
penguins %>% count(species)
```

```
## # A tibble: 3 x 2
##   species       n
##   <chr>     <int>
## 1 Adelie      152
## 2 Chinstrap    68
## 3 Gentoo      124
```

We saw this result earlier in the book—the most common species is Adelie, followed by the Gentoo and Chinstrap. Gentoo is roughly as common as Adelie, whereas Chinstrap is about half as frequent.

These numbers lead directly to an informative visualisation. The usual graphical tool for summarising categorical variables is the **bar chart**. A bar chart presents summaries of grouped data with rectangular bars. The lengths of each bar is proportional to the value it represents. When summarising categorical variables, the bar lengths show the raw counts or proportions of each category.

Constructing a bar graph to display category counts is easy with **ggplot2**. We'll do this for the `species` variable. As always, we start by using the `ggplot` function to construct a graphical object containing the default data and aesthetic mapping.

```
# create graphical object with data and aesthetic mapping
plot_bar <- ggplot(penguins, aes(x = species))
```

We've called the object `plot_bar` for the obvious reason. Notice that we only need to define one aesthetic mapping—we mapped `species` to the x-axis. This will produce a bar plot with vertical bars.

The next step is to add a layer using one of the `geom_XX` functions. There are two

functions we can use to create bar charts in ggplot, `geom_bar` and `geom_col`. By default, `geom_col` counts the number of observations in each category, whereas `geom_bar` plots the actual numbers we provide.

We want to show the counts of each `species` category, meaning we should use `geom_bar`:

```
# use geom_bar to add bar plot layer
plot_bar <- plot_bar + geom_bar()
```

Now examine the summary of the **ggplot2** object:

```
# print summary of graphical object
summary(plot_bar)
```

```
## data: species, island, bill_length_mm, bill_depth_mm,
##   flipper_length_mm, body_mass_g, sex, year [344x8]
## mapping:  x = ~species
## faceting: <ggproto object: Class FacetNull, Facet, gg>
##     compute_layout: function
##     draw_back: function
##     draw_front: function
##     draw_labels: function
##     draw_panels: function
##     finish_data: function
##     init_scales: function
##     map_data: function
##     params: list
##     setup_data: function
##     setup_params: function
##     shrink: TRUE
##     train_scales: function
##     vars: function
##     super:  <ggproto object: Class FacetNull, Facet, gg>
## --------------------------------
## geom_bar: width = NULL, na.rm = FALSE, orientation = NA
## stat_count: width = NULL, na.rm = FALSE, orientation = NA
## position_stack
```

Look at the layer information below ----. The `geom_bar` function uses `stat_count`. This means **ggplot2** summed the number of observations associated with each category of `species`. Counting a categorical variable is analogous to binning a numeric variable. The only difference is that there is no need to specify bin widths because `species` is categorical. Here's the resulting figure:

```
# show the bar plot
print(plot_bar)
```
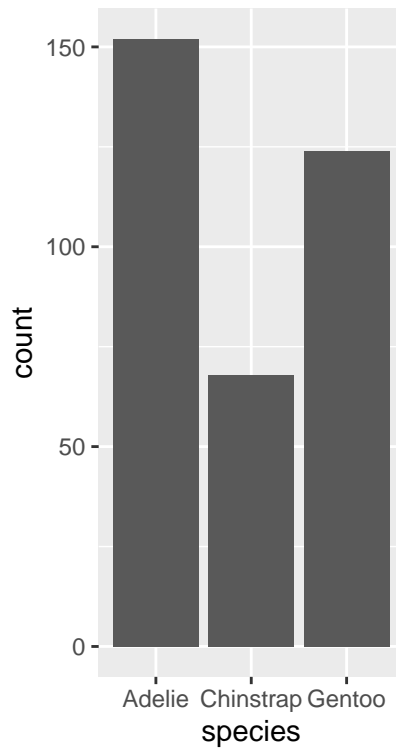

This is exactly the same information produced by the `count` function, only now it's presented in graphical form. We can customise this bar graph with functions like `labs` and by setting various properties inside `geom_bar`. For example:

```
# display bar plot of species counts
ggplot(penguins, aes(x = species)) +
  # use geom_bar to add bar plot layer of **counts**
  geom_bar(fill = "rosybrown4", width = 0.7) +
  # set the the x- and y-axis labels
  labs(x = "Species", y = "Number of Observations")
```
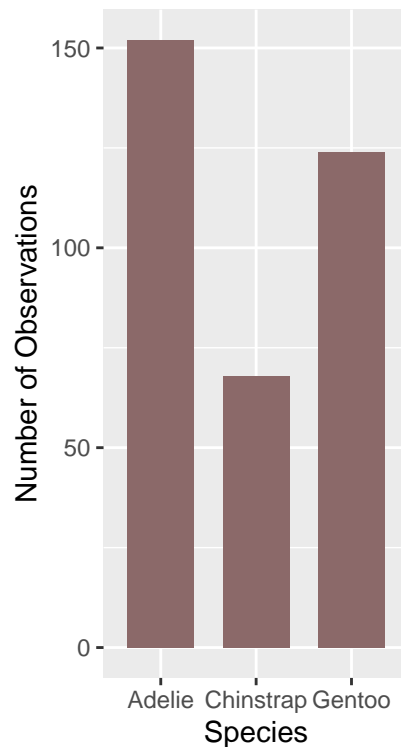
The only new thing here is that we used the `width` argument of `geom_bar` to make the bars a little narrower than the default. Notice that we use `fill` (not `colour`) to change the colour of the bars.

What might we like to change about this plot? The different species appear in alphabetical order. There's nothing particularly meaningful about that order. Perhaps it would make more sense to show them in frequency order from least to most common?

We need to customise the scale associated with the 'x' aesthetic to achieve this. We can start by making a short character vector containing the species names, listing them in the order they need to be shown:

```
# character vector of species names **in required order**
ords <- c("Chinstrap", "Gentoo", "Adelie")
```

Keep an eye on the spelling—R is not forgiving of spelling errors. We can use this with the `limits` argument of the `scale_x_discrete` function to adjust the ordering:

```
# display bar plot of species counts
ggplot(penguins, aes(x = species)) +
```

```
# use geom_bar to add bar plot layer of **counts**
geom_bar(fill = "rosybrown4", width = 0.7) +
# customise order of x-axis (limits)
scale_x_discrete(limits = ords) +
# set the the x- and y-axis labels
labs(x = "Species", y = "Number of Observations")
```
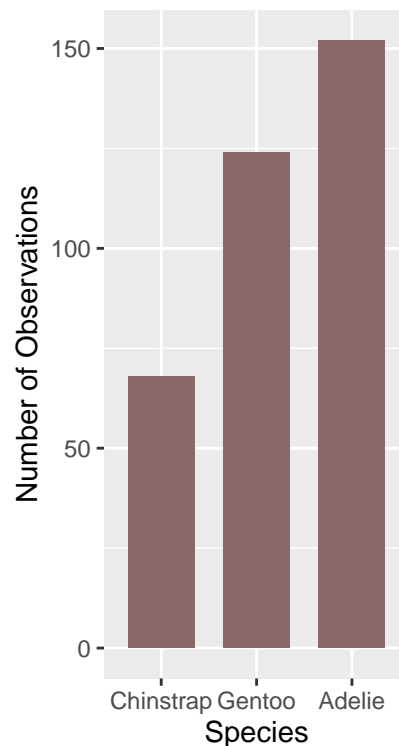

We had to use one of the `scale_x_YY` functions to change the way the 'x' aesthetic appears. We use `scale_x_discrete` because 'discrete' is **ggplot2**-speak for 'categorical'.

Is there anything else we might want to change? Sometimes it is clearer to plot horizontal bars, particularly if the categorical axis labels are all bunched together (not really the case here). Flipping the x and y axes makes a horizontal bar chart, which we can achieve with the `coord_flip` function (this is new):

```
# display bar plot of species counts
ggplot(penguins, aes(x = species)) +
  # use geom_bar to add bar plot layer of **counts**
  geom_bar(fill = "rosybrown4", width = 0.7) +
  # customise order of x-axis (limits)
```

```
scale_x_discrete(limits = ords) +
# set the the x- and y-axis labels
labs(x = "Species", y = "Number of Observations") +
# flip x- and y-axes
coord_flip()
```
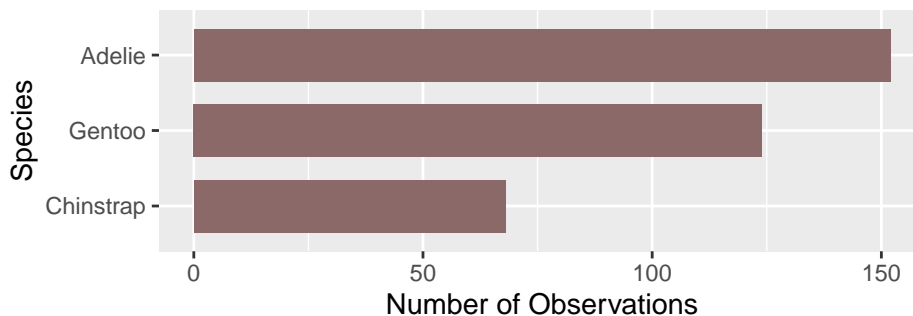

We use very similar R code to produce a bar plot summarising the annual counts, i.e. those associated with the `year` categories:

```
# display bar plot of year counts
ggplot(penguins, aes(x = year)) +
  # use geom_bar to add bar plot layer of **counts**
  geom_bar(fill = "steelblue", width = 0.7) +
  # set the the x- and y-axis labels
  labs(x = "Year", y = "Number of Observations") +
  # flip x- and y-axes
  coord_flip()
```
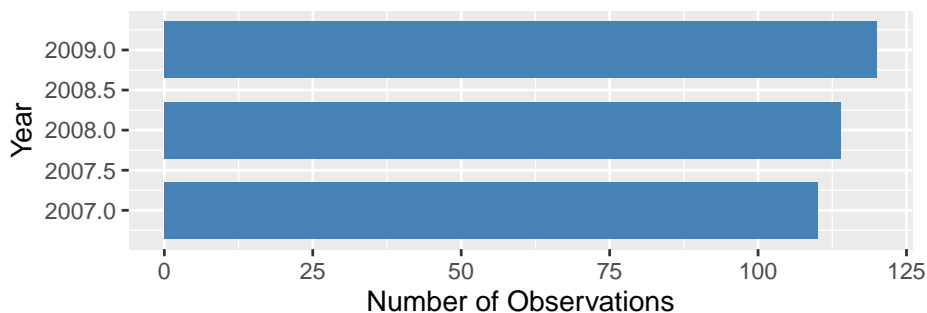

To make this, we changed the aesthetic mapping, colour and the labels, and dropped the `scale_x_discrete` part because the default ordering is acceptable. The resulting plot shows that there are roughly equal numbers of penguins

sampled in each year.

There is one small irritation. Look at the year axis labels and guides. These include non-existent 'half years' (e.g. 2007.5) rather than being limited to only the observed years (2007, 2008 and 2009). This has happened because **ggplot2** doesn't know that `year` is meant to be treated as a categorical variable. It sees a number, and so it constructs a continuous numeric axis for `year`.

We need to force **ggplot2** to treat `year` as a categorical 'thing' to avoid this behaviour. There is more than one way to do this. The simplest is to convert `year` into a character vector. The `as.character` function will do this. What's more, **ggplot2** will allow us to do this when we set up the aesthetic mappings within `aes`:

```r
# display bar plot of year counts
ggplot(penguins,
       # convert year to character when setting up mapping
       aes(x = as.character(year))) +
  # use geom_bar to add bar plot layer of **counts**
  geom_bar(fill = "steelblue", width = 0.7) +
  # set the the x- and y-axis labels
  labs(x = "Year", y = "Number of Observations") +
  # flip x- and y-axes
  coord_flip()
```
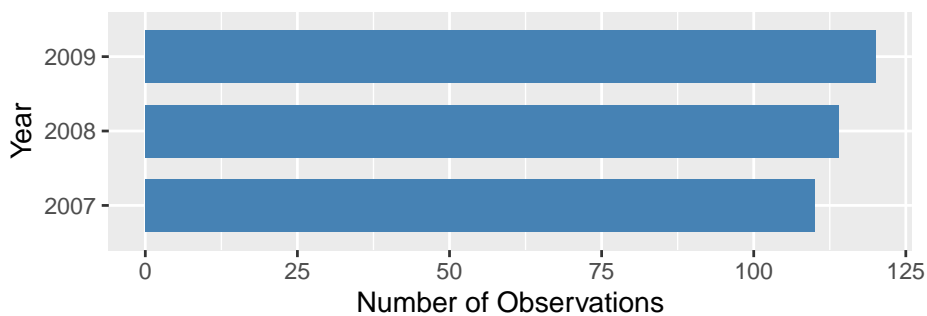

That's better! This may seem like a bit of an edge case but it comes up often enough to be worth highlighting. More generally, this example demonstrates that our conception of a variable needs to matches the way it is interpreted by **ggplot2** to arrive at a sensible plot.

Chapter 16

Exploring associations

A pair of variables are said to be associated when knowing the value of one variable gives us information about the possible values of another. This chapter's main goal is to show how to use visualisations to explore associations among different kinds of variables.

16.1   Associations between numeric variables

The standard graph for displaying an association among a pair of numeric variables is the scatter plot. This uses horizontal and vertical axes to show two variables as a series of points. We saw how to construct a scatter plot in the Introduction to ggplot2 chapter. Here is the initial plot we made to show the association between bill depth and bill length (ignoring species information):

```
# display bill morphology scatter plot
ggplot(penguins,
       # aesthetic mappings: bill depth (y) vs bill length (x)
       aes(x = bill_length_mm, y = bill_depth_mm)) +
  # geom_point to add points layer
  geom_point()
```
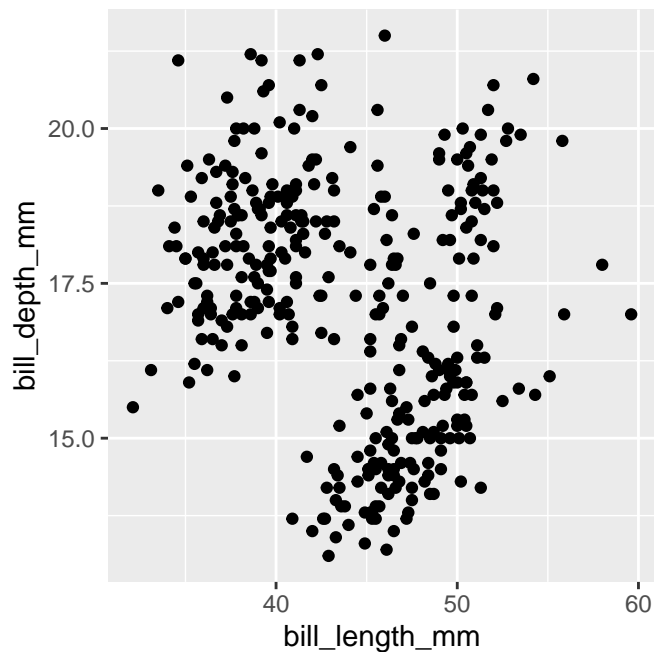
Most of the time, a scatter plot is exactly the right plot for visualising an association among numeric variables. One limitation of scatter plots is that they tend not to reveal associations when there is a lot of over-plotting happening. This occurs when:

1. there is a lot of data (e.g. 1000s of data points), meaning the distance between points is very small; or
2. there are many identical pairs of values, typically because both variables are discrete or a continuous variable is measured on a coarse scale.

In these circumstances, over-plotting of points tends to obscure the association. Sometimes we can mitigate this by making the points very small (problem #1) or semi-transparent (problem #1 or #2). But what should we do when those sorts of tricks fail to improve a plot? Luckily, **ggplot2** provides alternative geoms for visualising numeric associations in situations where over-plotting is a problem.

Two options for dealing with large data sets (problem #1) are the `geom_bin_2d` and `geom_hex` functions. These work like histograms in two dimensions. The `geom_bin_2d` divides the plane into rectangles, counts the number of cases in each rectangle, and then uses the number of cases to assign the rectangle's fill colour. The `geom_hex` function does essentially the same thing but instead divides the plane into regular hexagons.

Here's an example of `geom_hex` in action (n.b. `geom_hex` relies on the **hexbin** package, so this need to be installed to use it):

```r
# display bill morphology scatter plot
ggplot(penguins,
       # aesthetic mappings: bill depth (y) vs bill length (x)
       aes(x = bill_length_mm, y = bill_depth_mm)) +
  # add shaded hex bin layer
  geom_hex(bins = 12)
```
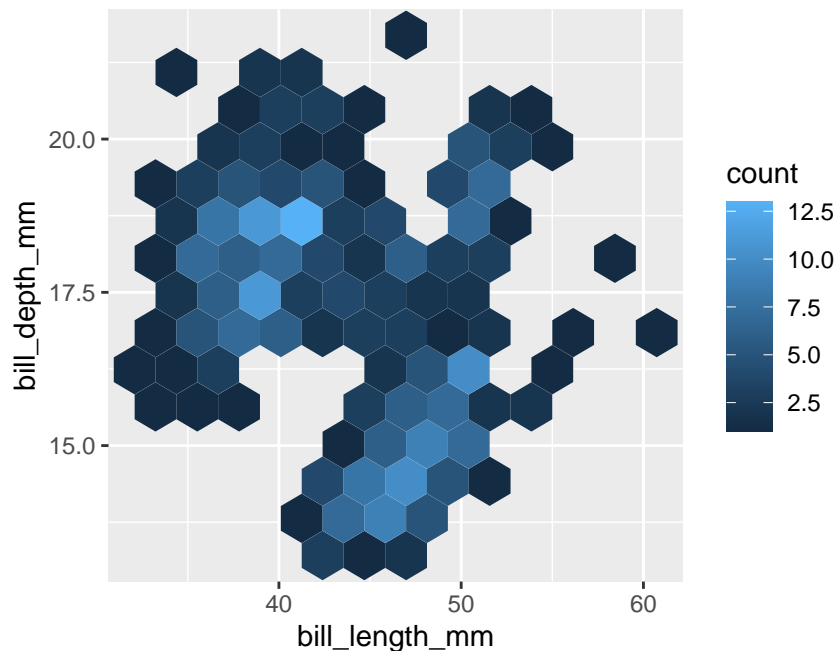

The bins argument controls how many hexagonal bins to use when spanning each dimension. The **penguins** data set is a bit small for this kind of plot (a scatter plot is actually better), so we had to use relatively coarse binning by setting the number of bins to 12.
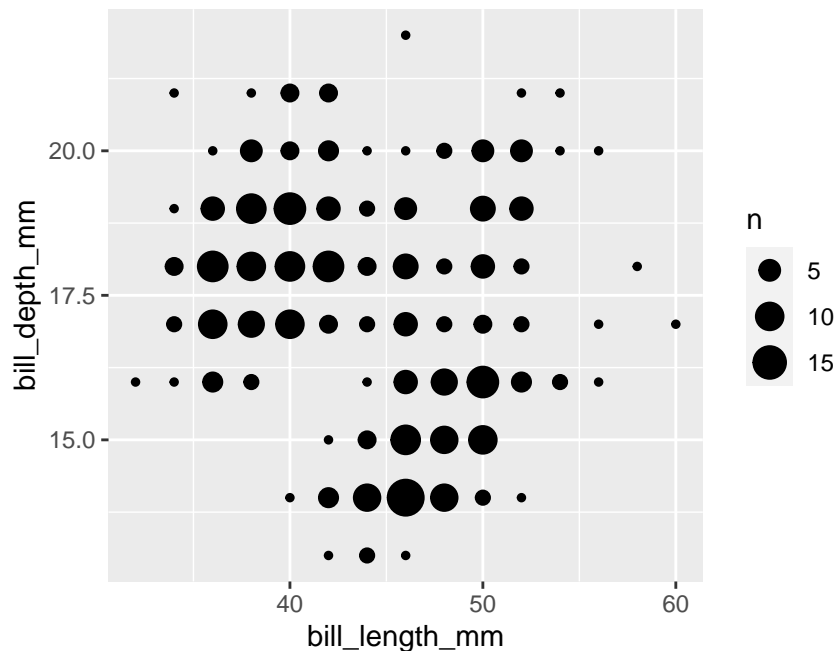
Notice how similar this looks to the **ggplot2** code for a scatter plot. We simply used `geom_hex` in place of `geom_point`. This reveals a powerful feature of **ggplot2**—once we have a working plot, we can often switch to a reasonable alternative by simply changing the type of geom. This makes it very quick to explore a data set in different ways.

What should we do if there are there are many identical pairs of values? For example, imagine that bill length had only been measured to the nearest 2 mm and bill depth had only been measured to the nearest 1 mm, like this:

