Introduction to data analysis and visualization with R

György Barabás

Table of contents

Welcome

Lecture notes for NBIC58 (Analysis of Biological Data) at Linköping University, organized into the beginnings of a book. This work is preliminary, so any feedback is much appreciated!

1 Introduction to R and RStudio

1.1 Overview

This chapter introduces R and RStudio. R is a free and open-source programming language for statistics, graphing, and modeling, originally developed by statisticians. In recent years, R has become extremely popular among biologists, and you will almost certainly encounter it as part of real-world research projects. In this course, we will be learning some of the ways in which R can be used for efficient data analysis and visualization.

RStudio is an "integrated development environment" (IDE) for R, which means it is a software application that lets you write, run, and interact graphically with programs. RStudio integrates a text editor, the R console (where you run R commands), package management, plotting, help, and more.

1.2 Installing R and RStudio

You can download the most up-to-date R distribution for free here:

http://www.r-project.org

Run the installer as directed and you should be set to go. We will not interact with the installed R application directly, but the R software components you install will be used by RStudio under the hood.

To install RStudio on your computer, download it from here:

https://www.rstudio.com/products/rstudio/download/

On a Mac, open the downloaded disk image and drag the RStudio application into your Applications folder. On Windows, run the installer you downloaded.

1.3 Getting Around RStudio

RStudio should be available in the usual places: the Applications folder (on a Mac) or the Start menu (on Windows). When you start it up, you will see four sections of the screen. In short:

- Upper-left: an area for viewing and editing text files
- Lower-left: **Console**, where you send commands to R
- Upper-right:
  - **Environment**: for loading, saving, and examining data
  - **History**: a list of commands you have typed
  - **Connections**: for establishing remote connections with data sources
  - **Tutorial**: resources for learning R
- Lower-right:
  - **Files**: a list of files in the "working directory" (more on this later)
  - **Plots**: where the plots (graphs) you draw show up
  - **Packages**: an area for managing installed R packages
  - **Help**: access to all the official documentation for R
  - **Viewer**: you can view certain objects here, e.g., formatted tables of data

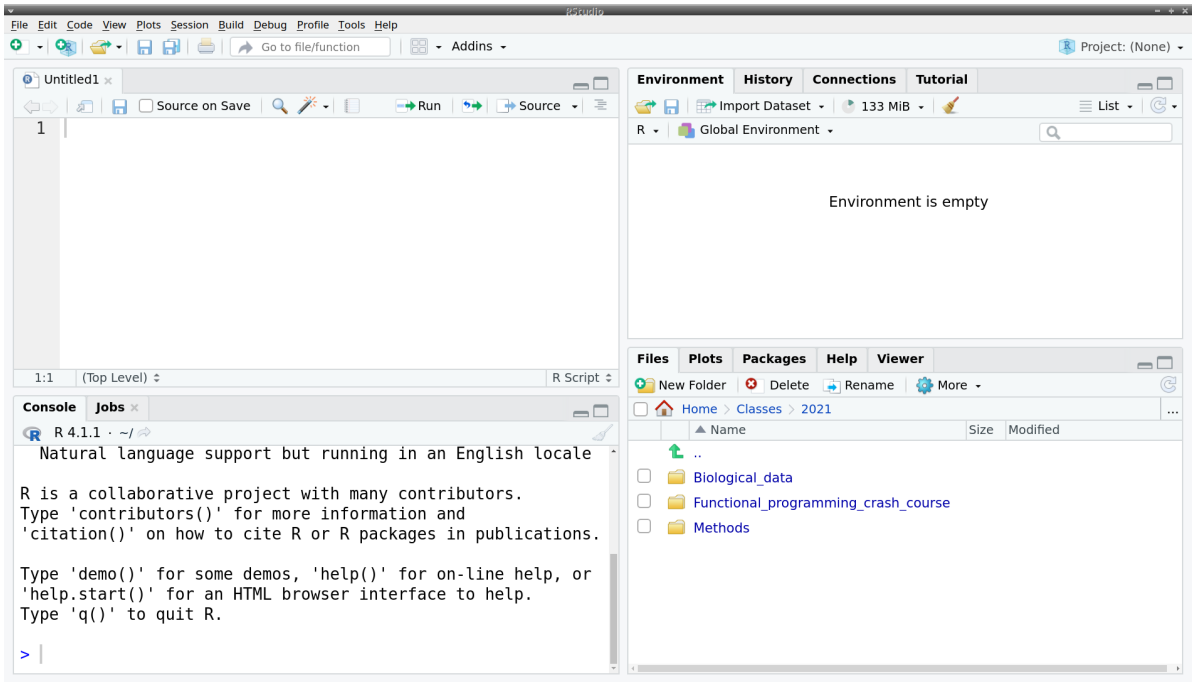


Figure 1.1: RStudio starting screen

1.3.1 A Simple Calculation

Even if you don't know R, you can start by typing some simple calculations into the console. The `>` symbol indicates that R is waiting for you to type something. Click on the console, type `2 + 2`, hit Enter (or Return on a Mac), and you should see R produce the right answer:

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

Now, press the Up arrow on the keyboard. You will notice that the `2 + 2` you typed before shows up. You can use the Up and Down arrows to go back and forth through past things you have typed, which can save a lot of repetitive typing when you are trying things out. You can change the text in these historical commands: change the `2` to a `3` and press Enter (Return, on a Mac) again:

```
> 2 + 3
[1] 5
>
```

(The `[1]` at the beginning of the result just means that the following number is at position 1 of a vector. In this case, the vector only has one element, but when R needs to print out a long vector, it splits it into into multiple lines tells you at the beginning of each line what position you are at.)

Before going deeper into R programming, we need to discuss a few things to enable you to get around R and RStudio more easily.

1.3.2 Writing R scripts

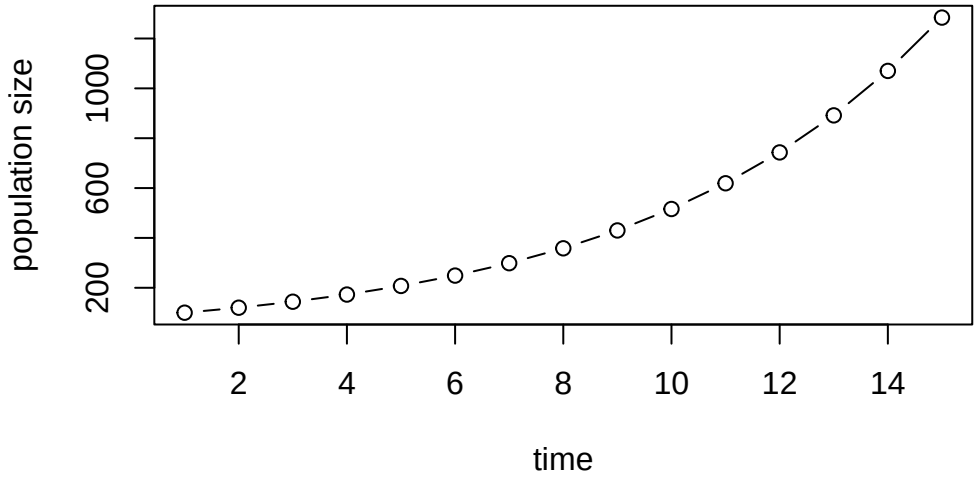
The upper-right part of RStudio is a simple text editor where you can write R code. But, instead of having to enter it one line at a time as we did in the console above, you can string long sequences of R instructions together that build on one another. You can save such a text file (Ctrl-S on Windows; Cmd-S on a Mac), giving it an appropriate name. It is then known as an R *script*, a text file containing R code that can be run from within R.

As an example, enter the following code. Do not worry about how or why it works just yet. It is a simple simulation and visualization of exponential population growth:

```
time <- 1:15
growthRate <- 1.2
popSize <- rep(100, times = length(time))
```

```
for (i in 1:(length(time) - 1)) {
  popSize[i + 1] <- popSize[i] * growthRate
}
plot(time, popSize, ylab = "population size", type = "b")
```

After having typed this, highlight all lines (either with a mouse, or by pressing Ctrl-A on Windows / Cmd-A on a Mac, or by using the arrow keys while holding the Shift key down). Then, to send these instructions to R for processing, press Ctrl-Enter (Windows) or Cmd-Return (Mac). If all went well, the lower right screen section should have jumped to the **Plots** panel, showing the following graph:



1.3.3 Setting the Working Directory

When you ask R to run a program or load data from a file, it needs to know where to find the file. Unless you specify the complete path to the file on your machine, it assumes that file names are relative to what is called the "working directory".

The first thing you should do when starting a project is to create a directory (folder) on your computer to store all the files related to the project, and then tell R to set the working directory to that location.

The R function `setwd("/path/to/directory")` is used to set the working directory, where you substitute in the actual path and directory name in place of `path/to/directory`. In turn, `getwd()` tells you what the current working directory is. The path can be found using File Explorer on a Windows PC, but you will need to change backslashes to forward slashes (\ to /). On a Mac, you can find the path by selecting a folder and choosing File > Get Info (the path is under "Where:").

There is also a convenient graphical way to set the working directory in RStudio. In the **Files** panel, you can navigate around the computer's file space. You can do this either in the panel itself, or using the ellipsis (…) to bring up the system-standard file browser. Looking around there does *not* immediately set the working directory, but you can set it by clicking the "More" button and choosing "Set as Working Directory".

> **i** Note
>
> It is worth repeating: **finding the appropriate directory in the Files panel is not enough.** It will not set the working directory automatically. You need to actually choose "Set as Working Directory" by clicking on it.

You may have also noticed that you don't actually need to type `getwd()`: the RStudio Console panel shows the current working directory below the word "Console".

1.3.4 Packages

One of the primary reasons ecologists use R is the availability of hundreds of free, user-contributed pieces of software, called packages. Packages are generally created by people who wanted to solve a particular problem for their own research and then realized that other people might find their code useful. Take a moment to browse the packages available on the main R site:

http://cran.r-project.org/web/packages/

To install a package, you take its name, put it in quotes, and put it in between the parentheses of `install.packages()`. For example, to install the package `tidyverse` (which we will be relying on later), you type:

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

and press Enter (Return, on a Mac). Note that some packages can take quite a while to install. If you are installing `tidyverse` for instance, it could take anywhere between five minutes to an hour (!) depending on your computer setup. This is normal, and the good news is that you only need to do this once on a computer. Once the package is installed, it will stick around.

However, to actually use a previously installed package in an R session, you need to load it from your disk directly into the computer's memory. That can be done like this:

```
library(tidyverse)
```

Note the *lack of* quotation marks when loading a package.

RStudio also makes package management a bit easier. In the **Packages** panel (top line of lower right portion of the screen) you can see a list of all installed packages. You can also load and unload packages simply by checking a checkbox, and you can install new packages using a graphical interface (although you will still need to know the name of the package you want to install).

1.4 Additional Reading

R:

- R website
- CRAN package index

RStudio:

- RStudio documentation

1.5 Exercises

1. Create a folder named `analysis_of_biological_data` on your computer and set it as the working directory in R. Make certain that the working directory has indeed been set properly.
2. Use the RStudio file browser to set the working directory somewhere else on your hard drive, and then set it back to the `analysis_of_biological_data` folder you created earlier. Make sure it is being set properly at each step.
3. Install an R package called `vegan`, using `install.packages` as discussed in Section 1.3.4. (The `vegan` package contains various utilities for community ecology.)
4. Load the `vegan` package invoking `library(vegan)`. Afterwards, try unloading and then loading the `vegan` package again, using the **Packages** panel in RStudio this time.

2 R programming basics

2.1 Using R as a calculator

As we have seen before, R can be used as a glorified pocket calculator. Elementary operations work as expected: `+` and `-` are symbols for addition and subtraction, while `*` and `/` are multiplication and division. Thus, we can enter things such as

```
3 * 4 - 6 / 2 + 1
```

in the console, and press Enter (Return, on a Mac) to get the result, 10. One even has exponentiation, denoted by the symbol `^`. To raise 2 to the 5th power), we enter

```
2^5
```

and obtain the expected 32. Furthermore, one is not restricted to integers. It is possible to calculate with fractional numbers:

```
1.62 * 34.56
```

whose result is 55.9872. **Note:** in line with Anglo-Saxon tradition, R uses decimal points instead of decimal commas. Entering `1,62 * 34,56` will throw an error.

R also has many basic mathematical functions built into it. For example, `sqrt()` is the square root function; `cos()` is the cosine function, `log()` is the (natural) logarithm, `exp()` is the exponential function, and so on. The following tables summarize the symbols for various arithmetic operations and basic mathematical functions built into R:

Symbol	Meaning	Example	Form in R
`+`	addition	$2 + 3$	`2 + 3`
`-`	subtraction	$5 - 1$	`5 - 1`
`*`	multiplication	$2 \cdot 6$	`2 * 6`
`/`	division	$9 / 5$	`9 / 5`
`^`	raise to power	3^2	`3 ^ 2`

Function	Meaning	Example	Form in R
`log()`	natural log	$\log(4)$	`log(4)`
`exp()`	exponential	e^4	`exp(4)`
`sqrt()`	square root	$\sqrt{4}$	`sqrt(4)`
`log2()`	base-2 log	$\log_2(4)$	`log2(4)`
`log10()`	base-10 log	$\log_{10}(4)$	`log10(4)`
`sin()`	sine (radians!)	$\sin(4)$	`sin(4)`
`abs()`	absolute value	$\|-4\|$	`abs(-4)`

Expressions built from these basic blocks can be freely combined. Try to calculate $3^{\log(4)}$ - $\sin(e^2)$ for instance. To do so, we simply type

```
3^log(4) - sin(exp(2))
```

and press Enter to get the result, 3.692108. Now obtain $e^{1.3}(4 - \sin(/3))$. Notice the parentheses enclosing 4 - sin(/3). This means, as usual, that this expression is evaluated first, before any of the other computations. It can be implemented in R the same way, by using parentheses:

```
exp(1.3) * (4 - sin(3.14159 / 3))
```

Note also that you *do* need to indicate the symbol for multiplication between closing and opening parentheses: omitting this results in an error. Try it: entering `exp(1.3)(4 - sin(3.14159/3))` instead of `exp(1.3)*(4 - sin(3.14159/3))` throws an error message. Also, be mindful that `exp(1.3)*(4 - sin(3.14159/3))` is not the same as `exp(1.3)*4 - sin(3.14159/3)`. This is because multiplication takes precedence over addition and subtraction, meaning that multiplications and divisions are performed first, and additions/subtractions get executed only afterwards—unless, of course, we override this behaviour with parentheses. In general, whenever you are uncertain about the order of execution of operations, it can be useful to explicitly use parentheses, even if it turns out they aren't really necessary. For instance, you might be uncertain whether `3 * 6 + 2` first multiplies 3 by 6 and then adds 2 to the result, or if it first adds 2 to 6 and then multiplies that by 3. In that case, if you want to be absolutely sure that you perform the multiplication first, just write `(3 * 6) + 2`, explicitly indicating with the parentheses that the multiplication should be performed first—even though doing so would not be strictly necessary in this case.