```
## # A tibble: 344 x 2
##    bill_length_mm bill_depth_mm
##             <dbl>         <dbl>
```

```
## 1                40                19
## 2                40                17
## 3                40                18
## 4                NA                NA
## 5                36                19
## 6                40                21
## 7                38                18
## 8                40                20
## 9                34                18
## 10               42                20
## # ... with 334 more rows
```

A scatter plot `bill_depth_mm` against `bill_length_mm` produced with those coarsened data would look like this:
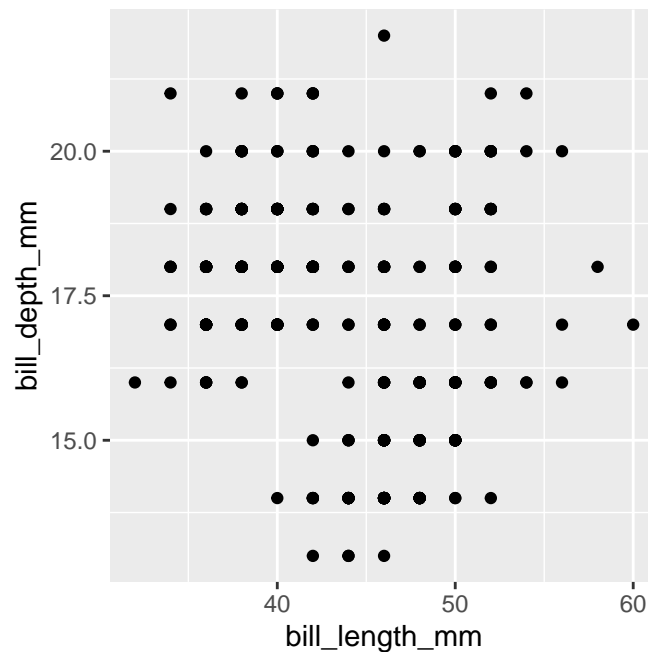
This is not very informative because groups of points appear in exactly the same x/y location.

The `geom_count` function solves this problem by producing a scatter plot where point sizes are scaled according to how many cases are associated with each point. For example, the following code produces a modified scatter plot using a coarsened bill morphology data set (called `penguins_coarse`):

```
# display coarsened bill morphology scatter plot
ggplot(penguins_coarse,
       # aesthetic mappings: bill depth (y) vs bill length (x)
```

```
        aes(x = bill_length_mm, y = bill_depth_mm)) +
  # use geom_count to add sample size scaled points layer
  geom_count()
```


Again, notice how similar this looks to the **ggplot2** code for a standard scatter plot. We simply used `geom_count` in place of `geom_point`.

16.2   Associations between categorical variables

Bar charts are often used to summarise associations between categorical variables. The basic idea is to produce a separate bar for *each combination* of categories in the two variables. The lengths of each bar is proportional to the value it represents, which is either a raw count or the proportion in each category combination.

Using **ggplot2** to display this information is not very different from producing a bar graph to summarise a single categorical variable. Let's do this for the `species` and `year` variables in `penguins`:

```
# display bar plot of year-species counts
ggplot(penguins,
       # aesthetic mappings: year (x) by species (fill)
       aes(x = year, fill = species)) +
```

```
# use geom_bar to add bar plot layer  with stacked bars
geom_bar()
```
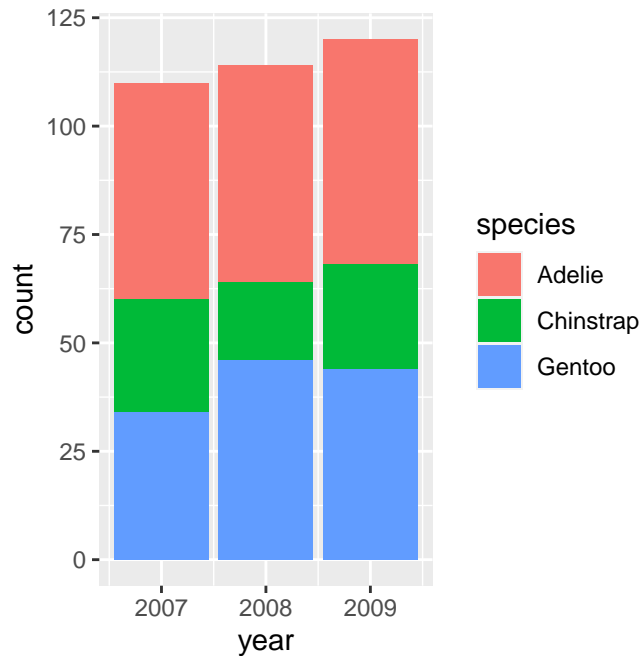

We want to display information from two categorical variables, so we have to define two aesthetic mappings. We mapped the year variable (`year`) to the x axis and species (`species`) to the fill colour. Predictably, we construct the layer using `geom_bar`.

This produced a **stacked bar chart**. Each year has its own bar (`x = year`), and each bar has been divided up into different coloured segments, the length of which is determined by the number of observations associated with each species in that year (`fill = species`).

We have all the right information in this graph, but it could be improved in a couple of ways:

- Look at the guides on the x axis. There are some pointless extra lines. We saw this issue in a different guise in the previous chapter. It occurs because `year` is stored as a numeric vector—**ggplot2** has no way of knowing we want to treat it as a non-numeric, categorical variable.
- The ordering of the species in the stacks does not reflect the relative frequency of each species. Again, we saw this problem in the previous chapter—**ggplot2** treats does not 'know' how we want to order the `species` categories, and so it uses alphabetical order.

We need to ensure `year` is interpreted as a categorical variable and provide information about the required category order of `species`. R has a special kind of vector, called a **factor**, that is designed to tackle these kinds of issues. Factors are used by R to represent categorical variables.

We need to convert `year` and `species` to factors. How do we convert a numeric or character vector into a factor? Use the `factor` function (this is the first time we've mentioned this function). This has an argument called `levels` that sets the allowed categories and their order.

One way to start is by setting up a character vector of species names **in the required order** (`species_names`):

```
# levels of new species factor **in required order**
species_names <- c("Chinstrap", "Gentoo", "Adelie")
```

We supplied the names in the order we want them to appear. Be careful with the spelling—the values of `species_names` have to match those used in `species`. We can now remake the bar plot, this time converting `species` and `year` to factors 'on the fly':