Incidentally, you do not need to type out `3.14159` to approximate in the mathematical expressions above. R has a built-in constant, `pi`, that you can use instead. Therefore, `exp(1.3)*(4 - sin(pi/3))` produces the same result as our earlier `exp(1.3)*(4 - sin(3.14159/3))`.

Another thing to note is that the number of spaces between various operations is irrelevant. `4*(9-6)` is the same as `4*(9 - 6)`, or `4 * (9 - 6)`, or, for that matter, `4  * (9-  6)`.

To the machine, they are all the same—it is only us, the human users, who might get confused by that last form…

It is possible to get help on any function from the system itself. Type either `help(asin)` or the shorter `?asin` in the console to get information on the function `asin`, for instance. Whenever you are not sure how to use a certain function, just ask the computer.

2.2 Variables and types

2.2.1 Numerical variables and variable names

You can assign a value to a named variable, and then whenever you call on that variable, the assigned value will be substituted. For instance, to obtain the square root of 9, you can simply type `sqrt(9)`; or you can assign the value 9 to a variable first:

```
x <- 9
sqrt(x)
```

This will calculate the square root of `x`, and since `x` was defined as 9, we get `sqrt(9)`, or 3.

The name for a variable can be almost anything, but a few restrictions apply. First, the name must consist only of letters, numbers, the period (`.`), and the underscore (`_`) character. Second, the variable's name cannot start with a number or an underscore. So `one_result` or `one.result` are fine variable names, but `1_result` or `_one_result` are not. Similarly, the name `crowns to $` is not valid because of the spaces and the dollar (`$`) symbol, neither of which are numbers, letters, period, or the underscore.

Additionally, there are a few *reserved words* which have a special meaning in R, and therefore cannot be used as variable names. Examples are: `if`, `NA`, `TRUE`, `FALSE`, `NULL`, and `function`. You can see the complete list by typing `?Reserved`.

However, one can override all these rules and give absolutely any name to a variable by enclosing it in backward tick marks (`` ` ` ``). So while `crowns to $` and `function` are not valid variable names, `` `crowns to $` `` and `` `function` `` are! For instance, you could type

```
`crowns to $` <- 0.09 # Approximate SEK-to-USD exchange rate
my_money <- 123 # Assumed to be given in Swedish crowns
my_money_in_USD <- my_money * `crowns to $`
print(my_money_in_USD)
```

```
[1] 11.07
```

to get our money's worth in US dollars. Note that the freedom of naming our variables whatever we wish comes at the price of having to always include them between back ticks to refer to them. It is entirely up to you whether you would like to use this feature or avoid it; however, be sure to recognize what it means when looking at R code written by others.

Notice also that the above chunk of code includes *comments*, prefaced by the hash (`#`) symbol. Anything that comes after the hash symbol on a line is ignored by R; it is only there for other humans to read.

> ⚠ Warning
>
> The variable `my_money_in_USD` above was defined in terms of the two variables `my_money` and `` `crowns to $` ``. You might be wondering: if we change `my_money` to a different value by executing `my_money <- 1000` (say), does `my_money_in_USD` also get automatically updated? **The answer is no:** the value of `my_money_in_USD` will remain unchanged. In other words, variables are not automatically recalculated the way Excel formula cells are. To recompute `my_money_in_USD`, you will need to execute `my_money_in_USD <- my_money * `crowns to $`` again. This leads to a recurring theme in programming: while assigning variables is convenient, it also carries some dangers, in case we forget to appropriately update them. In this course, we will be emphasizing a style of programming which avoids relying on (re-)assigning variables as much as possible.

2.2.2 Strings

So far we have worked with numerical data. R can also work with textual information. In computer science, these are called *character strings*, or just *strings* for short. To assign a string to a variable, one has to enclose the text in quotes. For instance,

```
s <- "Hello World!"
```

assigns the literal text `Hello World!` to the variable `s`. You can print it to screen either by just typing `s` at the console and pressing Enter, or typing `print(s)` and pressing Enter.

One useful function that works on strings is `paste()`, which makes a single string out of several ones (in computer lingo, this is known as *string concatenation*). For example, try

```
s1 <- "Hello"
s2 <- "World!"
message <- paste(s1, s2)
print(message)
```

```
[1] "Hello World!"
```

The component strings are separated by a space, but this can be changed with the optional `sep` argument to the `paste()` function:

```
message <- paste(s1, s2, sep = "")
print(message)
```

```
[1] "HelloWorld!"
```

This results in `message` becoming `HelloWorld!`, without the space in between. Between the quotes, you can put any character (including nothing, like above), which will be used as a separator when merging the strings `s1` and `s2`. So specifying `sep = "-"` would have set `message` equal to `Hello-World!` (try it out and see how it works).

It is important to remember that quotes distinguish information to be treated as text from information to be treated as numbers. Consider the following two variable assignments:

```
a <- 6.7
b <- "6.7"
```

Although they look superficially similar, `a` is the number 6.7 while `b` is the string "6.7", and the two are not equal! For instance, executing `2 * a` results in 13.4, but `2 * b` throws an error, because it does not make sense to multiply a bunch of text by 2.

2.2.3 Logical values

Let us type the following into the console, and press Enter:

```
2 > 1
```

```
[1] TRUE
```

We are asking the computer whether 2 is larger than 1. And it returns the answer: `TRUE`. By contrast, if we ask whether two is less than one, we get `FALSE`:

```
2 < 1
```

```
[1] FALSE
```

Similar to "greater than" and "less than", there are other logical operations as well, such as "greater than or equal to", "equal to", "not equal to", and others. The table below lists the most common options.

Symbol	Meaning	Example in R	Result
`<`	less than	`1 < 2`	`TRUE`
`>`	greater than	`1 > 2`	`FALSE`
`<=`	less than or equal	`2 <= 5.3`	`TRUE`
`>=`	greater than or equal	`4.2 >= 3.6`	`TRUE`
`==`	equal to	`5 == 6`	`FALSE`
`!=`	not equal to	`5 != 6`	`TRUE`
`!`	not	`!FALSE`	`TRUE`
`%in%`	is element of set	`2 %in% c(1, 2, 3)`	`TRUE`

The `==` and `!=` operators can also be used with strings: `"Hello World" == "Hello World!"` returns `FALSE`, because the two strings are not exactly identical, differing in the final exclamation mark. Similarly, `"Hello World" != "Hello World!"` returns `TRUE`, because it is indeed true that the two strings are unequal.

Logical values can either be `TRUE` or `FALSE`, with no other options.[1] This is in contrast with numbers and character strings, which can take on a myriad different values. Just like in the case of strings and numbers, logical values can be assigned to variables:

```
lgl <- 3 > 4 # Since 3 > 4 is FALSE, lgl will be assigned FALSE
print(!lgl) # lgl is FALSE, so !lgl ("not lgl") will be TRUE
```

```
[1] TRUE
```

The function `ifelse` takes advantage of logical values, doing different things depending on whether some condition is `TRUE` or `FALSE` ("*if* the condition is true *then* do something, *else* do some other thing"). It takes three arguments: the first is a condition, the second is the expression that gets executed only if the condition is true, and the third is the expression that executes only if the condition is false. To illustrate its use, we can apply it in a program that simulates a coin toss. R will generate n random numbers between 0 and 1 by invoking `runif(n)`. Here `runif` is a shorthand for "random-uniform", randomly generated numbers from a uniform distribution between 0 and 1. The function call `runif(1)` therefore produces a single random number, and we can interpret values less than 0.5 as having tossed heads, and other values as having tossed tails. The following lines implement this:

[1] Technically, there is a third option: a logical value could be equal to `NA`, indicating missing data.

```
toss <- runif(1)
coin <- ifelse(toss < 0.5, "heads", "tails")
print(coin)
```

```
[1] "heads"
```

This time we happened to have tossed heads, but try re-running the above three lines over and over again, to see that the results keep coming up at random.

2.3 Vectors

A *vector* is simply a sequence of variables of the same type. That is, the sequence may consist of numbers *or* strings *or* logical values, but one cannot intermix them. The `c()` function will create a vector in the following way:

```
x <- c(2, 5, 1, 6, 4, 4, 3, 3, 2, 5)
```

This is a vector of numbers. If, after entering this line, you type `x` or `print(x)` and press Enter, all the values in the vector will appear on screen:

```
x
```

```
 [1] 2 5 1 6 4 4 3 3 2 5
```

What can you do if you want to display only the third entry? The way to do this is by applying brackets:

```
x[3]
```

```
[1] 1
```

Never forget that vectors and its elements are simply variables! To show this, calculate the value of `x[1] * (x[2] + x[3])`, but before pressing Enter, guess what the result will be. Then check if you were correct. You can also try typing `x * 2`:

```
x * 2
```

```
 [1]  4 10  2 12  8  8  6  6  4 10
```

What happened? Now you performed an operation on the vector as a whole, i.e., you multiplied each element of the vector by two. Remember: you can perform all the elementary operations on vectors as well, and then the result will be obtained by applying the operation on each element separately.

Certain functions are specific to vectors. Try `mean(x)` and `var(x)` for instance (if you are not sure what these do, just ask by typing `?mean` or `?var`). Some others to try: `max`, `min`, `length`, and `sum`.

One can quickly generate vectors of sequences of values, using one of two ways. First, the notation `1:10` generates a vector of integers ranging from 1 to 10, in steps of 1. (Similarly, `2:7` generates the same vector as `c(2, 3, 4, 5, 6, 7)`, and so on). Second, the function `seq()` generates sequences, starting with the first argument, ending with the last, in steps defined by an optional `by` argument. So calling

```
seq(0, 10, by = 0.1)
```

```
  [1]  0.0  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1.0  1.1  1.2  1.3  1.4
 [16]  1.5  1.6  1.7  1.8  1.9  2.0  2.1  2.2  2.3  2.4  2.5  2.6  2.7  2.8  2.9
 [31]  3.0  3.1  3.2  3.3  3.4  3.5  3.6  3.7  3.8  3.9  4.0  4.1  4.2  4.3  4.4
 [46]  4.5  4.6  4.7  4.8  4.9  5.0  5.1  5.2  5.3  5.4  5.5  5.6  5.7  5.8  5.9
 [61]  6.0  6.1  6.2  6.3  6.4  6.5  6.6  6.7  6.8  6.9  7.0  7.1  7.2  7.3  7.4
 [76]  7.5  7.6  7.7  7.8  7.9  8.0  8.1  8.2  8.3  8.4  8.5  8.6  8.7  8.8  8.9
 [91]  9.0  9.1  9.2  9.3  9.4  9.5  9.6  9.7  9.8  9.9 10.0
```

creates a vector of numbers ranging from 0 to 10, in steps of 0.1.

Just as one can create a vector of numerical values, it is also possible to create a vector of character strings of logical values. For example:

```
stringVec <- c("I am the first string", "I am the second", "And I am the 3rd")
```

Now `stringVec[1]` is simply equal to the string `"I am the first string"`, `stringVec[2]` is equal to `"I am the second"`, and so on. Similarly, defining

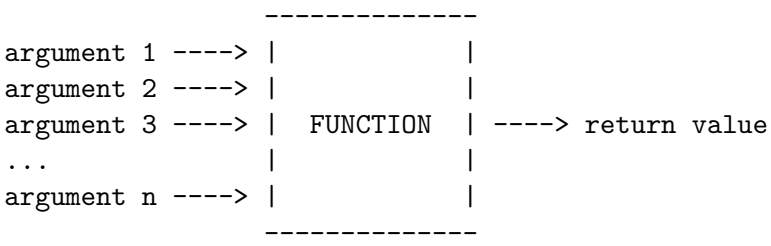
```
logicVec <- c(TRUE, FALSE, TRUE, TRUE)
```

gives a vector whose second entry, `logicVec[2]`, is equal to `FALSE`, and its other three entries are `TRUE`.

2.4 Functions

A *function* in R can be thought of as a black box which receives *inputs* and, depending on those inputs, produces some *output*. Vending machines provide a good working model of what a "function" is in computer science: depending on the inputs they receive (in the form of coins of various denomination, plus the buttons you press for a particular item) they give you some output (Mars bars, Coke, and the like). It's just that computer scientists like to refer to the inputs as "function arguments" or simply "arguments" instead of coins, and to the output as the "return value" instead of Red Bull. Arguments are often also referred to as "parameters" to the function.

We have already seen some functions at work in R: `sqrt` and `log` are functions, but so are `setwd` (which, as you may recall, will set your working directory) and `library` (which loads R packages). The general workings of a function are illustrated below:



When you ask a function to do something, you're *calling* the function. The arguments of functions are always enclosed in parentheses. For example, executing `sqrt(9)`, calls the built-in square root function. Its argument (or input, or parameter) is 9, and its return value is the square root of 9, which is 3.

2.4.1 User-defined functions

Thus far, we have been using many built-in functions in R, such as `exp()`, `log()`, `sqrt()`, `setwd()`, and others. However, it is also possible to define our own functions, which can then be used just like any built-in function. The way to do this is to use the `function` keyword, followed by the function's arguments in parentheses, and then the R code comprising the function's body enclosed in curly braces `{}`. For example, here is a function which calculates the area of a circle with radius `r`:

```
circleArea <- function(r) {
  area <- r^2 * pi
  return(area)
}
```

The function implements the formula that the area of a circle is equal to times its radius squared. The `return` keyword determines what result the function will output when it finishes executing. In this case, the function returns the value of `area` that is created within the function. After running the above lines, the computer now "knows" the function. Calling `circleArea(3)` will, for example, calculate the area of a circle with radius 3, which is approximately 28.27433.