```
# display bar plot of year-species counts
ggplot(penguins,
       # aesthetic mappings: year (x) by species (fill) AND convert to factor
       aes(x = factor(year), fill = factor(species, levels = species_names))) +
  # use geom_bar to add bar plot layer with stacked bars
  geom_bar()
```
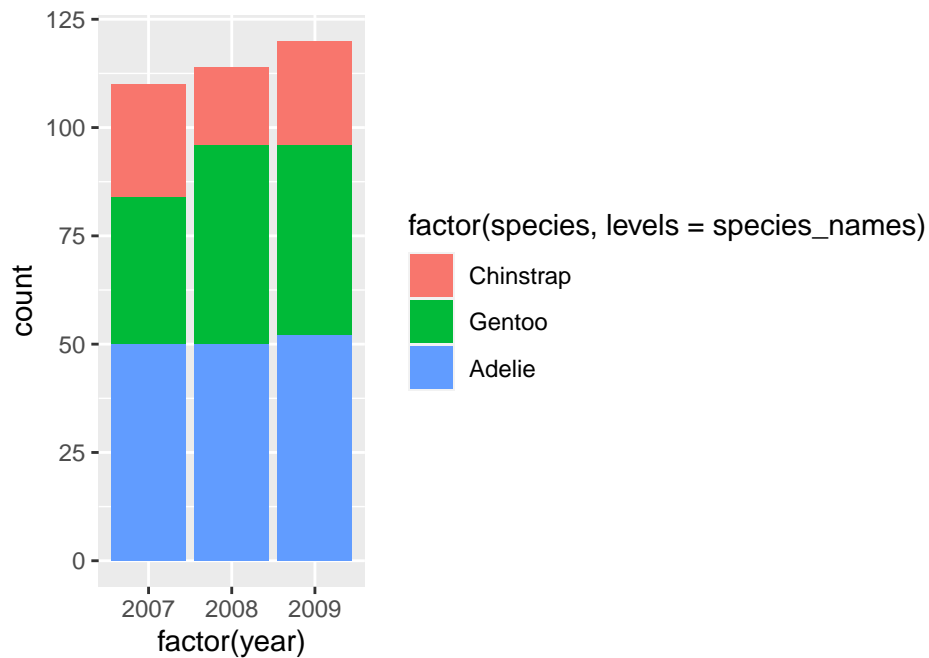
This uses `species_names` to set the levels of `species` when we convert it to a factor. The stacking order of the bars then reflects their order in `species_names`. There was no need to set the levels of `year` because R will use the numeric order by default. Yes… the labels in that plot are ugly, but we could easily change these using `labs` function if we wanted to.

> ⚠️ **Factors**
>
> Factors are very useful and crop up all the time in R. A complete treatment of factors would require a whole new chapter. We've just shown one way to work with them via the `factor` function. This is enough to solve the reordering trick required to get **ggplot2** to work the way we want it to, but there's a lot more to learn about factors. There is even a package (**forcats**) that exists solely to make working with factors a bit easier.
>
> One thing to be aware of is that it is sometimes simpler to convert variables to factors in a data set, rather than doing it every time we use them with **ggplot2**. For example, we could use `mutate` to create factor versions of `year` and `species`:

```r
# convert year and species to factors to ensure correct ordering
penguins <- penguins %>%
  mutate(year    = factor(year),
         species = factor(species, levels = species_names))
```

Any **ggplot2** code that uses the updated version of `penguins` would then automatically respect the intended categorical nature of `year` of and the `species` ordering we desire.

A stacked bar chart is the `geom_bar` default. If we want to know how two categorical variables are associated it is sometimes better to plot all bars side-by-side. This is not hard to do. We switch to a side-by-side bar chart by assigning a value of `"dodge"` to the `position` argument of `geom_bar`:

```r
# display bar plot of year-species counts
ggplot(penguins,
       # aesthetic mappings: year (x) by species (fill) AND convert to factor
       aes(x = factor(year), fill = factor(species, levels = species_names))) +
  # use geom_bar to add bar plot layer with adjacent fill bars
  geom_bar(position = "dodge")
```
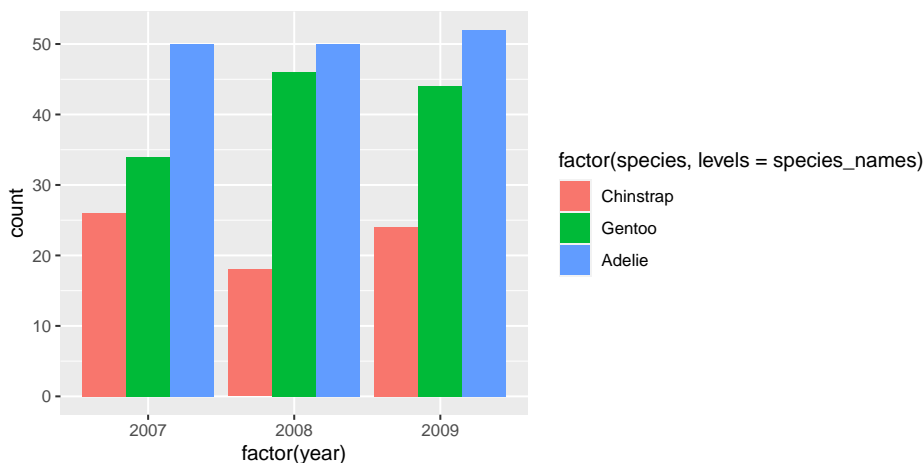

The `position = "dodge"` argument says that we want the bars to 'dodge' one another along the x axis. This slightly odd language is how **ggplot2** specifies the position of grouped items so they are displayed next to one another.

Remember—the customisation techniques introduced in the Customising plots chapter are completely general. For example, we could improve the plot by using `labs` to set the labels, setting custom fill colours with `scale_fill_manual`, and

applying a theme via `theme_minimal`:

```
# display bar plot of year-species counts
ggplot(penguins,
       # aesthetic mappings: year (x) by species (fill) AND convert to factor
       aes(x = factor(year), fill = factor(species, levels = species_names))) +
  # use geom_bar to add bar plot layer with fill bars next to one another
  geom_bar(position = "dodge") +
  # customise fill colour scale
  scale_fill_manual(values = c("cornflowerblue", "seagreen", "orangered3")) +
  # specify labels for all mappings
  labs(x = "Year", y = "Number of cases", fill = "Species") +
  # apply minimal theme
  theme_minimal()
```
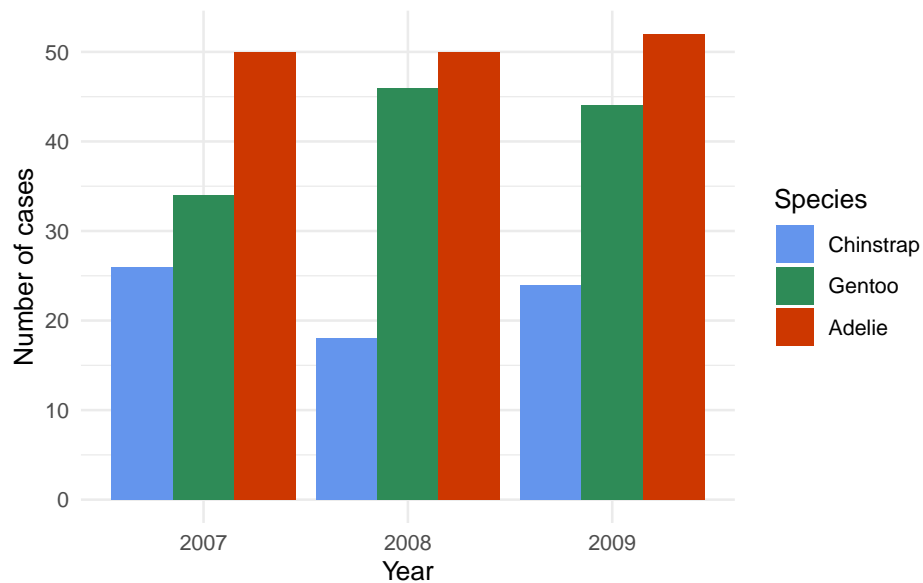

16.3   Categorical-numerical associations

The next obvious question is, "How do we display the association between a categorical and numeric variable?" As usual, there are a range of different options.

The most common is the 'box and whiskers plot' (or just 'box plot'). This is best understood by example. To construct a box and whiskers plot we need to set 'x' and 'y' axis aesthetics for the categorical and numeric variable; then use the `geom_boxplot` function to add the appropriate layer. To examine the association between penguin species (`species`) and body mass measurements

(`body_mass_g`), use:

```r
# display box and whiskers plot of body mass for each species
ggplot(penguins,
       # aesthetic mappings: body mass (y) vs species (x)
       aes(x = species, y = body_mass_g)) +
  # use geom_boxplot to add box plots layer
  geom_boxplot() +
  # specify labels for x- and y-axes
  labs(x = "Species", y = "Body mass (g)")
```
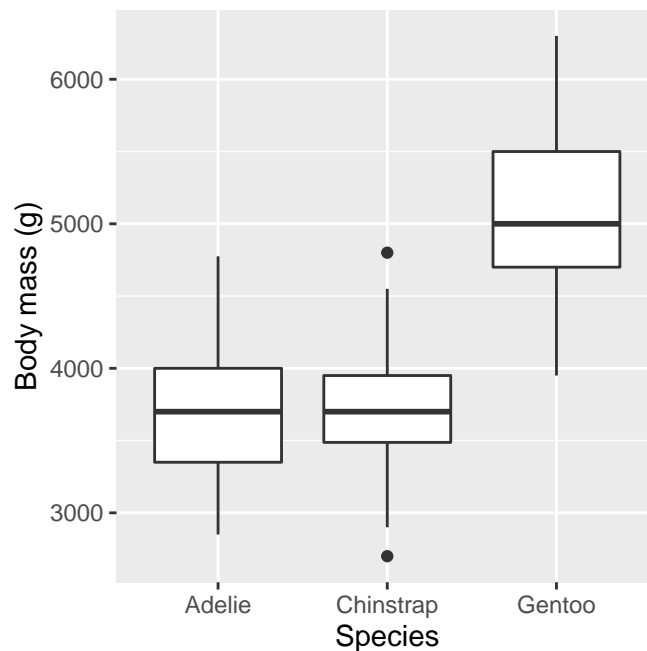

We can see why this is called a box and whiskers plot. There are four components of each box and whiskers:

- The horizontal line inside the box is the sample median of the numeric variable—a measure of its central tendency. This allows us to compare the most likely value of the numeric variable across the different categories.

- The boxes display the interquartile range (IQR) of the numeric variable in each category, i.e. the middle 50% of each group. This allows us to compare the spread (=dispersion) of the numeric values across categories.

- The vertical lines above and below each box are the "whiskers". The interpretation of these depends on which kind of box plot we are making. **ggplot2** produces a traditional Tukey box plot by default. Each whisker is drawn from each end of the box (the upper and lower quartiles) to a well-

defined extreme point. To find where the upper whisker ends, we have to find the largest observation that is no more than 1.5 times the IQR above the upper quartile. The lower whisker ends at the most extreme point that is no more than 1.5 times the IQR below the lower quartile.

- Any cases that do not fall inside the whiskers are plotted as an individual point. These may be outliers, although they could also be perfectly consistent with the wider distribution.

The resulting plot compactly summarises the distribution of the numeric variable within categories. It provides information about the central tendency, dispersion and skewness of each distribution. We also get a sense of whether there are potential outliers by noting the presence of individual points outside the whiskers.

What does the above plot tell us about body mass and species? It shows the body mass distributions of Chinstrap and Adelie penguins are very similar, in terms of both central tendency and dispersion. In contrast, Gentoo penguins are generally larger and exhibit great absolute variation. The Gentoo body mass distribution is also the only one that seems to display much skew.

16.3.1   Alternatives to box and whiskers plots

Box and whiskers plots are a good choice for exploring categorical-numerical associations. They provide a lot of information about how the distribution of the numeric variable changes across categories. Sometimes we may want to squeeze even more information about these distributions into a plot. One way to do this is to make multiple histograms.

We know how to make a histogram, and we have seen how aesthetic properties such as `colour` and `fill` are used to distinguish different categories of a variable. This suggests we could overlay more than one histogram on a single plot. Let's use this idea to see how the sample distribution of body mass differs among the species again:

```r
# display histogram of body mass for each species in one plot
ggplot(penguins,
       # aesthetic mappings: body mass (x) by species groups (fill)
       aes(x = body_mass_g, fill = species))  +
  # use geom_histogram to add histogram layer with customisation
  geom_histogram(position = "identity", alpha = 0.8, binwidth = 200) +
  # specify labels for all mappings
  labs(x = "Body mass (g)", fill = "Species")
```
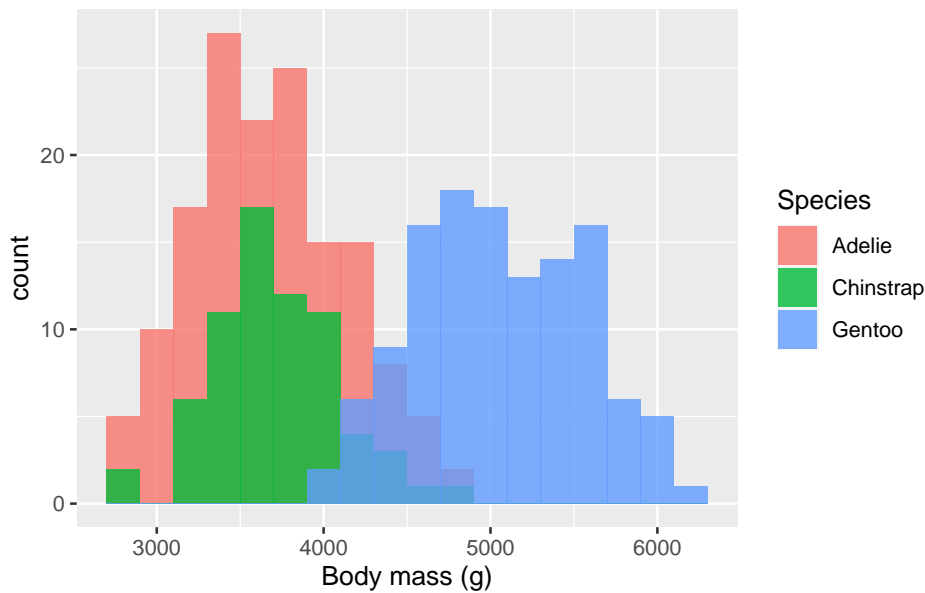
We defined two mappings: the continuous variable (`body_mass_g`) was mapped to the x axis and the categorical variable (`species`) was mapped to the fill colour. Notice that we also set the `position` argument to `"identity"`. This tells **ggplot2** not to stack the histograms on top of one another. Instead, they are allowed to overlap. It's for this reason that we also made them semi-transparent by setting the `alpha` argument.

Plotting several histograms in one layer like this places a lot of information in one plot, but it can be hard to make sense of when the histograms overlap a lot. If the overlapping histograms are too difficult to interpret, we might consider producing a separate one for each category. We've already seen a quick way to do this. Faceting works well here:

```
# display histogram of body mass for each species in separate panels
ggplot(penguins,
       # aesthetic mappings: body mass (x) (only one!)
       aes(x = body_mass_g))  +
  # use geom_histogram to add histogram layer with sensible bin width
  geom_histogram(binwidth = 200) +
  # specify label for x-axis
  labs(x = "Body mass (g)") +
  # facet by species to get one panel per species
  facet_wrap(vars(species))
```
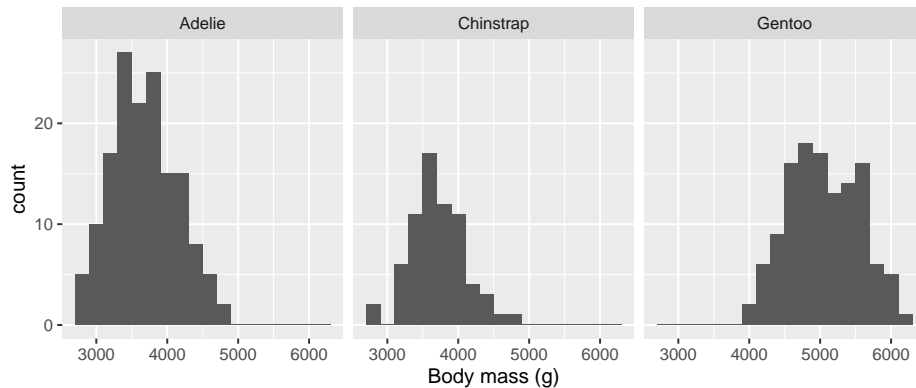
The two histogram plots tell much the same story as the box and whiskers plot. The body mass distributions of Chinstrap and Adelie penguins are very similar, Gentoo penguins are generally a bit larger, but all three distributions overlap to some extent.

16.4   Multivariate associations

We have now examined several ways for summarising associations between two variables. How do we explore associations between more than two variables in a single graph? That is, how do we explore **multivariate associations**? It is difficult to give a concrete answer because it depends on the scientific question we're addressing, the kinds of variables we're working with, and to a large extent, our creativity and aptitude with a graphing framework like **ggplot2**.

We have already seen enough of **ggplot2** works to build some fairly sophisticated visualisations.The key point is that we don't really need to learn anything new to visualise multivariate associations. There are two ways to add additional information to a visualisation, both of which we have already reviewed:

1. Define aesthetic mappings to allow a layer's properties to depend on the values of one or more variables.
2. Use faceting to construct a multi-panel plot according to the values of categorical variables.

Constructing a multivariate visualisation is a matter of combining these approaches to squeeze information from several variables into a single graph.

We saw these approaches used together in the Introduction to ggplot2 chapter. Let's look at one more example to cement the idea. We want to understand how the flipper length relates to body mass and how this numeric-numeric relationship varies by species, sex and island. That's five variables in one plot!

The obvious idea is to produce a multi-panel scatter plot, where each panel is associated with combinations of the categorical variables. We have already seen how to do this kind of thing with `facet_wrap`. This time, we'll use the

`facet_grid` function to organise separate panels for each sex-species combination into a nice 2d grid:

```
#
ggplot(na.omit(penguins),
       # aesthetic mappings
       aes(
         # flipper length (y) vs body mass (x)
         x = body_mass_g, y = flipper_length_mm,
         # colour AND shape of points vary by island
         colour = island, shape = island)) +
  # geom_point to add points layer
  geom_point() +
  # specify labels for ALL mappings (both Island labels required)
  labs(x = "Body size (g)",
       y = "Flipper length (mm)",
       colour = "Island", shape = "Island") +
  # facet on sex by species grid
  facet_grid(rows = vars(sex), cols = vars(species))
```
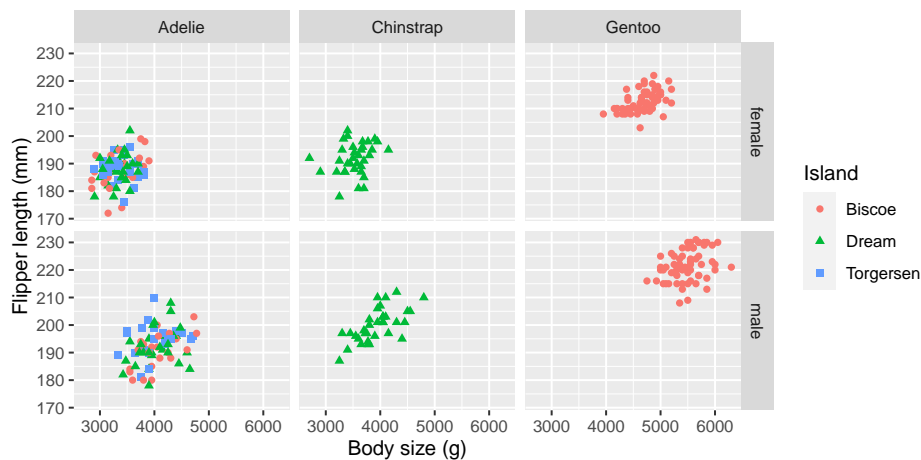

Simple. We set some aesthetics and used faceting to squeeze the five variables into one plot. The `facet_grid` function represents the values two or more categorical variables by row and column position, which means we have to set `rows` and `cols` arguments. The `vars(…)` bit is not optional by the way. We also took the opportunity to introduce a couple of new tricks:

- We used the base R `na.omit` function on penguins to strip out any rows with a missing value. This avoids introducing an 'NA' category in the faceting. This trick can be a bit dangerous because `na.omit` removes every case where a value is missing.
- We introduced some redundancy by mapping one variable (`island`) to

two different aesthetics (`colour` and `shape`). This achieves two things. It makes it easier to differentiate the points, and it ensures a plot can be understood if printed in black and white.

There is lots of information in this plot—for example, the association between flipper length and body mass is weakly positive, sexual dimorphism in body size seems to be greatest in Gentoo penguins, and the Adelie species is the only one present on all three islands.

Chapter 17

Doing more with ggplot2

We have now seen a range of different ways to visualise and explore data using **ggplot2**. This last chapter will cover a few bits of miscellanea that don't particularly fit anywhere else. That doesn't mean the ideas aren't important!

17.1   Comparing descriptive statistics

So far, we have focussed on displaying the raw data (e.g. scatter plots) or a distributional summary (e.g. box plots). What other types of quantities might we need to visualise? Descriptive statistics such as the sample mean are one possibility. These often feature in data analysis when 'comparing groups'.

We need to know how to construct plots that display such summaries. Let's start with a simple question: how does the (arithmetic) mean body mass varies by sex and across penguin species? One option is to produce a bar plot for which the lengths of bars represent the mean body mass in each category.

There are two different routes to produce this with **ggplot2**. One way to build such a plot is to break the problem into two steps. In the first step, we calculate whatever it is we want to display, i.e. the species- and sex-specific mean body mass. **dplyr** is usually the best tool to use for this first step:

```
penguins_summary <- penguins %>%
  # group data by species and penguins
  group_by(species, sex) %>%
  # calculate mean body mass
  summarise(mean_mass = mean(body_mass_g)) %>%
  # remove rows generated by sex = NA cases
  na.omit()
```

```
## `summarise()` has grouped output by 'species'. You can override using the `.groups` argument.
```

We used `group_by` and `summarise` to calculate the set of means, which we decided to call `mean_mass`. Notice that we used `na.omit` to exclude the `NA` categories that arise when sex is unknown. The resulting `penguins_summary` object is a small 6 row by 3 column tibble:

```
# show the summary data
penguins_summary
```

```
## # A tibble: 6 x 3
## # Groups:   species [3]
##   species   sex     mean_mass
##   <chr>     <chr>       <dbl>
## 1 Adelie    female      3369.
## 2 Adelie    male        4043.
## 3 Chinstrap female      3527.
## 4 Chinstrap male        3939.
## 5 Gentoo    female      4680.
## 6 Gentoo    male        5485.
```

The second step uses `penguins_summary` with **ggplot2** to create the required bar plot:

```
# display bar plot of body mass means
ggplot(penguins_summary,
       # aesthetic mappings: mass (y) vs species (x) by sex (fill)
       aes(x = species, fill = sex, y = mean_mass)) +
  # use geom_col to add bar plot layer 'as is' with adjacent bars
  geom_col(position = "dodge") +
  # specify labels for all mappings
  labs(x = "Species", y = "Mean body mass (g)", fill = "Sex")
```
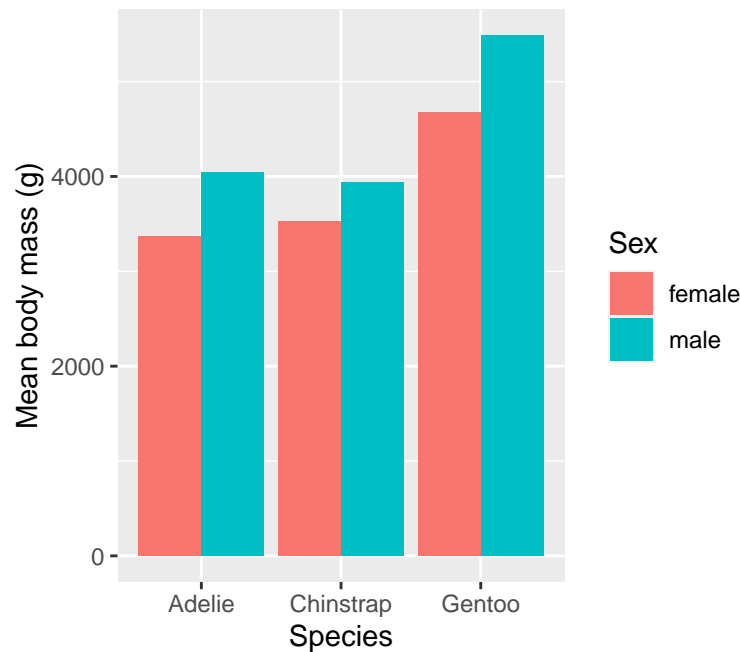
Two points about this are worth noting:

1. We used `geom_col` instead of `geom_bar`. `geom_bar` counts the observations in each category by default. Using `geom_col` tells **ggplot2** to plot the information in `mean_mass` 'as is'.
2. We paid close attention to the names of things. The plotting data lives in `penguins_summary` which means the y aesthetic must be associated with `mean_mass` rather than `body_mass_g`.

It is possible to produce this plot with less code by using **ggplot2**'s built-in 'stat' facility (see box). We recommend the long-winded way when starting out with R because it makes it a bit easier to fix mistakes—we can check whether the right information is in a summary data set before plotting it.

> ⓘ **Using ggplot2 'stats'**
>
> Another way to arrive at the above bar plot is by using the built-in **ggplot2** `stat` facility:

```
# display bar plot of body mass means
ggplot(na.omit(penguins), # <- remove missing values
       # aesthetic mappings: mass (y) vs species (x) by sex (fill)
       aes(x = species, fill = sex, y = body_mass_g)) +
  # use geom_bar to add bar plot layer using a stats summary
  geom_bar(stat = "summary", fun=mean, position = "dodge") +
  # specify labels for all mappings
  labs(x = "Species", y = "Mean body mass (g)", fill = "Sex")
```
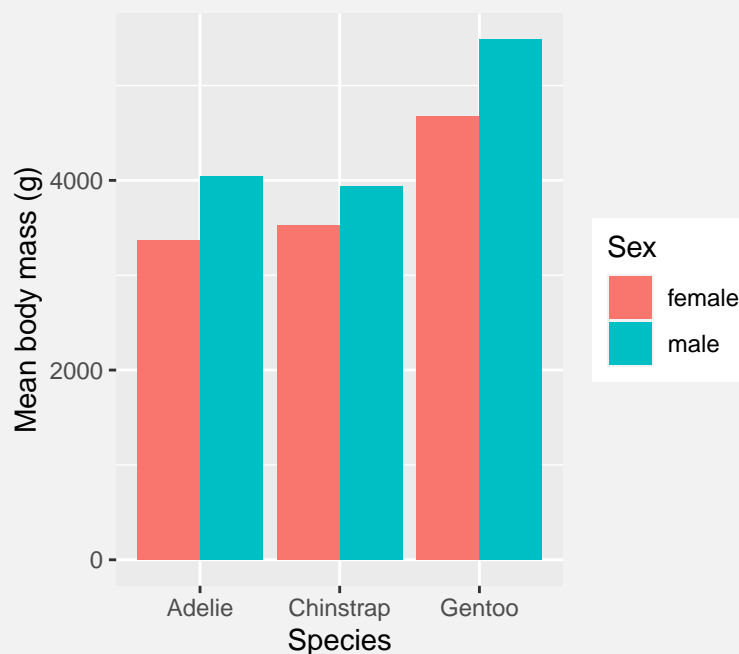

This involves a couple of new tricks. When we add a layer using `geom_bar` we set two additional arguments:

- `stat = "summary"` tells **ggplot2** not to plot the raw values of the y aesthetic mapping, but instead, to calculate and then plot a summary of the 'y' variable.
- `fun = mean` tells **ggplot2** how to summarise the 'y' variable. The name of the function on the right-hand side can be any function that takes a vector of values as input and returns a summary. We want the `mean` function.

Notice that this works on the raw data (`penguins`)—there is no need to manually calculate the means because that happens inside **ggplot2**.

17.1.1 Error bars

A descriptive statistic like the mean isn't much use on its own. To properly interpret means we also need to know something about their uncertainty. There are many ways to quantify the uncertainty of an estimate. Whichever one we are using, displaying the uncertainty typically involves adding **error bars** to a plot.

We'll demonstrate how to do this by extending the previous example to show the mean and standard error of body mass for each species.

> ℹ️ **Er.... standard error?**
>
> The standard error is a measure of how precise an estimate like the sample mean is. A small standard error indicates that we can have more confidence the estimate reflects the 'true' value. The standard error of the mean can be calculated from a well-known formula:
>
> $$\text{Standard Error} = \frac{\text{Standard Deviation}}{\sqrt{\text{(Sample Size)}}} = \frac{\sigma}{\sqrt{n}}$$
>
> It is hard to give an explanation of that formula without knowing a bit of statistical theory—it is just one of those things we have to learn. In terms of code that works with **dplyr**, that calculation looks like this:
>
> ```
> standard_error = sd(x) / sqrt(n())
> ```
>
> This assumes the numeric variable we are working with is called `x`, and we want the result to be called `standard_error`. Notice this uses a special **dplyr** function called `n` to find the sample size.

Once again, constructing the plot is a two-step process. Start by calculating the means and standard error of body mass for each species and sex combination:

```
penguins_summary <- penguins %>%
  # group data by species and penguins
  group_by(species, sex) %>%
  # calculate summaries
  summarise(
    # mean mass
    mean_mass = mean(body_mass_g, na.rm = TRUE),
    # standard error
    ster_mass = sd(body_mass_g, na.rm = TRUE) / sqrt(n())
  ) %>%
  # remove rows generated by sex = NA cases
  na.omit()
```

```
## `summarise()` has grouped output by 'species'. You can override using the `.groups`
```
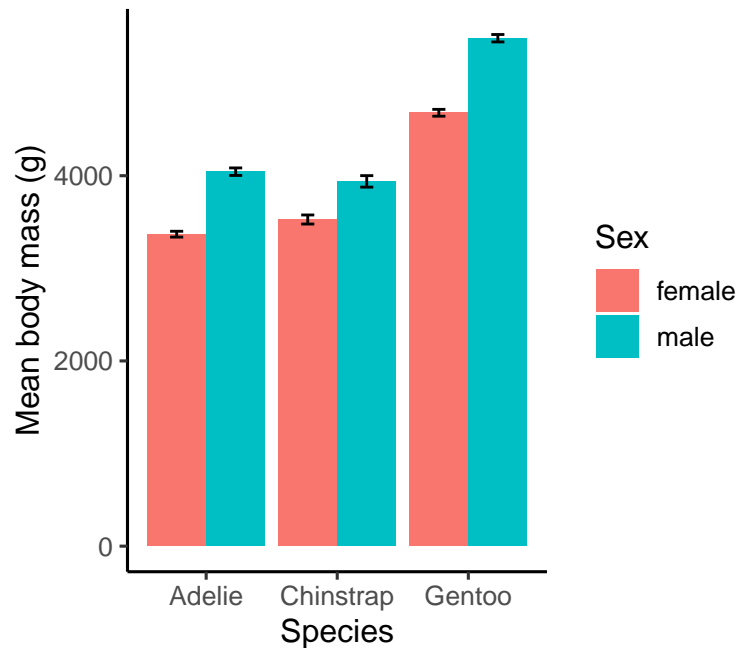
We use **dplyr** to calculate the means and standard errors of each species' body
mass, which we called `mean_mass` and `ster_mass`, respectively. Again, we used
`na.omit` to exclude the `NA` categories cases. The new `penguins_summary` object
is a 6 row by 4 column tibble:

```
# show the new summary data
penguins_summary
```

```
## # A tibble: 6 x 4
## # Groups:   species [3]
##   species   sex    mean_mass ster_mass
##   <chr>     <chr>      <dbl>     <dbl>
## 1 Adelie    female     3369.      31.5
## 2 Adelie    male       4043.      40.6
## 3 Chinstrap female     3527.      48.9
## 4 Chinstrap male       3939.      62.1
## 5 Gentoo    female     4680.      37.0
## 6 Gentoo    male       5485.      40.1
```

Next we use `penguins_summary` to make the plot. We can use `geom_col` to show
the means as a bar plot, but what about the error bars? Use the unsurprisingly
named `geom_errorbar` to add those! Here is the code:

```
# display bar plot of body mass means + SE's
ggplot(penguins_summary,
       # aesthetic mappings: mass (y) vs species (x) by sex (fill)
       aes(x = species, fill = sex, y = mean_mass)) +
  # use geom_col to add an adjacent bar plot layer
  geom_col(position = "dodge") +
  # use geom_errorbar to add an error bar layer
  geom_errorbar(aes(ymin = mean_mass - ster_mass, ymax = mean_mass + ster_mass),
                # customise size of hat + ensure error bars are centred
                width = 0.2, position = position_dodge(0.9)) +
  # use a more professional theme
  theme_classic(base_size = 12) +
  # specify labels for all mappings
  labs(x = "Species", y = "Mean body mass (g)", fill = "Sex")
```

The only new things appear in the `geom_errorbar` part:

- The `ymin` and `ymax` arguments of the `geom_errorbar` function give the lower and upper limits of error bars. Here, we have plotted the mean +/- 1 standard error, i.e. each error bar is two standard errors long.
- Use the `width` argument to set the width of the 'hat' on each error bar. A value of zero gets rid of it altogether.
- The position has to be set in two places: `geom_col` and `geom_errorbar`. For a bar plot, we use the not-at-all-intuitive `position = position_dodge(0.9)` to locate the error bars at the centre of each bar.

⚠ **Warning!**

Data visualisations can include many different kinds of 'error bar'— standard deviation, standard error, 95% confidence intervals, etc. When we include any error bars on a plot, we must remember to state which kind was used in the figure legend. Otherwise, readers will have no way of knowing how to interpret the plot.

17.1.2 Alternatives to bar plots

Do we have to use a bar plot to display means? Certainly not. In some ways bar plots are a bit old fashioned. People often prefer to show a set of means as points with error bars. This kind of visualisation is very easy to make now that we already have working bar plot code:

```r
# display points plot of body mass means + SE's
ggplot(penguins_summary,
       # aesthetic mappings: mass (y) vs species (x) by sex (colour)
       aes(x = species, colour = sex, y = mean_mass)) +
  # use geom_col to add an adjacent bar plot layer
  geom_point(position = position_dodge(0.5)) +
  # use geom_errorbar to add an error bar layer
  geom_errorbar(aes(ymin = mean_mass - ster_mass, ymax = mean_mass + ster_mass),
                # customise size of hat + ensure error bars are centred
                width = 0.2, position = position_dodge(0.5)) +
  # use a more professional theme
  theme_classic(base_size = 12) +
  # specify labels for all mappings
  labs(x = "Species", y = "Mean body mass (g)", colour = "Sex")
```
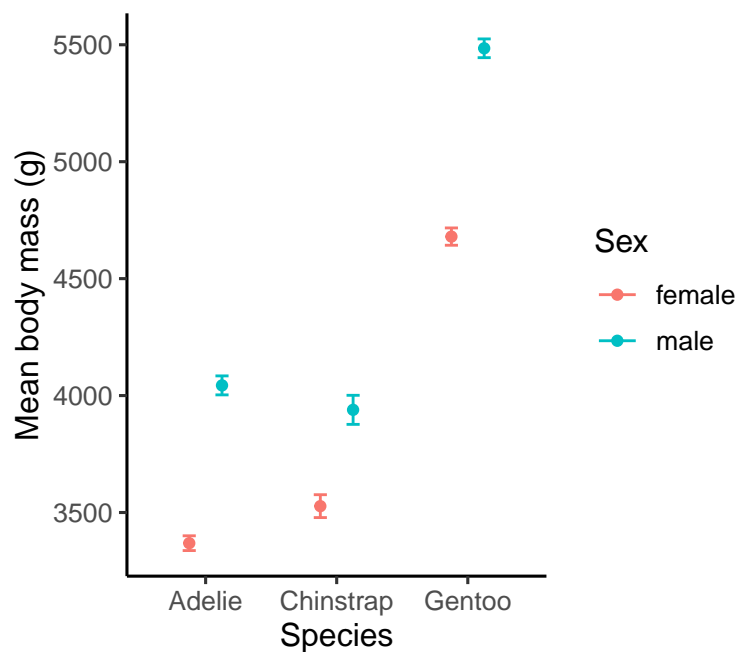

We made three small changes to the bar plot code to produce this. We...

- mapped `sex` to the `colour` aesthetic instead of `fill`,
- switched from using `geom_col` to `geom_point`, and
- altered the position adjustments to 'dodge' by a smaller amount.

That's it!

17.2   Adding text annotations

We often need to include text annotations on a plot. For example, we might want
to show sample sizes or highlight 'statistically significant' differences. Adding
text is no different from adding other kinds of objects to a plot—we use an
appropriate geom (e.g. `geom_text`) to include textual information. Let's see
how this works by adding sample size labels the bar chart we've been building.

We start by placing the labels we want to use in a data frame (or tibble) alongside
any other variables used in aesthetic mappings. This can be distinct from the
data set used to construct the plot but its often simplest to put everything
together in one place. Let's do that by rebuilding the `penguins_summary` data
set to include the mean body mass, standard error and sample size for each
group.