One can define functions with more than one argument. For instance, here is a function that calculates the volume of a cylinder with radius `r` and height `h`:

```
cylinderVol <- function(r, h) {
  baseArea <- circleArea(r)
  volume <- baseArea * h
  return(volume)
}
```

Here we used the fact that the volume of a cylinder is the area of its base circle, times its height. Notice also that we made use of our earlier `circleArea` function within the body of `cylinderVol`. While this was not a necessity and we could have simply written `volume <- r^2 * pi * h` above, this is generally speaking good practice: by constructing functions to solve smaller problems, you can write slightly more complicated functions which make use of those simpler ones. Then, you will be able to write even more complex functions using the slightly more complex ones in turn—and so on. We will discuss this principle in more detail below, in Section 2.4.3.

One very important property of functions is that any variables defined within them (such as `volume` above) are *local* to that function. This means that they are not visible from outside: even after calling the function, the variable `volume` will not be accessible to the rest of the program, despite the fact that it was declared in the function. This helps us create programs with modular structure, where functions operate as black boxes: we can use them without looking inside.

When calling a function, it is optional but possible to name the arguments explicitly. This means that calling `circleArea(3)` is the same as calling `circleArea(r = 3)`, and calling `cylinderVol(2, 3)` is the same as calling `cylinderVol(r = 2, h = 3)`. Even more is true: since naming the arguments removes any ambiguity about which argument is which, one may even call `cylinderVol(h = 3, r = 2)`, with the arguments in reverse order, and this will still be equivalent to `cylinderVol(2, 3)`. As mentioned, naming arguments this way is optional, but it can be useful to do so, because it can increase the clarity of our programs. To give an example from a built-in function in R, take `rep(5, 3)`. Does this function create a vector with 5 entries, each equal to 3, or does it make a vector with 3 entries, each equal to 5? While reading the documentation (or simply executing these two function calls and comparing the outputs) reveals that it is the latter, one can clarify this easily, because the second argument of `rep()` is called `times`, as seen from reading the help after typing `?rep`. We can then write

`rep(5, times = 3)`, which is now easy to interpret: it is a vector with the number 5 repeated 3 times.

```
rep(5, times = 3)
```

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

One may even define default values for one or more of the arguments to any function. If defaults are given, the user does not even have to specify the value for that argument. It will then automatically be set to the default value instead. For example, one could rewrite the `cylinderVol()` function to specify default values for `r` and `h`. Making these defaults be 1 means we can write:

```
cylinderVol <- function(r = 1, h = 1) {
  baseArea <- circleArea(r)
  volume <- baseArea * h
  return(volume)
}
```

If we now call `cylinderVol()` without specifying arguments, the defaults will be substituted for `r` and `h`. Since both are equal to 1, the cylinder volume will simply be (about 3.14159), which is the result we will get back. Alternatively, if we call `cylinderVol(r = 2)`, then the function returns 4 (approximately 12.56637), because the default value of 1 is substituted in place of the unspecified height argument `h`. Importantly, if we *don't* define default values and yet omit to specify one or more of those parameters, we get back an error message. For example, our earlier `circleArea` function had no default value for its argument `r`, so leaving it unspecified throws an error:

```
circleArea()
```

```
Error in circleArea(): argument "r" is missing, with no default
```

2.4.2 Naming rules for functions and the concept of syntactic sugar

The rules for naming functions is the same as for naming variables. A valid function name is a combination of letters, numbers, and underscores (_), as long as the first character is not a number or underscore. Additionally, a function's name cannot be one of the reserved words (see `?Reserved`). Just like in the case of variables, one can override this and give any name whatsoever to functions if one encloses the name between back ticks. So while `crowns to $` is not a valid function name, `` `crowns to $` `` is.

One thing to know about R is that even elementary operations are treated as function calls internally. When we write down even something as innocuous as `2 + 5`, what really happens is that R calls the *function* called `+`, with arguments 2 and 5. In fact, we can write it that way too: `2 + 5` is completely equivalent to writing `` `+`(2, 5) ``. Note the back ticks around `` `+` ``: these are required because `+` is not a letter, number, or underscore. Whenever we write down `2 + 5`, the system internally converts it into `` `+`(2, 5) `` first, and then proceeds with the execution. Thus, the fact that we can add two numbers by writing `2 + 5` is just a convenience, a way of entering addition in a way that we tend to be more used to. Such constructions have a name in computer science: they are called *syntactic sugar*. Writing `2 + 5` is just syntactic sugar for the actual internal form `` `+`(2, 5) ``, because the latter would be stranger to write. Of course, the same holds for all other elementary operations: `` `-` ``, `` `*` ``, `` `/` ``, and `` `^` `` are also functions in R. This means that, e.g., writing `` `-`(`^`(2, 3), `*`(4, 2)) `` is equivalent to `2^3 - 4 * 2`.

Another example for the fact that internally R treats operations as functions is the subsetting of vectors or matrices. As we have learned, given the vector `x`, typing `x[3]` will extract the third entry of the vector. In fact, this is again syntactic sugar for easier use. Internally, an expression such as `x[3]` is actually interpreted as `` `[`(x, 3) ``. The function `` `[` `` (note the back ticks, which are necessary due to the fact that the symbol `[` is not a letter, number, or underscore) takes two arguments: a vector, and the index (or indices) which we request from that vector.

While generally speaking, one would never actually want to type `` `[`(x, 3) `` instead of `x[3]` (the reason we have the syntactic sugar is to make our lives easier!), there are situations where being aware of these details of the internal workings of R can be helpful. We will see an example later in this chapter.

2.4.3 Function composition

A function is like a vending machine: we give it some input(s), and it produces some output. The output itself may then be fed as input to another function—which in turn produces an output, which can be fed to yet another function, and so on. Chaining functions together in this manner is called the *composition* of functions. For example, we might need to take the square root of a number, then calculate the logarithm of the output, and finally, obtain the cosine of the result. This is as simple as writing `cos(log(sqrt(9)))`, if the number we start with is 9. More generally, one might even define a new function (let us call it `cls()`, after the starting letters of `cos`, `log`, and `sqrt`) like this:

```
cls <- function(x) {
  return(cos(log(sqrt(x))))
}
```

A remarkable property of composition is that the composed function (in this case, `cls`) is in many ways just like its constituents: it is also a black box which takes a single number as input and produces another number as its output. Putting it differently, if one did not know that the function `cls()` was defined by me manually as the composition of three more "elementary" functions, and instead claimed it was just another elementary built-in function in R, there would be no way to tell the difference just based on the behaviour of the function itself. The composition of functions thus has the important property of *self-similarity*: if we manage to solve a problem through the composition of functions, then that solution itself will behave like an "elementary" function, and so can be used to solve even more complex problems via composition—and so on.

If we conceive of a program written in R as a large lego building, then one can think of functions as the lego blocks out of which the whole construction is made. Lego pieces are designed to fit well together, one can always combine them in various ways. Furthermore, any combination of lego pieces itself behaves like a more elementary lego piece: it can be fitted together with other pieces in much the same way. Thus, the composition of functions is analogous to building larger lego blocks out of simpler ones. Remarkably, just as the size of a lego block does not hamper our ability to stick them together, the composability of functions is retained regardless of how many more elementary pieces each of them consist of. Thus, the composition of functions is an excellent way (some claim it is *the* way) to handle the complexity of large software systems.

2.4.4 Function piping

One problem with composing many functions together is that the order of application must be read backwards. An expression such as `sqrt(sin(cos(log(1))))` means: "take the square root of the sine of the cosine of the natural logarithm of 1". But it is more convenient for the human brain to think of it the other way round: we first take the log of 1, then the cosine of the result, then the sine of what we got, and finally the square root. The problem of interpreting composed functions gets more difficult when the functions have more than one argument. Even something as relatively simple as

```
exp(mean(log(seq(-3, 11, by = 2)), na.rm = TRUE))
```

```
[1] 4.671655
```

may cause one to stop and have to think about what this expression actually does—and it only involves the composition of four simple functions. One can imagine the difficulties of having to parse the composition of dozens of functions in this style.

The above piece of R code generates the numeric sequence -3, -1, 1, ..., 11 (jumping in steps of 2), and computes their geometric mean. To do so, it takes the logarithms of each value, takes their mean, and finally, exponentiates the result back. The problem is that the logarithm of

negative numbers does not exist (more precisely, they are not real numbers), and therefore, `log(-3)` and `log(-1)` both produce undefined results. Thus, when taking the `mean` of the logarithms, we must remove any such undefined values. This can be accomplished via an extra argument to `mean`, called `na.rm` ("NA-remove"). By default, this is set to `FALSE`, but by changing it to `TRUE`, undefined values are simply ignored when computing the mean. For example `mean(c(1, 2, 3, NA))` returns `NA`, because of the undefined entry in the vector; but `mean(c(1, 2, 3, NA), na.rm = TRUE)` returns 2, the result one gets after discarding the `NA` entry.

All the above is difficult to see when looking at the expression

```
exp(mean(log(seq(-3, 11, by = 2)), na.rm = TRUE))
```

Part of the reason is the awkward "backwards order" of function applications, and that it is hard to see which function the argument `na.rm = TRUE` belongs to. Fortunately, there is a simple operator in R called a *pipe* (written `%>%`), which allows one to write the same code in a more streamlined way. The pipe was originally provided by the `magrittr` package,[2] but invoking `tidyverse` will also load it automatically:

```
library(tidyverse)
```

The pipe allows one to write function application in reverse order (first the argument and then the function), making the code more transparent. Formally, `x %>% f()` is equivalent to `f(x)` for any function `f`. For example, `sqrt(9)` can also be written `9 %>% sqrt()`. Thus, `sqrt(sin(cos(log(1))))` can be written as `1 %>% log() %>% cos %>% sin() %>% sqrt()`, which reads straightforwardly as "start with the number 1; *then* take its log; *then* take the cosine of the result; *then* take the sine of that result; and *then*, finally, take the square root to obtain the final output". In general, it helps to pronounce `%>%` as "then".

The pipe also works for functions with multiple arguments. In that case, `x %>% f(y, ...)` is equivalent to `f(x, y, ...)`. That is, the pipe refers to the function's first argument (though it is possible to override this). Instead of `mean(log(seq(-3, 11, by = 2)), na.rm = TRUE)`, we can therefore write:

```
seq(-3, 11, by = 2) %>%
  log() %>%
  mean(na.rm = TRUE) %>%
  exp()
```

[2] As of R 4.1.0, the R language also supports a native, built-in pipe operator `|>` as well. The package name `magrittr` is an allusion to Belgian surrealist artist René Magritte (1898-1967) because of his famous painting La trahison des images.

```
[1] 4.671655
```

This is fully equivalent to the traditional form, but is much more readable, because the functions are written in the order in which they actually get applied. Moreover, even though the program is built only from the composition of functions, it reads straightforwardly as if it was a sequence of imperative instructions: we start from the vector of integers `c(-3, -1, 1, 3, 5, 7, 9, 11)`; *then* we take the logarithm of each; *then* we take their average, discarding any invalid entries (produced in this case by taking the logarithm of negative numbers); and *then*, finally, we exponentiate back the result to obtain the geometric mean.

2.5 Exercises

1. Which of the variable names below are valid, and why?

   - `first.result.of_computation`
   - `2nd.result.of_computation`
   - `dsaqwerty`
   - `dsaq werty`
   - `` `dsaq werty` ``
   - `break`
   - `is this valid?...`
   - `` `is this valid?...` ``
   - `is_this_valid?...`

2. Create a vector called `z`, with entries 1.2, 5, 3, 13.7, 6.66, and 4.2 (in that order). Then, by applying functions to this vector, obtain:

   - Its smallest entry.
   - Its largest entry.
   - The sum of all its entries.
   - The number of entries in the vector.
   - The vector's entries sorted in increasing order (Hint: look up the help for the built-in function `sort`).
   - The vector's entries sorted in decreasing order.
   - The product of the fourth entry with the difference of the third and sixth entries. Then take the absolute value of the result.

3. Define a vector of strings, called `s`, with the three entries `"the fat cat"`, `"sat on"`, and `"the mat"`.

   - Combine these three strings into a single string, and print it on the screen. (Hint: look up the help for the `paste` function, in particular its `collapse` argument.)

- Reverse the entries of `s`, so they come in the order `"the mat"`, `"sat on"`, and `"the fat cat"`. (Hint: check out the `rev` function.) Then merge the three strings again into a single one, and print it on the screen.

4. Assume you have a population of some organism in which one given allele of some gene is the only one available in the gene pool. If a new mutant organism with a different, selectively advantageous allele appears, it would be reasonable to conclude that the new allele will fix in the population and eliminate the original one over time. This, however, is not necessarily true, because a very rare allele might succumb to being eliminated by chance, regardless of how advantageous it is. According to Motoo Kimura's famous formula,[3] the probability of such a new allele eventually fixing in the population is given as:

$$P = \frac{1 - \mathrm{e}^{-s}}{1 - \mathrm{e}^{-2Ns}}$$

Here P is the probability of eventual fixation, s is the selection differential (the degree to which the new allele is advantageous over the original one), and N is the (effective) population size.

- Write a function that implements this formula. It should take the selection differential s and the population size N as parameters, and return the fixation probability as its result.
- A selection differential of 0.5 is very strong (though not unheard of). What is the likelihood that an allele with that level of advantage will fix in a population of 1000 individuals? Interpret the result.

5. A text is *palindromic* if it reads backwards the same as it reads forwards. For example, "racecar", "deified", and "rotator" are all palindromic words. Assume that you are given a word in all lowercase, broken up by characters. For instance, you could be given the vector `c("r", "a", "c", "e", "c", "a", "r")` (a palindrome) or `c("h", "e", "l", "l", "o")` (not a palindrome).

- Write a function which checks whether the vector encodes a palindromic text. The function should return `TRUE` if the text is a palindrome, and `FALSE` otherwise. (Hint: reverse the text, collapse both the original and the reversed vectors into single strings, and then compare them using logical equality.)
- Modify the function to allow for both upper- and lowercase text, treating case as irrelevant (i.e., `"A"` is treated to be equal to `"a"` when evaluating whether the text is palindromic). One simple way to do this is to convert each character of the text into uppercase (or lowercase; it doesn't matter which), and use this standardized text

[3] See, e.g., John H. Gillespie (2004) *Population Genetics: A Concise Guide*, Johns Hopkins University press, p. 93.

for reversing and comparing with. Look up the functions `toupper` and `tolower`, and implement this improvement in your palindrome checker function.

- If you haven't done so already: try to rewrite your function to rely as much on function composition as possible.

3 Reading tabular data from disk

3.1 The `tidyverse` package suite

A suite of R packages, sharing the same design philosophy, are collected under the name `tidyverse`. In case this is not yet installed on your computer, type

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

at the R console and press Enter. After making sure that the package is installed, you must load it. This is done via the function call

```
library(tidyverse)
```

```
-- Attaching packages --------------------------------------- tidyverse 1.3.1 --

v ggplot2 3.3.5     v purrr   0.3.4
v tibble  3.1.6     v dplyr   1.0.8
v tidyr   1.2.0     v stringr 1.4.0
v readr   2.1.2     v forcats 0.5.1

-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

As you see, eight packages are now loaded, called `ggplot2`, `tibble`, and so on. We will get to know these in more detail throughout the course.

There are actually even more packages that are part of the tidyverse. Typing and executing `tidyverse_packages()` will show all such packages. Of all these options, only eight are loaded by default when invoking `library(tidyverse)`. The others must be loaded separately. For example, `readxl` is a tidyverse package for loading Excel files in R. To use it, execute `library(readxl)`.

In general, it is a good idea to load all necessary packages at the top of your R script. There are two reasons for this. First, once you close RStudio, it forgets the packages, which do not get automatically reloaded after reopening RStudio. Second, often other users will run the scripts you write on their own computers, and they will not be able to do so unless the proper packages are loaded first. It is then helpful to others if the necessary packages are all listed right at the top, showing what is needed to run your program.

3.2 Reading tabular data

One of the packages loaded by default with `tidyverse` is called `readr`. This package contains tools for loading data files, and writing them to disk. To see how it works, download the files `Goldberg2010_data.csv`, `Goldberg2010_data.xlsx`, and `Smith2003_data.txt` from Lisam. Then set the working directory in RStudio to the folder where you have saved them (as a reminder, you can do this by executing `setwd(/path/to/files)`, where you should substitute in your own path in place of `/path/to/files`).

3.2.1 The CSV file format

Goldberg et al. (2010)[1] collected data on self-incompatibility in the family *Solanaceae* (nightshades). It contains a list of 356 species, along with a flag determining self-incompatibility status (0: self-incompatible; 1: self-compatible; 2-5: more complicated selfing scenarios). The data are in the file `Goldberg2010_data.csv`. This is a so-called *comma-separated value* (CSV) file, meaning that the different columns of the data are separated by commas. One can see this by viewing the file in any simple text editor. For example, this can be done in RStudio itself, by clicking on the file in the **Files** panel in the lower right part of the RStudio window, and then choosing the option "View file" (ignore the other option called "Import dataset…"). Having done this, a new tab opens in your editor panel (upper left region) where you should see something like the following:

```
Species,Status
Acnistus_arborescens,1
Anisodus_tanguticus,1
Atropa_belladonna,1
Brachistus_stramonifolius,1
Brugmansia_aurea,0
Brugmansia_sanguinea,0
Capsicum_annuum,1
Capsicum_baccatum,1
Capsicum_cardenasii,2
```

[1] Goldberg, E. E. et al. (2010). Species Selection Maintains Self-Incompatibility. *Science* 330, 493.

```
Capsicum_chacoense,1
```

And so on. As you can see, the first line (`Species,Status`) is actually an indicator of what the corresponding columns of data will contain: the first column has the species name, and the second one the numerical flag indicating self-compatibility status. The subsequent rows hold the actual data. Notice that the boundary between the columns is always indicated by a comma. This is what gave rise to the name "comma-separated value" (CSV) file.

The above raw format is not yet amenable to processing within R. To make it so, we first need to import the data. For comma-separated value files, there is a convenient function, `read_csv`, that makes this especially simple:[2]

```
read_csv("Goldberg2010_data.csv")
```

```
# A tibble: 356 x 2
   Species                   Status
   <chr>                      <dbl>
 1 Acnistus_arborescens           1
 2 Anisodus_tanguticus            1
 3 Atropa_belladonna              1
 4 Brachistus_stramonifolius      1
 5 Brugmansia_aurea               0
 6 Brugmansia_sanguinea           0
 7 Capsicum_annuum                1
 8 Capsicum_baccatum              1
 9 Capsicum_cardenasii            2
10 Capsicum_chacoense             1
# ... with 346 more rows
```

(We will interpret the output in the next subsection.) The above line loads the data, but does not save it into a variable. That is perfectly fine in case we immediately start performing operations on it via function composition (we will see many, many examples later on). However, in case we do want to assign the result to a variable, we can do so without problems. For instance, to put the table into the variable `dat`, we simply write:

```
dat <- read_csv("Goldberg2010_data.csv")
```

[2]Warning: there exists a similarly-named function called `read.csv` which is part of base R. It does much the same thing as `read_csv`; however, its use is far clunkier and less flexible. You can think of `read_csv` as a `tidyverse`-provided upgrade to the original `read.csv`. My recommendation is to stick with using just `read_csv`—it is simpler and at the same time more powerful than its predecessor.

3.2.2 The `tibble` data structure

Look at the output produced by `read_csv("Goldberg2010_data.csv")` above. You can mostly ignore the top part of that output—it simply provides information on how it interpreted the data it just read in. Instead, the interesting part starts with `A tibble: 356 x 2`. A *tibble* (or *data frame*[3]) is the R-equivalent of an Excel-style spreadsheets. In this case, it has 356 rows and 2 columns (hence the `356 x 2`). The simplest way to conceive of a tibble is as a collection of vectors, glued together side-by-side to form a table of data. Importantly, although each vector must consist of entries of the same type, as usual (e.g., they can be vectors of numbers, vectors of strings, or vectors of logical values), the different columns need not share types. For example, in the above table, the first column consists of character strings, but the second one consists of numerical values. This can be seen right below the header information. Below `Species`, you can see `<chr>`, which stands for "character string". Below `Status`, we have `<dbl>` which, confusing as it may look at first sight, refers simply to ordinary numbers.[4] In turn, columns comprising of logical values would have the tag `<lgl>` underneath them (in this case though, we don't have such a column). The point is that by looking at the type information below the header, you can see how R has interpreted each of the columns at a glance.

The fact that the individual columns are simply vectors can be made explicit, by relying on the `$`-notation. To access a given column of the table as a vector, we write the name of the table, followed by the `$` symbol, followed by the name of the column in question. For example, we can access the `Status` column from the `dat` table as a vector of numbers like this:

```
dat$Status
```

```
  [1] 1 1 1 1 0 0 1 1 2 1 1 1 1 1 1 1 2 1 1 1 1 0 0 1 1 1 1 0 4 0 0 0 1 1 1 0
 [38] 0 0 0 0 1 1 1 1 1 1 0 0 0 0 4 0 3 0 0 4 0 4 4 0 4 1 0 0 0 4 4 4 0 1 1 0 1
 [75] 1 1 1 0 0 1 1 1 1 1 1 1 0 1 2 1 1 1 1 1 1 2 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1
[112] 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 0 1 2 0 0 2 0 0 0 1 0 0 1 1 0 0 1 1 0 0 0
[149] 1 1 1 1 1 1 1 1 1 1 1 1 1 4 1 4 1 1 0 0 1 0 0 0 1 1 1 1 1 0 4 0 4 0 1 2 0
[186] 1 0 1 0 1 1 0 1 1 0 0 1 0 1 0 4 4 1 1 1 4 0 0 0 0 1 1 0 1 1 0 0 1 2 1 1 0
[223] 1 2 0 0 1 1 1 1 1 1 0 0 1 1 0 0 0 1 1 0 0 4 0 1 1 1 1 0 0 1 1 1 1 0 1 1 0
[260] 1 0 1 1 2 1 1 1 1 0 0 0 1 2 1 0 0 1 1 4 1 0 2 0 4 1 1 0 0 0 1 2 4 1 1 1 1
[297] 1 1 1 1 1 1 1 0 2 0 1 1 0 2 0 0 0 1 0 1 1 0 1 0 0 1 1 0 1 1 1 1 1 1 1 2 4
```

[3] There is a nuance of a difference between a data frame (which is a feature of base R) and a tibble (a `tidyverse` construct). The two are mostly equivalent, but tibbles offer some features that are absent from data frames, and omit certain things which data frames do but are usually not needed. Like with `read_csv` and `read.csv`, tibbles can be thought of as a slightly upgraded and more user-friendly version of data frames. You do not need to be overly concerned with the precise differences between the two; in this course, we will mostly be using tibbles anyway.

[4] The abbreviation `<dbl>` happens to stand for double-precision numerical value, a standard way of representing numbers on computers.

```
[334] 0 4 0 1 1 1 1 1 0 1 1 5 4 4 1 4 1 0 0 0 0 0 2
```

Here `dat$Status` is really just a vector, and can be treated as such. For example, to get the 9th entry of this vector, we can use the usual bracket notation:

```
dat$Status[9]
```

```
[1] 2
```

The result is an ordinary numerical value.

Finally, let us take one more look at the output again:

```
print(dat)
```

```
# A tibble: 356 x 2
   Species                   Status
   <chr>                      <dbl>
 1 Acnistus_arborescens           1
 2 Anisodus_tanguticus            1
 3 Atropa_belladonna              1
 4 Brachistus_stramonifolius      1
 5 Brugmansia_aurea               0
 6 Brugmansia_sanguinea           0
 7 Capsicum_annuum                1
 8 Capsicum_baccatum              1
 9 Capsicum_cardenasii            2
10 Capsicum_chacoense             1
# ... with 346 more rows
```

When displaying large tibbles, R will not dump all the data at you. Instead, it will display the first 10 rows, with a message indicating how many more rows remain (in our case, we have `...with 346 more rows` written at the end of the printout). The system is still aware of the other rows; it just does not show them. To get a full view of a tibble in a more digestible, spreadsheet-like style, one can use the `view` function. Try running `view(dat)` and see what happens!

3.2.3 The TSV file format

Another type of file is one where the columns are separated by tabulators instead of commas. These are called *tab-separated value* (TSV) files. An example is provided by the file associated with data from Smith et al. (2003).[5] The authors compiled a database of the body mass of mammals of the late Quaternary period. The data file is `Smith2003_data.txt`. Its rows are the different mammal species; its columns are: the species' native continent; whether the species is still alive or extinct; the order, family, genus, and species names; the base-10 log body mass; the actual body mass (in grams); and numbered references representing research papers which served as the source of the data.

Viewing the TSV file `Smith2003_data.txt` in its raw form begins something like the following:

```
AF  extant  Artiodactyla    Bovidae Addax   nasomaculatus   4.85    70000.3 60
AF  extant  Artiodactyla    Bovidae Aepyceros   melampus    4.72    52500.1 "63, 70"
AF  extant  Artiodactyla    Bovidae Alcelaphus  buselaphus  5.23    171001.5    "63, 70"
AF  extant  Artiodactyla    Bovidae Ammodorcas  clarkei 4.45    28049.8 60
AF  extant  Artiodactyla    Bovidae Ammotragus  lervia  4.68    48000   75
AF  extant  Artiodactyla    Bovidae Antidorcas  marsupialis 4.59    39049.9 60
AF  extinct Artiodactyla    Bovidae Antidorcas  bondi   4.53    34000   1
AF  extinct Artiodactyla    Bovidae Antidorcas  australis   4.6 40000   2
AF  extant  Artiodactyla    Bovidae Bos taurus  5.95    900000  -999
AF  extant  Artiodactyla    Bovidae Capra   walie   5   100000  -999
```

Since the file is tab- and not comma-separated, trying to load it using `read_csv` will not work correctly:

```
read_csv("Smith2003_data.txt")
```

```
Warning: One or more parsing issues, see `problems()` for details

Rows: 5730 Columns: 1
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): AF extant  Artiodactyla    Bovidae Addax   nasomaculatus   4.85    70000.3 60

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

[5] Smith, F. A. et al. (2003). Body mass of late Quaternary mammals. *Ecology* 84:3403-3403.

```
# A tibble: 5,730 x 1
   `AF\textant\tArtiodactyla\tBovidae\tAddax\tnasomaculatus\t4.85\t70000.3\t60`
   <chr>
 1 "AF\textant\tArtiodactyla\tBovidae\tAepyceros\tmelampus\t4.72\t52500.1\t\"63~
 2 "AF\textant\tArtiodactyla\tBovidae\tAlcelaphus\tbuselaphus\t5.23\t171001.5\t~
 3 "AF\textant\tArtiodactyla\tBovidae\tAmmodorcas\tclarkei\t4.45\t28049.8\t60"
 4 "AF\textant\tArtiodactyla\tBovidae\tAmmotragus\tlervia\t4.68\t48000\t75"
 5 "AF\textant\tArtiodactyla\tBovidae\tAntidorcas\tmarsupialis\t4.59\t39049.9\t~
 6 "AF\textinct\tArtiodactyla\tBovidae\tAntidorcas\tbondi\t4.53\t34000\t1"
 7 "AF\textinct\tArtiodactyla\tBovidae\tAntidorcas\taustralis\t4.6\t40000\t2"
 8 "AF\textant\tArtiodactyla\tBovidae\tBos\ttaurus\t5.95\t900000\t-999"
 9 "AF\textant\tArtiodactyla\tBovidae\tCapra\twalie\t5\t100000\t-999"
10 "AF\textant\tArtiodactyla\tBovidae\tCapra\tibex\t5\t100999.7\t65"
# ... with 5,720 more rows
```

As you can see, there is even a warning at the top about "One or more parsing issues", meaning that `read_csv` had a hard time reading in the file. Below the message, you can also see that the attempted read is a mess, with all data in the rows treated as being part of a single column.

Instead, to correctly read TSV files, one should use `read_tsv`:

```
read_tsv("Smith2003_data.txt")
```

```
# A tibble: 5,730 x 9
   AF    extant  Artiodactyla Bovidae Addax nasomaculatus `4.85` `70000.3` `60`
   <chr> <chr>   <chr>        <chr>   <chr> <chr>          <dbl>     <dbl> <chr>
 1 AF    extant  Artiodactyla Bovidae Aepy~ melampus        4.72    52500. 63, ~
 2 AF    extant  Artiodactyla Bovidae Alce~ buselaphus      5.23   171002. 63, ~
 3 AF    extant  Artiodactyla Bovidae Ammo~ clarkei         4.45    28050. 60
 4 AF    extant  Artiodactyla Bovidae Ammo~ lervia          4.68    48000  75
 5 AF    extant  Artiodactyla Bovidae Anti~ marsupialis     4.59    39050. 60
 6 AF    extinct Artiodactyla Bovidae Anti~ bondi           4.53    34000  1
 7 AF    extinct Artiodactyla Bovidae Anti~ australis       4.6     40000  2
 8 AF    extant  Artiodactyla Bovidae Bos   taurus          5.95   900000  -999
 9 AF    extant  Artiodactyla Bovidae Capra walie           5      100000  -999
10 AF    extant  Artiodactyla Bovidae Capra ibex            5      101000. 65
# ... with 5,720 more rows
```

This is now a neatly formatted table, with 9 columns as needed.