```r
penguins_summary <- penguins %>%
  # group data by species and penguins
  group_by(species, sex) %>%
  # calculate summaries
  summarise(
    # mean mass
    mean_mass = mean(body_mass_g, na.rm = TRUE),
    # sample size
    samp_size = n(),
    # standard error
    ster_mass = sd(body_mass_g, na.rm = TRUE) / sqrt(n())
  ) %>%
  # remove rows generated by sex = NA cases
  na.omit()
```

```
## `summarise()` has grouped output by 'species'. You can override using the `.groups` argument.
```

Then we add the text showing the sample size to our plot using the function
`geom_text`:

```r
# display bar plot of body mass means + SE's + sample sizes
ggplot(penguins_summary,
       # aesthetic mappings: mass (y) vs species (x) by sex (fill)
       aes(x = species, fill = sex, y = mean_mass)) +
  # use geom_col to add an adjacent bar plot layer
  geom_col(position = "dodge") +
  # use geom_errorbar to add an error bar layer
  geom_errorbar(aes(ymin = mean_mass - ster_mass, ymax = mean_mass + ster_mass),
                # customise size of hat + ensure error bars are centred
                width = 0.2, position = position_dodge(0.9)) +
  # place sample size annotations above error bars
```

```
geom_text(aes(label = samp_size, y = mean_mass + ster_mass + 200),
          size = 3, colour = "darkgrey", position = position_dodge(0.9)) +
# use a more professional theme
theme_classic(base_size = 12) +
# specify labels for all mappings
labs(x = "Species", y = "Mean body mass (g)", fill = "Sex")
```
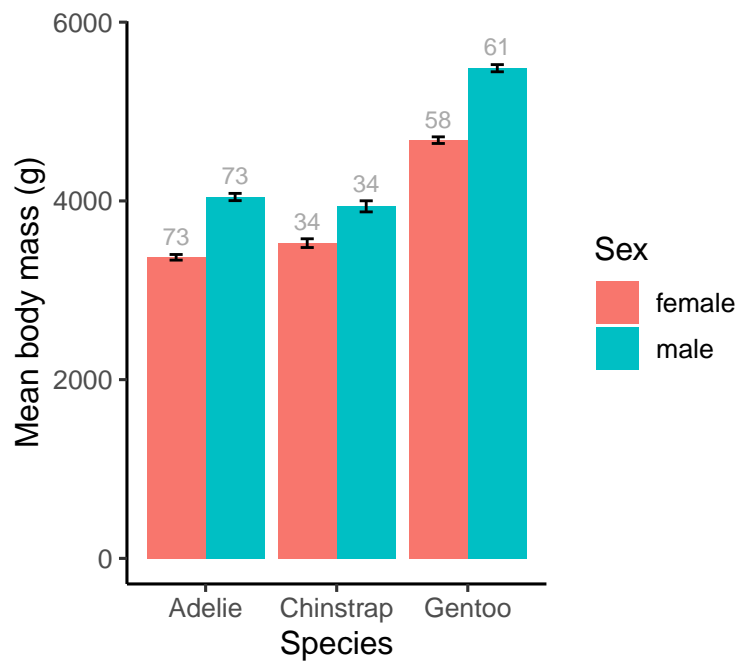

The `y = mean_mass + ster_mass + 200` aesthetic mapping in `geom_text` positions each number just above the error bar. This works because we placed the textual information in the plotting data set, meaning we can use the x/y variables to set the position of the annotations. We also have to use the correct position adjustment to ensure the annotations are located at each bar's centre. Everything else inside `geom_text` is adjusting the appearance of the actual text.

17.3   Saving plots

Plots can be saved using the **Export** button in the RStudio **Plots** tab. However, saving plots this way often leads to a pixelated, low resolution image. The `ggsave` function in **ggplot2** produces much better output and supports a wide range of image file types. There's not much to it. Using `ggsave` looks like this:

```
ggsave("penguins-plot.pdf", height = 5, width = 5)
```

By default, `ggsave` will save the last plot we made to file. If we had made the bar chart and then ran that `ggsave` line, R would save a copy of the plot to as a PDF file called penguins-plot.pdf.

Here is a quick summary of how the function works:

- The first argument is the path and name of the file we want to create. This example would save the plot to a file called penguins-plot.pdf. Because we only provide a file name, `ggsave` would save it to the current working directory.

- `ggsave` supports many different formats (e.g. "pdf", "jpeg", "png", "svg"). This can be set via the `device` argument (e.g. `device = "jpeg"`). Alternatively, if we don't set the format, **ggplot2** will guess it from the file extension.

- The `width` and `height` arguments specify the plot dimensions. These usually require a bit of experimentation to get right. If we do not specify them, `ggsave` will use the current size of the plotting window.

17.4 Multi-panel plots

We have seen how to use `facet_wrap` and `facet_grid` to produce multi-panel plots. These functions are useful when a plot needs to show the same visualisation for different subsets of data. How do we produce a figure that shows a different visualisation in each panel?

There are many ways to do this using an external add-on package. We'll examine the framework provided by the **cowplot** package. Cowplot provides various features that help create high-quality figures, including custom themes and facilities to arrange plots into multi-panel figures.

Constructing a multi-panel plot with **cowplot** happens in two stages. First, make the individual component plots using **ggplot2** as usual, but instead of 'printing' them, assign each graphical object a name. Then use the `plot_grid` function from **cowplot** to construct the multi-panel plot from the stored objects.

For example, assume we want to display at the bill morphology scatter plot and the mean body mass bar chart side-by-side. First we make the two plots assign them names (with minimal commenting to save space):

```
# first plot
plt_a <-
  ggplot(penguins,
         aes(x = bill_length_mm, y = bill_depth_mm, colour = species)) +
```

```
  geom_point() +
  scale_color_manual(values = c("cornflowerblue", "seagreen", "orangered3")) +
  labs(x = "Bill length (mm)", y = "Bill depth (mm)", colour = "Species") +
  theme_classic(base_size = 12)
# second plot
plt_b <-
  ggplot(penguins_summary,
         aes(x = species, fill = sex, y = mean_mass)) +
  geom_col(position = "dodge") +
  geom_errorbar(aes(ymin = mean_mass - ster_mass, ymax = mean_mass + ster_mass),
                width = 0.2, position = position_dodge(0.9)) +
  labs(x = "Species", y = "Mean body mass (g)", fill = "Sex") +
  theme_classic(base_size = 12)
```
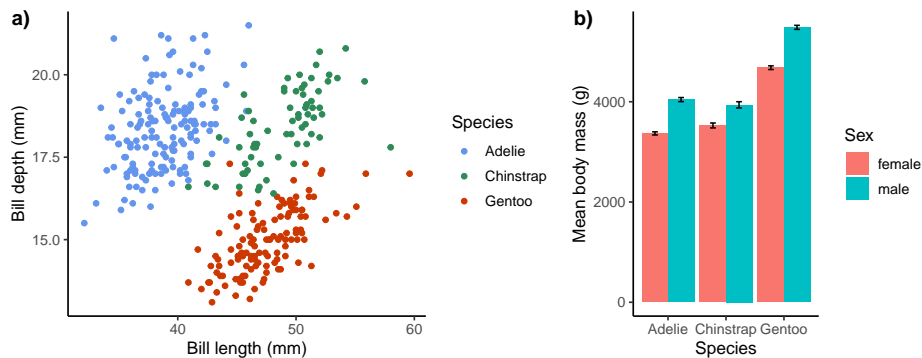
Then we use **cowplot**'s `plot_grid` to produce the two-panel plot:

```
plot_grid(
  # two-panel plot
  plt_a, plt_b,
  # set location and size of panels
  nrow = 1, rel_widths = c(60, 40),
  # set the panel labels
  labels = c("a)", "b)"), label_size = 14
)
```


Once we've made a multi-panel plot, we can use the **ggsave** function to save it
as before. The `plot_grid` function allows the figure to be customised in various
ways. For example:

- **nrow** and **ncol** control the number of plots in each row/column,
- **labels** sets the labels used for each panel,
- **label_size** controls the size of the label font, and
- **rel_widths** and **rel_heights** control how much space each plot gets.

The package website has some good articles that explain how this all works (along with the many other **cowplot** facilities).

Appendix A

Getting help

A.1 Introduction

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

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

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

A.2 Browsing the help system

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

```
help.start()
```

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

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

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

1. The **Manuals** section looks like it might be useful for novice users. Unfortunately, it's not. Even the "Introduction to R" manual is only helpful for someone who understands what terms like 'data structure' and 'data type' mean. The others are more or less impenetrable unless the reader already knows quite a bit about computing in general.

2. The **Reference** section is a little more helpful. The "Packages" link takes us to the same page opened by `help.start`. From here we can browse the help pages on a package-specific basis. The "Search Engine & Keywords" link takes us to a page we can use to search for specific help pages, either by supplying a search term or by navigating through the different keywords.

3. The **Miscellaneous Material** section has a couple of potentially useful links. The "User Manuals" link lists any user manuals supplied by package authors. The "Frequently Asked Questions" link is definitely worth reviewing at some point, tough again, most of the FAQs are a little difficult for novice users to fully understand.

A.3   Searching for help files

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

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

After we run this line RStudio opens up the help file for the trigonometry topic in the **Help** tab. This file provides information about the various trigonometric

functions such as `sin` or `cos` (we'll see how to make sense of such help pages in a bit).

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

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

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

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

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

```
help(mutate)
```

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

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

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

```
?sin
```

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

A.4   Navigating help files

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

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

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

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

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

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

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

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

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

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

A.5 Vignettes

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

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

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

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

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

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

```
browseVignettes()
```

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

Appendix B

Managing projects, scripts and data files

Content will be added in block three.