There is a problem though. This file does not contain a header—a row, which is the first in a data file, specifying the names of the various columns. (Recall that the first row of

`Goldberg2010_data.csv` contained not data, but the names of the columns.) Instead, the first row is itself part of the data. To override the default behavior of treating the first row as one of column names, one can use the `col_names = FALSE` option:

```
read_tsv("Smith2003_data.txt", col_names = FALSE)
```

```
# A tibble: 5,731 x 9
   X1    X2      X3           X4      X5         X6              X7     X8 X9
   <chr> <chr>   <chr>        <chr>   <chr>      <chr>        <dbl>  <dbl> <chr>
 1 AF    extant  Artiodactyla Bovidae Addax      nasomaculat~  4.85 7.00e4 60
 2 AF    extant  Artiodactyla Bovidae Aepyceros  melampus      4.72 5.25e4 63, ~
 3 AF    extant  Artiodactyla Bovidae Alcelaphus buselaphus    5.23 1.71e5 63, ~
 4 AF    extant  Artiodactyla Bovidae Ammodorcas clarkei       4.45 2.80e4 60
 5 AF    extant  Artiodactyla Bovidae Ammotragus lervia        4.68 4.8 e4 75
 6 AF    extant  Artiodactyla Bovidae Antidorcas marsupialis   4.59 3.90e4 60
 7 AF    extinct Artiodactyla Bovidae Antidorcas bondi         4.53 3.4 e4 1
 8 AF    extinct Artiodactyla Bovidae Antidorcas australis     4.6  4   e4 2
 9 AF    extant  Artiodactyla Bovidae Bos        taurus        5.95 9   e5 -999
10 AF    extant  Artiodactyla Bovidae Capra      walie         5    1   e5 -999
# ... with 5,721 more rows
```

The `col_names` argument is set by default to `TRUE`; in case we wish to override this, we must explicitly change is, just like above.

3.2.4 Renaming columns

While the above works, the column names now default to the moderately informative labels `X1`, `X2`, and so on. Fortunately, columns can be renamed using the `rename` function from the `tidyverse`. This function takes a tibble as its first argument, and a renaming instruction as its second, of the form `new_name = old_name`. For example, to rename the first column (which, as you may remember, refers to the continent of the corresponding mammal):

```
smithData <- read_tsv("Smith2003_data.txt", col_names = FALSE)
rename(smithData, Continent = X1) # Rename column "X1" to "Continent"
```

```
# A tibble: 5,731 x 9
   Continent X2      X3           X4      X5         X6          X7     X8 X9
   <chr>     <chr>   <chr>        <chr>   <chr>      <chr>    <dbl>  <dbl> <chr>
 1 AF        extant  Artiodactyla Bovidae Addax      nasomac~  4.85 7.00e4 60
 2 AF        extant  Artiodactyla Bovidae Aepyceros  melampus  4.72 5.25e4 63, ~
```

```
 3 AF        extant  Artiodactyla Bovidae Alcelaphus buselap~  5.23 1.71e5 63, ~
 4 AF        extant  Artiodactyla Bovidae Ammodorcas clarkei   4.45 2.80e4 60
 5 AF        extant  Artiodactyla Bovidae Ammotragus lervia    4.68 4.8 e4 75
 6 AF        extant  Artiodactyla Bovidae Antidorcas marsupi~  4.59 3.90e4 60
 7 AF        extinct Artiodactyla Bovidae Antidorcas bondi     4.53 3.4 e4 1
 8 AF        extinct Artiodactyla Bovidae Antidorcas austral~  4.6  4   e4 2
 9 AF        extant  Artiodactyla Bovidae Bos        taurus    5.95 9   e5 -999
10 AF        extant  Artiodactyla Bovidae Capra      walie     5    1   e5 -999
# ... with 5,721 more rows
```

As seen, the name of the first column now reads `Continent` instead of `X1`. One can similarly rename other columns as well.

3.2.5 Excel tables

Finally, although their use is discouraged in science, one should know how to read in data from an Excel spreadsheet. To do this, one needs to load the `readxl` package. This package is part of the `tidyverse`, but does not get automatically loaded when executing `library(tidyverse)`. Therefore, we first load the package:

```
library(readxl)
```

We can now load Excel files with the function `read_excel()`. At the start, we downloaded an Excel version of the data from Goldberg et al. (2010), called `Goldberg2010_data.xlsx`. It holds the exact same data as the original CSV file, just saved in Excel format for instructive purposes. Let us load this file:

```
read_excel("Goldberg2010_data.xlsx")
```

```
# A tibble: 356 x 2
   Species                   Status
   <chr>                      <dbl>
 1 Acnistus_arborescens           1
 2 Anisodus_tanguticus            1
 3 Atropa_belladonna              1
 4 Brachistus_stramonifolius      1
 5 Brugmansia_aurea               0
 6 Brugmansia_sanguinea           0
 7 Capsicum_annuum                1
 8 Capsicum_baccatum              1
```

```
 9 Capsicum_cardenasii            2
10 Capsicum_chacoense             1
# ... with 346 more rows
```

The functions `read_csv`, `read_tsv`, and `read_excel` have several further options. For example, given an Excel table with multiple sheets, one can specify which one to import, using the `sheet` argument. Check the help pages of these functions, and experiment with their options.

3.2.6 Writing data to files

Finally, data can not only be read from a file, but also written out to one. Then, instead of `read_csv`, `read_tsv` and the like, one uses `write_csv`, `write_tsv`, and so on. For instance, to save `dat` in CSV form:

```
write_csv(dat, "/path/to/file.csv")
```

where `/path/to/file.csv` should be replaced by the path and file name with which the data should be saved.

3.3 Exercises

1. Load the data from the file `Smith2003_data.txt`. Note that the file is tab-separated and lacks headers!
2. The columns of this file are, in order: Continent (AF=Africa, etc.), Status (extinct, historical, introduction, or extant), Order, Family, Genus, Species, Base-10 Log Mass, Combined Mass (grams), and Reference (numbers, referring to a numerically ordered list of published works – no need to worry about the details). Rename each column appropriately, using the `rename()` function.

4 Basic data manipulation

4.1 Selecting and manipulating data

Let us start by loading `tidyverse`, in case you have not done so yet:

```
library(tidyverse)
```

```
-- Attaching packages --------------------------------------- tidyverse 1.3.1 --

v ggplot2 3.3.5     v purrr   0.3.4
v tibble  3.1.6     v dplyr   1.0.8
v tidyr   1.2.0     v stringr 1.4.0
v readr   2.1.2     v forcats 0.5.1

-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

As you can see from the message output above, the `dplyr` package is part of `tidyverse`, which gets loaded by default. It allows one to arrange and manipulate data efficiently. The basic functions one should know are `rename`, `select`, `filter`, `slice`, `arrange`, and `mutate`. The first of these we have already looked at in Section 3.2.4. Let us then see some examples of the latter ones. First, we will load the `Goldberg2010_data.csv` data file (also discussed in the previous chapter):

```
dat <- read_csv("Goldberg2010_data.csv")
print(dat)
```

```
# A tibble: 356 x 2
   Species                  Status
   <chr>                     <dbl>
 1 Acnistus_arborescens          1
```

```
 2 Anisodus_tanguticus            1
 3 Atropa_belladonna              1
 4 Brachistus_stramonifolius      1
 5 Brugmansia_aurea               0
 6 Brugmansia_sanguinea           0
 7 Capsicum_annuum                1
 8 Capsicum_baccatum              1
 9 Capsicum_cardenasii            2
10 Capsicum_chacoense             1
# ... with 346 more rows
```

Now we will give examples of each of the functions `select`, `filter`, `slice`, `arrange`, and `mutate`. They are similar to our earlier `rename` in that the first argument they take is the data, in the form of a tibble. Their other arguments, and what they each do, are explained below.

4.1.1 `select`

This function chooses *columns* of the data. The second and subsequent arguments of the function are the columns which should be retained. For example, `select(dat, Species)` will keep only the `Species` column of `dat`:

```
select(dat, Species)
```

```
# A tibble: 356 x 1
   Species
   <chr>
 1 Acnistus_arborescens
 2 Anisodus_tanguticus
 3 Atropa_belladonna
 4 Brachistus_stramonifolius
 5 Brugmansia_aurea
 6 Brugmansia_sanguinea
 7 Capsicum_annuum
 8 Capsicum_baccatum
 9 Capsicum_cardenasii
10 Capsicum_chacoense
# ... with 346 more rows
```

It is also possible to deselect columns, by prepending a minus sign (`-`) in front of the column names. To drop the `Species` column, we can type:

```
  select(dat, -Species)
```

```
# A tibble: 356 x 1
   Status
    <dbl>
 1      1
 2      1
 3      1
 4      1
 5      0
 6      0
 7      1
 8      1
 9      2
10      1
# ... with 346 more rows
```

Since there were only two columns in the data to begin with, only the `Status` column remained in the data after removing `Species`.

4.1.2 `filter`

While `select` chooses columns, `filter` chooses rows from the data. As with all these functions, the first argument of `filter` is the data. The second argument is a logical condition on the columns. Those rows which satisfy the condition are retained; the rest are dropped. Thus, `filter` keeps only those rows of the data which fulfill some condition.

Recall that in the `Goldberg2010_data.csv` dataset, a `Status` of 0 means self-incompatibility; a `Status` of 1 means self-compatibility, and `Status` values between 2 and 5 refer to various, more complex selfing mechanisms. So in case we wanted to focus only on those species which exhibit complex selfing, we could `filter` the data like this:

```
  filter(dat, Status >= 2)
```

```
# A tibble: 44 x 2
   Species               Status
   <chr>                  <dbl>
 1 Capsicum_cardenasii        2
 2 Capsicum_pubescens         2
 3 Dunalia_solanacea          4
```

```
 4 Lycium_arenicola           4
 5 Lycium_californicum        3
 6 Lycium_exsertum            4
 7 Lycium_fremontii           4
 8 Lycium_gariepense          4
 9 Lycium_horridum            4
10 Lycium_strandveldense      4
# ... with 34 more rows
```

4.1.3 `slice`

With `slice`, one can choose rows of the data, just like with `filter`. Unlike with `filter` however, `slice` receives a vector of row indices instead of a condition to be tested on each row. So, for example, if one wanted to keep only the first, second, and fifth rows, then one can do so with `slice`:

```
slice(dat, c(1, 2, 5))
```

```
# A tibble: 3 x 2
  Species              Status
  <chr>                 <dbl>
1 Acnistus_arborescens      1
2 Anisodus_tanguticus       1
3 Brugmansia_aurea          0
```

(Note: the numbers in front of the rows in the output generated by tibbles always pertain to the row numbers of the *current table*, not the one from which they were created. So the row labels `1`, `2`, and `3` above simply enumerate the rows of the sliced data. The actual rows still correspond to rows 1, 2, and 5 in the original `dat`.)

4.1.4 `arrange`

This function rearranges the rows of the data, in increasing order of the column given as the second argument. For example, to arrange in increasing order of `Status`, we write:

```
arrange(dat, Status)
```

```
# A tibble: 356 x 2
   Species                 Status
   <chr>                    <dbl>
 1 Brugmansia_aurea             0
 2 Brugmansia_sanguinea         0
 3 Cuatresia_exiguiflora        0
 4 Cuatresia_riparia            0
 5 Dunalia_brachyacantha        0
 6 Dyssochroma_viridiflora      0
 7 Eriolarynx_lorentzii         0
 8 Grabowskia_duplicata         0
 9 Iochroma_australe            0
10 Iochroma_cyaneum             0
# ... with 346 more rows
```

To arrange in decreasing order, apply the `desc` function to `Status` within `arrange`, like this:

```
  arrange(dat, desc(Status))
```

```
# A tibble: 356 x 2
   Species               Status
   <chr>                  <dbl>
 1 Solanum_wendlandii         5
 2 Dunalia_solanacea          4
 3 Lycium_arenicola           4
 4 Lycium_exsertum            4
 5 Lycium_fremontii           4
 6 Lycium_gariepense          4
 7 Lycium_horridum            4
 8 Lycium_strandveldense      4
 9 Lycium_tetrandrum          4
10 Lycium_villosum            4
# ... with 346 more rows
```

It is also perfectly possible to arrange by a column whose type is character string. In that case, the system will automagically sort the rows in alphabetical order—or reverse alphabetical order, in case `desc` is applied. For example, to sort in reverse alphabetical order of species binomials:

```
  arrange(dat, desc(Species))
```

```
# A tibble: 356 x 2
   Species                   Status
   <chr>                      <dbl>
 1 Witheringia_solanacea          2
 2 Witheringia_mexicana           0
 3 Witheringia_meiantha           0
 4 Witheringia_macrantha          0
 5 Witheringia_cuneata            0
 6 Witheringia_coccoloboides      0
 7 Withania_somnifera             1
 8 Withania_coagulans             4
 9 Vassobia_breviflora            1
10 Symonanthus_bancroftii         4
# ... with 346 more rows
```

Notice that when we sorted the rows by `Status`, there are many ties—rows with the same value of `Status`. In those cases, `arrange` will not be able to decide which rows should come earlier, and so any ordering that was present before invoking `arrange` will be retained. In case we would like to break the ties, we can give further sorting variables, as the third, fourth, etc. arguments to `arrange`. To sort the data by `Status`, and to resolve ties in alphabetical order of `Species`, we write:

```
arrange(dat, Status, Species)
```

```
# A tibble: 356 x 2
   Species                  Status
   <chr>                     <dbl>
 1 Brugmansia_aurea              0
 2 Brugmansia_sanguinea          0
 3 Cuatresia_exiguiflora         0
 4 Cuatresia_riparia             0
 5 Dunalia_brachyacantha         0
 6 Dyssochroma_viridiflora       0
 7 Eriolarynx_lorentzii          0
 8 Grabowskia_duplicata          0
 9 Iochroma_australe             0
10 Iochroma_cyaneum              0
# ... with 346 more rows
```

This causes the table to be sorted primarily by `Status`, but in case there are ties (equal `Status` between multiple rows), they will be resolved in priority of alphabetical order—first those starting with "A" (if they exist), then "B", and so on.

4.1.5 `mutate`

The `mutate` function allows us to create new columns from existing ones. We may apply any function or operator we learned about to existing columns, and the result of the computation will go into the new column. We do this in the second argument of `mutate` (the first, as always, is the data), by first giving a name to the column, then writing `=`, and then the desired computation. For example, we may find it strange that the selfing status of the species is encoded with a number ranging from 0 to 5, instead of 1 to 6. This is easy to fix however, using `mutate`:

```
mutate(dat, NewStatus = Status + 1)
```

```
# A tibble: 356 x 3
   Species                   Status NewStatus
   <chr>                      <dbl>     <dbl>
 1 Acnistus_arborescens           1         2
 2 Anisodus_tanguticus            1         2
 3 Atropa_belladonna              1         2
 4 Brachistus_stramonifolius      1         2
 5 Brugmansia_aurea               0         1
 6 Brugmansia_sanguinea           0         1
 7 Capsicum_annuum                1         2
 8 Capsicum_baccatum              1         2
 9 Capsicum_cardenasii            2         3
10 Capsicum_chacoense             1         2
# ... with 346 more rows
```

The original columns of the data are retained, but we now also have the additional `NewStatus` column.

Perhaps more interestingly, we could create a new column indicating whether the selfing mechanism of the species is simple (`Status` either 0 or 1) or complex (`Status` between 2 and 5). We can do this using an `ifelse` function within `mutate`:

```
mutate(dat, SelfingMechanism = ifelse(Status < 2, "simple", "complex"))
```

```
# A tibble: 356 x 3
   Species                   Status SelfingMechanism
   <chr>                      <dbl> <chr>
 1 Acnistus_arborescens           1 simple
 2 Anisodus_tanguticus            1 simple
```

```
 3 Atropa_belladonna               1 simple
 4 Brachistus_stramonifolius       1 simple
 5 Brugmansia_aurea                0 simple
 6 Brugmansia_sanguinea            0 simple
 7 Capsicum_annuum                 1 simple
 8 Capsicum_baccatum               1 simple
 9 Capsicum_cardenasii             2 complex
10 Capsicum_chacoense              1 simple
# ... with 346 more rows
```

4.2 Using pipes to our advantage

When composing multiple `tidyverse` functions together, things can get unwieldy quite quickly. Let us take the same data, and create the new column `SelfingMechanism` as above. What happens if we then filter for only those entries with complex selfing mechanism, and finally, we select the column `Species` only? Here is the solution:

```
select(
  filter(
    mutate(dat, SelfingMechanism = ifelse(Status < 2, "simple", "complex")),
    SelfingMechanism == "complex"
  ),
  Species
)
```

```
# A tibble: 44 x 1
   Species
   <chr>
 1 Capsicum_cardenasii
 2 Capsicum_pubescens
 3 Dunalia_solanacea
 4 Lycium_arenicola
 5 Lycium_californicum
 6 Lycium_exsertum
 7 Lycium_fremontii
 8 Lycium_gariepense
 9 Lycium_horridum
10 Lycium_strandveldense
# ... with 34 more rows
```

The expression is highly unpleasant: to a human reader, it is not at all obvious what is happening above. To clarify, we have two options. One is to rely on repeated assignment:

```
mutatedDat <- mutate(dat, SelfingMechanism = ifelse(Status < 2, "simple", "complex"))
filteredDat <- filter(mutatedDat, SelfingMechanism == "complex")
onlySpeciesDat <- select(filteredDat, Species)
print(onlySpeciesDat)
```

```
# A tibble: 44 x 1
   Species
   <chr>
 1 Capsicum_cardenasii
 2 Capsicum_pubescens
 3 Dunalia_solanacea
 4 Lycium_arenicola
 5 Lycium_californicum
 6 Lycium_exsertum
 7 Lycium_fremontii
 8 Lycium_gariepense
 9 Lycium_horridum
10 Lycium_strandveldense
# ... with 34 more rows
```

This, however, requires inventing arbitrary variable names at every step, or else overwriting variables. For such a short example, this is not problematic, but doing the same for a long pipeline of dozens of steps could get confusing, as well as dangerous due to the repeatedly modified variables.

It turns out that one can get the best of both worlds: the safety of function composition with the conceptual clarity of repeated assignments. This only requires that we make use of the pipe operator `%>%` that we learned about earlier. As a reminder, for any function `f` and function argument `x`, `f(x, y, ...)` is the same as `x %>% f(y, ...)`, where the `...` denote potential further arguments to `f`. That is, the first argument of the function can be moved from the argument list to in front of the function, before the pipe symbol. The `tidyverse` functions take the data as their first argument, which means that the use of pipes allow us to very conveniently chain together multiple steps of data analysis. In our case, we can rewrite the original

```
select(
  filter(
    mutate(dat, SelfingMechanism = ifelse(Status < 2, "simple", "complex")),
    SelfingMechanism == "complex"
```

```
  ),
  Species
)
```

with the use of pipes, in a much more transparent way:

```
dat %>%
  mutate(SelfingMechanism = ifelse(Status < 2, "simple", "complex")) %>%
  filter(SelfingMechanism == "complex") %>%
  select(Species)
```

Again, the pipe **%>%** should be pronounced *then*. We take the data, *then* we mutate it, *then* we filter for complex selfing, and *then* we select one of the columns. In performing these steps, each function both receives and returns data. Thus, by starting out with the original `dat`, we no longer need to write out the data argument of the functions explicitly. Instead, the pipe takes care of that automatically for us, making the functions receive as their first input argument the piped-in data, and in turn producing transformed data as their output—which becomes the input for the next function in line.

In fact, there is technically no need to even assign `dat`. The pipe can just as well start with the `read_csv` call to import the dataset:

```
read_csv("Goldberg2010_data.csv") %>%
  mutate(SelfingMechanism = ifelse(Status < 2, "simple", "complex")) %>%
  filter(SelfingMechanism == "complex") %>%
  select(Species)
```

```
# A tibble: 44 x 1
   Species
   <chr>
 1 Capsicum_cardenasii
 2 Capsicum_pubescens
 3 Dunalia_solanacea
 4 Lycium_arenicola
 5 Lycium_californicum
 6 Lycium_exsertum
 7 Lycium_fremontii
 8 Lycium_gariepense
 9 Lycium_horridum
10 Lycium_strandveldense
# ... with 34 more rows
```

4.3 Exercises

1. The `Smith2003_data.txt` dataset we worked with last time occasionally has the entry `-999` in its last three columns. This stands for unavailable data. In R, there is a built-in way of referring to such information: by setting a variable to `NA`. (So, for example, `x <- NA` will set the variable `x` to `NA`.) Modify these columns (using `mutate`) so that the entries which are equal to `-999` are replaced with `NA`.
2. Remove all rows from the data which contain one or more `NA` values (hint: look up the function `drop_na`). How many rows are retained? And what was the original number of rows?

The `iris` dataset is a built-in table in R. It contains measurements of petal and sepal characteristics from three flower species belonging to the genus *Iris* (*I. setosa*, *I. versicolor*, and *I. virginica*). If you type `iris` in the console, you will see the dataset displayed. In solving the problems below, feel free to use the all-important `dplyr` cheat sheet.

3. The format of the data is not a `tibble`, but a `data.frame`. As mentioned in the previous chapter, the two are basically the same for practical purposes, though internally, `tibbles` do offer some advantages. Convert the `iris` data frame into a tibble. (Hint: look up the `as_tibble` function.)
4. Select the columns containing petal and sepal length, and species identity.
5. Get those rows of the data with petal length less than 4 cm, but sepal length greater than 4 cm.
6. Sort the data by increasing petal length but decreasing sepal length.
7. Create a new column called `MeanLength`. It should contain the average of the petal and sepal length (i.e., petal length plus sepal length, divided by 2) of each individual flower.
8. Perform the operations from the previous two exercises in a single long function call (using function composition).

5 Summary statistics and data normalization

Last time we learned the basics of reading writing, and manipulating data. To review, download `pop_data.csv` from Lisam and set your working directory to the folder where you saved it. This is a comma-separated file (CSV), so you can load it using `read_csv`. As usual, we first load the `tidyverse` package:

```
library(tidyverse)
```

And now we may use the `tidyverse` functionalities, such as `read_csv`:

```
pop <- read_csv("pop_data.csv")
```

The data we just loaded contain population densities of three species at two spatial patches (A and B) at various points in time, ranging from 1 to 50 in steps of 1:

```
pop
```

```
# A tibble: 100 x 5
    time patch species1 species2 species3
   <dbl> <chr>    <dbl>    <dbl>    <dbl>
 1     1 A         8.43     6.62    10.1
 2     1 B        10.1      3.28     6.27
 3     2 A         7.76     6.93    10.3
 4     2 B        10.1      3.04     6.07
 5     3 A         7.09     7.24    10.5
 6     3 B        10.1      2.8      5.82
 7     4 A         6.49     7.54    10.6
 8     4 B        10.1      2.56     5.57
 9     5 A         5.99     7.83    10.7
10     5 B        10.1      2.33     5.32
# ... with 90 more rows
```

One can now perform various manipulations on these data, by using the functions `rename`, `select`, `filter`, `arrange`, and `mutate` we have learned about in Chapter 4. For instance, we

could create a new column called `total` which contains the total community density (sum of the three species' population densities) at each point in time and each location:

```
  mutate(pop, total = species1 + species2 + species3)
```

```
# A tibble: 100 x 6
    time patch species1 species2 species3 total
   <dbl> <chr>    <dbl>    <dbl>    <dbl> <dbl>
 1     1 A         8.43     6.62    10.1   25.1
 2     1 B        10.1      3.28     6.27  19.7
 3     2 A         7.76     6.93    10.3   25
 4     2 B        10.1      3.04     6.07  19.2
 5     3 A         7.09     7.24    10.5   24.8
 6     3 B        10.1      2.8      5.82  18.7
 7     4 A         6.49     7.54    10.6   24.7
 8     4 B        10.1      2.56     5.57  18.2
 9     5 A         5.99     7.83    10.7   24.5
10     5 B        10.1      2.33     5.32  17.8
# ... with 90 more rows
```

As a reminder, this can be written equivalently, using pipes:

```
  pop %>% mutate(total = species1 + species2 + species3)
```

```
# A tibble: 100 x 6
    time patch species1 species2 species3 total
   <dbl> <chr>    <dbl>    <dbl>    <dbl> <dbl>
 1     1 A         8.43     6.62    10.1   25.1
 2     1 B        10.1      3.28     6.27  19.7
 3     2 A         7.76     6.93    10.3   25
 4     2 B        10.1      3.04     6.07  19.2
 5     3 A         7.09     7.24    10.5   24.8
 6     3 B        10.1      2.8      5.82  18.7
 7     4 A         6.49     7.54    10.6   24.7
 8     4 B        10.1      2.56     5.57  18.2
 9     5 A         5.99     7.83    10.7   24.5
10     5 B        10.1      2.33     5.32  17.8
# ... with 90 more rows
```

5.1 Creating summary data

One can create summaries of data using the `summarise` function. This will simply apply some function to a column. For example, to calculate the average population density of species 1 in `pop`, across both time and patches, one can write

```
pop %>% summarise(meanDensity1 = mean(species1))
```

```
# A tibble: 1 x 1
  meanDensity1
         <dbl>
1         5.30
```

Here `meanDensity1` is the name of the new column to be created, and the `mean` function is our summary function, collapsing the data into a single number.

So far, this is not particularly interesting; in fact, the exact same effect would have been achieved by typing the shorter `mean(pop$species1)` instead. The real power of `summarise` comes through when combined with `group_by`. This groups the data based on the given grouping variables. Let us see how this works in practice:

```
pop %>% group_by(patch)
```

```
# A tibble: 100 x 5
# Groups:   patch [2]
    time patch species1 species2 species3
   <dbl> <chr>    <dbl>    <dbl>    <dbl>
 1     1 A         8.43     6.62    10.1
 2     1 B        10.1      3.28     6.27
 3     2 A         7.76     6.93    10.3
 4     2 B        10.1      3.04     6.07
 5     3 A         7.09     7.24    10.5
 6     3 B        10.1      2.8      5.82
 7     4 A         6.49     7.54    10.6
 8     4 B        10.1      2.56     5.57
 9     5 A         5.99     7.83    10.7
10     5 B        10.1      2.33     5.32
# ... with 90 more rows
```

Seemingly nothing has happened; the only difference is the extra line of comment above, before the printed table, saying `Groups: patch [2]`. What this means is that the rows of the data were internally split into two groups. The first have `"A"` as their patch, and the second have `"B"`. Whenever one groups data using `group_by`, rows which share the same unique combination of the grouping variables now belong together, and *subsequent* operations will act separately on each group instead of acting on the table as a whole (which is what we have been doing so far). That is, `group_by` does not actually alter the data; it only alters the behaviour of the functions applied to the grouped data.

If we group not just by `patch` but also by `time`, the comment above the table will read `Groups: patch, time [100]`:

```
pop %>% group_by(patch, time)
```

```
# A tibble: 100 x 5
# Groups:   patch, time [100]
    time patch species1 species2 species3
   <dbl> <chr>    <dbl>    <dbl>    <dbl>
 1     1 A         8.43     6.62    10.1
 2     1 B        10.1      3.28     6.27
 3     2 A         7.76     6.93    10.3
 4     2 B        10.1      3.04     6.07
 5     3 A         7.09     7.24    10.5
 6     3 B        10.1      2.8      5.82
 7     4 A         6.49     7.54    10.6
 8     4 B        10.1      2.56     5.57
 9     5 A         5.99     7.83    10.7
10     5 B        10.1      2.33     5.32
# ... with 90 more rows
```

This is because there are 100 unique combinations of patch and time: two different `patch` values (`"A"` and `"B"`), and fifty points in time (1, 2, ..., 50). So we have "patch A, time 1" as group 1, "patch B, time 1" as group 2, "patch A, time 3" as group 3, and so on until "patch B, time 50" as our group 100.

As mentioned, functions that are applied to grouped data will act on the groups separately. To return to the example of calculating the mean population density of species 1 in the two patches, we can write:

```
pop %>%
  group_by(patch) %>%
  summarise(meanDensity1 = mean(species1))
```

```
# A tibble: 2 x 2
  patch meanDensity1
  <chr>        <dbl>
1 A             5.29
2 B             5.32
```

One may obtain multiple summary statistics within the same `summarize` function. Below we compute both the mean and the standard deviation of the densities per patch:

```
pop %>%
  group_by(patch) %>%
  summarise(meanDensity1 = mean(species1), sdDensity1 = sd(species1))
```

```
# A tibble: 2 x 3
  patch meanDensity1 sdDensity1
  <chr>        <dbl>      <dbl>
1 A             5.29      0.833
2 B             5.32      3.81
```

Let us see what happens if we calculate the mean density of species 1—but grouping by `time` instead of `patch`:

```
pop %>%
  group_by(time) %>%
  summarise(meanDensity1 = mean(species1))
```

```
# A tibble: 50 x 2
    time meanDensity1
   <dbl>        <dbl>
 1     1         9.28
 2     2         8.94
 3     3         8.60
 4     4         8.3
 5     5         8.04
 6     6         7.84
 7     7         7.68
 8     8         7.54
 9     9         7.44
10    10         7.35
# ... with 40 more rows
```

The resulting table has 50 rows—half the number of rows in the original data, but many more than the two rows we get after grouping by `patch`. The reason is that there are 50 unique time points, and so the average is now computed over those rows which share `time`. But there are only two rows per moment of time: the rows corresponding to patch A and patch B. When we call `summarise` after having grouped by `time`, the averages are computed over the densities in these two rows only, per group. That is why here we end up with a table which has a single row per point in time.

> ⚠ Warning
>
> An easy mistake to make when one first meets with grouping and summaries is to assume that if we call `group_by(patch)`, then the subsequent summaries will be taken over patches. *This is not the case*, and be sure to take a moment to understand why. When we apply `group_by(patch)`, we are telling R to treat different patch values as group indicators. Therefore, when creating a summary, only the patch identities are retained from the original data (apart from the new summary statistics we calculate, of course). This means that the subsequent summaries are taken over everything *except* the patches. This should be clear after comparing the outputs of
>
> ```
> pop %>% group_by(patch) %>% summarise(meanDensity1 = mean(species1))
> ```
>
> and
>
> ```
> pop %>% group_by(time) %>% summarise(meanDensity1 = mean(species1))
> ```
>
> The first distinguishes the rows of the data only by `patch`, and therefore the average is taken over time. The second distinguishes the rows by `time`, so the average is taken over the patches. Run the two expressions again to see the difference between them!

We can use functions such as `mutate` or `filter` on grouped data. For example, we might want to know the deviation of species 1's density from its average *in each patch*. Doing the following does not quite do what we want:

```
pop %>% mutate(species1Dev = species1 - mean(species1))
```

```
# A tibble: 100 x 6
    time patch species1 species2 species3 species1Dev
   <dbl> <chr>    <dbl>    <dbl>    <dbl>       <dbl>
 1     1 A         8.43     6.62    10.1         3.13
 2     1 B        10.1      3.28     6.27        4.83
 3     2 A         7.76     6.93    10.3         2.46
 4     2 B        10.1      3.04     6.07        4.82
```

```
 5     3 A         7.09     7.24    10.5        1.79
 6     3 B        10.1      2.8      5.82       4.82
 7     4 A         6.49     7.54    10.6        1.19
 8     4 B        10.1      2.56     5.57       4.81
 9     5 A         5.99     7.83    10.7        0.685
10     5 B        10.1      2.33     5.32       4.80
# ... with 90 more rows
```

This will put the difference of species 1's density from its mean density across both time and patches into the new column `species1Dev`. Which is not the same as calculating the difference from the mean in a given patch—patch A for rows corresponding to patch A, and patch B for the others. To achieve this, all one needs to do is to group the data by `patch` before invoking `mutate`:

```
pop %>%
  group_by(patch) %>%
  mutate(species1Dev = species1 - mean(species1))
```

```
# A tibble: 100 x 6
# Groups:   patch [2]
    time patch species1 species2 species3 species1Dev
   <dbl> <chr>    <dbl>    <dbl>    <dbl>       <dbl>
 1     1 A         8.43     6.62    10.1        3.14
 2     1 B        10.1      3.28     6.27       4.81
 3     2 A         7.76     6.93    10.3        2.47
 4     2 B        10.1      3.04     6.07       4.80
 5     3 A         7.09     7.24    10.5        1.80
 6     3 B        10.1      2.8      5.82       4.80
 7     4 A         6.49     7.54    10.6        1.20
 8     4 B        10.1      2.56     5.57       4.79
 9     5 A         5.99     7.83    10.7        0.702
10     5 B        10.1      2.33     5.32       4.78
# ... with 90 more rows
```

Comparing this with the previous table, we see that the values in the `species1Dev` column are now different, because this time the differences are taken with respect to the average densities per each patch.

Finally, since `group_by` changes subsequent behaviour, we might eventually want to get rid of the grouping in our data. To do so, one must use `ungroup`. For example:

```r
pop %>%
  group_by(patch) %>%
  summarise(meanDensity1 = mean(species1), sdDensity1 = sd(species1)) %>%
  ungroup()
```

```
# A tibble: 2 x 3
  patch meanDensity1 sdDensity1
  <chr>        <dbl>      <dbl>
1 A             5.29      0.833
2 B             5.32      3.81
```

It is good practice to always `ungroup` the data after we have calculated what we wanted using the group structure.

5.2 Data normalization

In science, we often strive to work with so-called *normalized data*. A dataset is normalized if:

1. Each variable is in its own column;
2. Each observation is in its own row.

Normalized data are suitable for performing operations, statistics, and plotting on. Furthermore, normalized data have a certain tidy feel to them, in the sense that their organization always follows the same general pattern regardless of the type of dataset one studies. (By contrast, every non-normalized dataset tends to be messy in its own unique way.) The `tidyverse` offers a simple and convenient way to normalize data.

For example, the `pop` table from the previous section is not normalized. This is because although each variable is in its own column, it is not true that each observation is in its own row. In fact, each row contains three observations: the densities of species 1, 2, and 3 at a given time and place. To normalize these data, we create *key-value pairs*. We merge the columns for species densities into just two new ones. The first of these (the *key*) indicates whether it is species 1, or 2, or 3 which the given row refers to. The second column (the *value*) contains the population density of the given species. Such key-value pairs are created by the function `pivot_longer`:

```r
pop %>% pivot_longer(cols = 3:5, names_to = "species", values_to = "density")
```

```
# A tibble: 300 x 4
    time patch species  density
   <dbl> <chr> <chr>      <dbl>
 1     1 A     species1    8.43
 2     1 A     species2    6.62
 3     1 A     species3   10.1
 4     1 B     species1   10.1
 5     1 B     species2    3.28
 6     1 B     species3    6.27
 7     2 A     species1    7.76
 8     2 A     species2    6.93
 9     2 A     species3   10.3
10     2 B     species1   10.1
# ... with 290 more rows
```

The function `pivot_longer` takes three arguments (apart, of course, from the first data argument that we may also pipe in, like above). First, `cols` is the list of columns to be converted into key-value pairs. One can refer to the columns by number: `3:5` is the same as `c(3, 4, 5)` and selects the third, fourth, and fifth columns—the ones corresponding to the population densities. We could also have written `c("species1", "species2", "species3")` instead, choosing columns by their names. This can give greater clarity, albeit at the cost of more typing. Second, the argument `names_to` is the name of the new key column. Finally, `values_to` is the name of the new value column.

Notice that the above table is now normalized: each column records a single variable, and each row contains a single observation. Notice also that, unlike the original `pop` which had 100 rows and 5 columns, the normalized version has 300 rows and 4 columns. This is natural: since the number of rows was reduced, there must be some extra rows to prevent the loss of information.

It is possible to "undo" the effect `pivot_longer`. To do so, use `pivot_wider`:

```
pop %>%
  pivot_longer(cols = 3:5, names_to = "species", values_to = "density") %>%
  pivot_wider(names_from = "species", values_from = "density")
```

```
# A tibble: 100 x 5
    time patch species1 species2 species3
   <dbl> <chr>    <dbl>    <dbl>    <dbl>
 1     1 A         8.43     6.62    10.1
 2     1 B        10.1      3.28     6.27
 3     2 A         7.76     6.93    10.3
```

```
 4     2 B        10.1      3.04     6.07
 5     3 A         7.09     7.24    10.5
 6     3 B        10.1      2.8      5.82
 7     4 A         6.49     7.54    10.6
 8     4 B        10.1      2.56     5.57
 9     5 A         5.99     7.83    10.7
10     5 B        10.1      2.33     5.32
# ... with 90 more rows
```

The two named arguments of `pivot_wider` above are `names_from` (which specifies the column from which the names for the new columns will be taken), and `values_from` (the column whose values will be used to fill in the rows under those new columns).

As a remark, one could make the data even "wider", by not only making columns out of the population densities, but the densities at a given patch. Doing so is very simple: one just needs to specify both the `species` and `patch` columns from which the new column names will be compiled:

```
pop %>%
  pivot_longer(cols = 3:5, names_to = "species", values_to = "density") %>%
  pivot_wider(names_from = c("species", "patch"), values_from = "density")
```

```
# A tibble: 50 x 7
    time species1_A species2_A species3_A species1_B species2_B species3_B
   <dbl>      <dbl>      <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
 1     1       8.43       6.62      10.1        10.1       3.28       6.27
 2     2       7.76       6.93      10.3        10.1       3.04       6.07
 3     3       7.09       7.24      10.5        10.1       2.8        5.82
 4     4       6.49       7.54      10.6        10.1       2.56       5.57
 5     5       5.99       7.83      10.7        10.1       2.33       5.32
 6     6       5.58       8.1       10.7        10.1       2.12       5.08
 7     7       5.27       8.34      10.6        10.1       1.92       4.86
 8     8       5.02       8.54      10.4        10.1       1.74       4.64
 9     9       4.82       8.7       10.0        10.0       1.58       4.43
10    10       4.66       8.82       9.66       10.0       1.43       4.23
# ... with 40 more rows
```

If normalized data are what we strive for, what is the practical use of `pivot_wider`? There are two answers to this question. First, while non-normalized data are indeed less efficient from a computational and data analysis standpoint, they are often more human-readable. For example, the `pop` table is easy to read despite the lack of normalization, because each row

corresponds to a given time and place. By normalizing the data, information referring to any given time and place will be spread out over multiple (in our case, three) rows—one for each species. While this is preferable from a data analysis point of view, it can be more difficult to digest visually. Second, wide data lend themselves very well to one particular class of statistical techniques called *multivariate analysis*. In case one wants to perform multivariate analysis, wide-format data are often better than normalized data.

Finally, it is worth noting the power of normalized data in, e.g., generating summary statistics. To obtain the mean and the standard deviation of the population densities for each species in each patch, all one has to do is this:

```
pop %>%
  pivot_longer(3:5, names_to = "species", values_to = "density") %>% # Normalize data
  group_by(patch, species) %>% # Group data by both species and patch
  summarise(meanDensity = mean(density), sdDensity = sd(density)) %>% # Obtain statistics
  ungroup() # Don't forget to ungroup the data at the end
```

```
# A tibble: 6 x 4
  patch species  meanDensity sdDensity
  <chr> <chr>          <dbl>     <dbl>
1 A     species1        5.29     0.833
2 A     species2        8.05     0.559
3 A     species3        7.51     1.56
4 B     species1        5.32     3.81
5 B     species2        1.07     0.737
6 B     species3        6.57     2.48
```

5.3 Exercises

The exercises below use the `iris` dataset—the same that we used for last chapter's data wrangling exercises. Convert the `iris` data to a tibble with the `as_tibble()` function, and assign it to a variable.

1. Create a new column in the `iris` dataset which contains the deviation of petal lengths from the average of the whole dataset.
2. Create a new column in the `iris` dataset which contains the deviation of petal lengths from the average *of each species*. (Hint: `group_by` the species and then `mutate`!)
3. Create a table where the rows are the three species, and the columns are: average petal length, variance of petal length, average sepal length, and variance of sepal length.

4. Create key-value pairs in the `iris` dataset for the petal characteristics. In other words, have a column called `Petal.Trait` (whose values are either `Petal.Length` or `Petal.Width`), and another column called `Petal.Value` (with the length/width values).

5. Repeat the same exercise, but now for sepal traits.

6. Finally, do it for both petal and sepal traits simultaneously, to obtain a fully normalized form of the `iris` data. That is, the key column (call it `Flower.Trait`) will have the values `Petal.Length`, `Petal.Width`, `Sepal.Length`, and `Sepal.Width`. And the value column (which you can call `Trait.Value`) will have the corresponding measurements.

6 Creating publication-grade figures

In science, we want good, clear plots. Each figure should make it obvious what data you are plotting, what the axes, colors, shapes, and size differences represent, and the overall message the figure is conveying. When writing a scientific paper, or a report for an environmental consulting company, etc., remember that your future readers are busy people. They often do not have the time to delve into the subtleties of overly refined verbal arguments. Instead, they will most often glance at the figures to learn what your work is about. You will want to create figures which makes this possible for them to do.

Here we will learn how to create accessible, publication-quality scientific graphs in a simple way. We do this using the R package `ggplot2` which is a standard part of the `tidyverse`. The `ggplot2` package follows a very special philosophy for creating figures. In 2006, Leland Wilkinson proposed a *grammar of graphics*.[1] Just like sentences, graphs have fixed parts whose specification defines the plot. The grand idea is that the data ought not be changed in order to display it in different formats. For instance, the same data should be possible to represent either as a box plot, or as a histogram, without changing the data format.

This last claim needs to be qualified somewhat. It is more accurate to say that one should not need to change the data format *as long as the data are normalized*. As a reminder, "normalized data" means that every variable is in its own column, and every observation in its own row. In case the data are not normalized, one should first wrangle them into such form, for example by using `pivot_wider()`. For certain kinds of graphs though, this step is not required. But when working with larger datasets, it can be very useful to normalize the data before analyzing and plotting them—we will see examples of this in the exercises.

To see how `ggplot2` works, let us load `tidyverse`, and then use the built-in `iris` dataset to create some figures. As a reminder, here is what the data look like:

```
library(tidyverse)
as_tibble(iris)
```

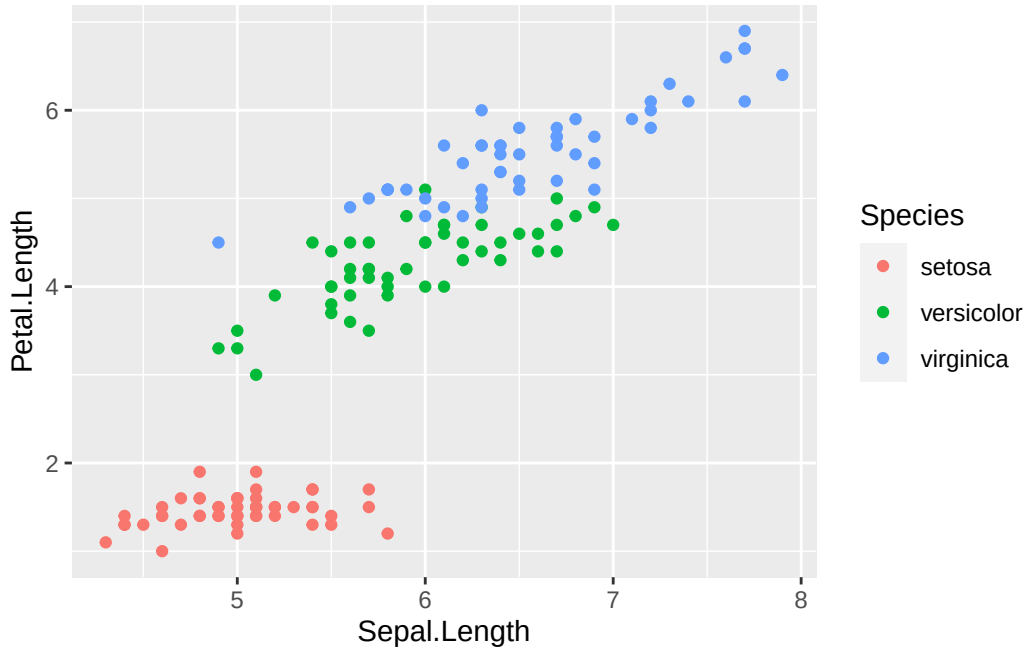
```
# A tibble: 150 x 5
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          <dbl>       <dbl>        <dbl>       <dbl> <fct>
 1          5.1         3.5          1.4         0.2 setosa
```

[1] Wilkinson, L. (2006), *The Grammar of Graphics*. Springer Science & Business Media.

```
 2          4.9         3           1.4         0.2 setosa
 3          4.7         3.2         1.3         0.2 setosa
 4          4.6         3.1         1.5         0.2 setosa
 5          5           3.6         1.4         0.2 setosa
 6          5.4         3.9         1.7         0.4 setosa
 7          4.6         3.4         1.4         0.3 setosa
 8          5           3.4         1.5         0.2 setosa
 9          4.4         2.9         1.4         0.2 setosa
10          4.9         3.1         1.5         0.1 setosa
# ... with 140 more rows
```

Let us, as a first step, create a plot where sepal length (x-axis) is plotted against petal length (y-axis), with the points referring to different species shown in different colors:

```
ggplot(iris) +
  aes(x = Sepal.Length, y = Petal.Length, colour = Species) +
  geom_point()
```



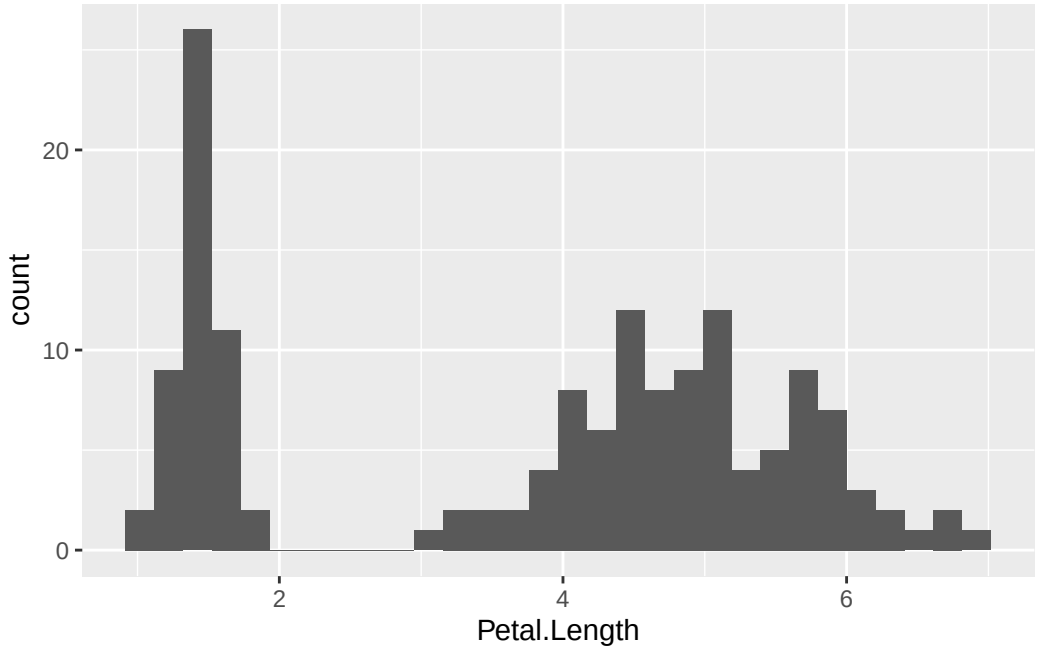
Here we defined the **data** (`iris`; feel free to use pipes, as in `iris %>% ggplot() + ...`), then the **aesthetic mappings** with `aes()`, and finally the **geometry** of how to display the data with `geom_point()`. The important thing to remember is that the aesthetic mappings are all those aspects of the figure that are governed by your data. For instance, if you wanted

to set the color of all points to blue, this would not be an aesthetic mapping, because it applies regardless of what the data are (in case you want to do this, you would have to specify `geom_point(colour = "blue")` in the last line). The geometry of your plot, on the other hand, governs the overall visual arrangement of your data (points, lines, histograms, etc). There are many different `geom_`s; we will learn about some here, but when in doubt, Google and a ggplot2 cheat sheet are your best friends.

Notice that the different "grammatical" components of the plot (aesthetics, geometry) are *added* to the plot, using the `+` symbol for addition. This is due to somewhat of an unfortunate historical accident. As it happens, `ggplot2` is older than the rest of the `tidyverse`, and so when it was first designed, the pipe operator `%>%` did not yet exist. This means that within the creating of a ggplot graph, one must use `+` to compose the various graph elements; but outside that, the usual `%>%` is used for function composition.

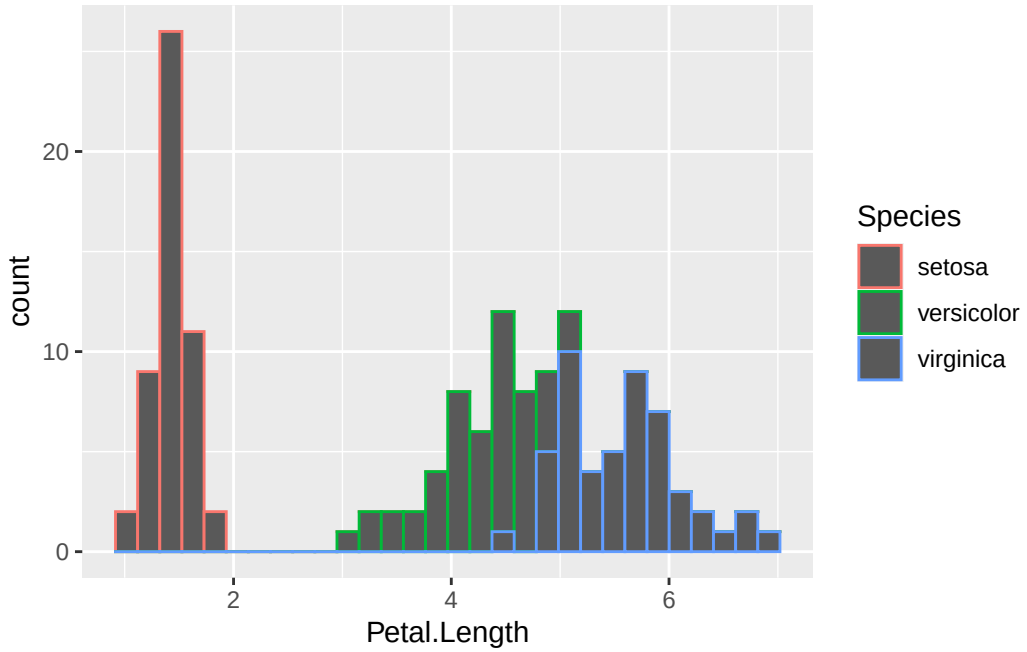
To look at a different kind of geometry, let us create a histogram of the petal lengths. This is done using `geom_histogram`:

```
iris %>%
  ggplot() +
  aes(x = Petal.Length) +
  geom_histogram()
```



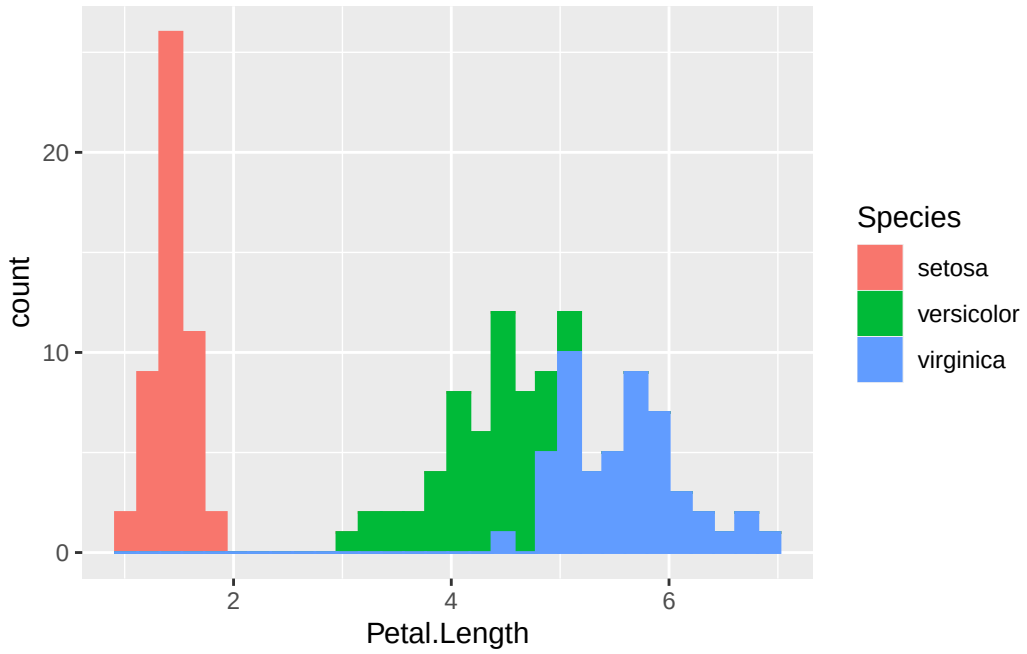
We see two clusters of data. Why is that? One might suspect that this is because of a species-level difference. To check if that is the case, let us color the histogram by species:

```
iris %>%
  ggplot() +
  aes(x = Petal.Length, colour = Species) +
  geom_histogram()
```



(A color legend was created automatically on the right.) This changes the color of the outline of the histograms, but not the color of their fill. To do so, we need to change the `fill` color as well, which is a separate aesthetic property:

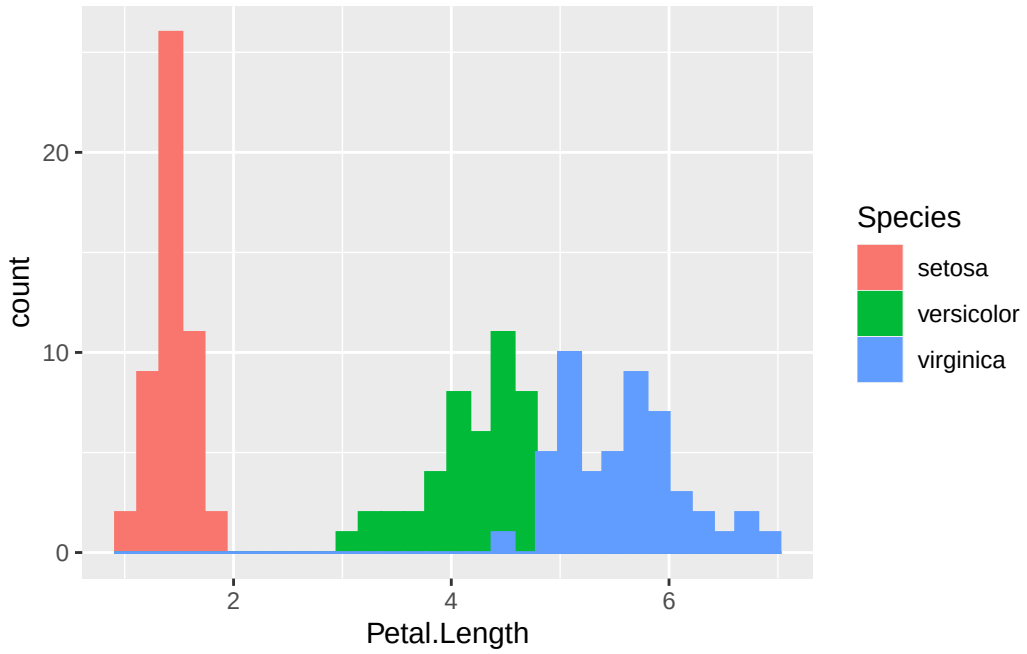
```
iris %>%
  ggplot() +
  aes(x = Petal.Length, colour = Species, fill = Species) +
  geom_histogram()
```



This is fine, but now the histogram bars of different species are stacked on top of one another. This means that, for example at around a petal length of 5 cm where individuals of both *Iris versicolor* and *I. virginica* are found, the green bar on top of the blue one does not begin from the bottom of the y-axis at 0, but from wherever the blue bar ends and the green one begins (in our case, from around 10 upwards). So the number of *I. versicolor* individuals in the 5 cm bin is not 12, but only 2.

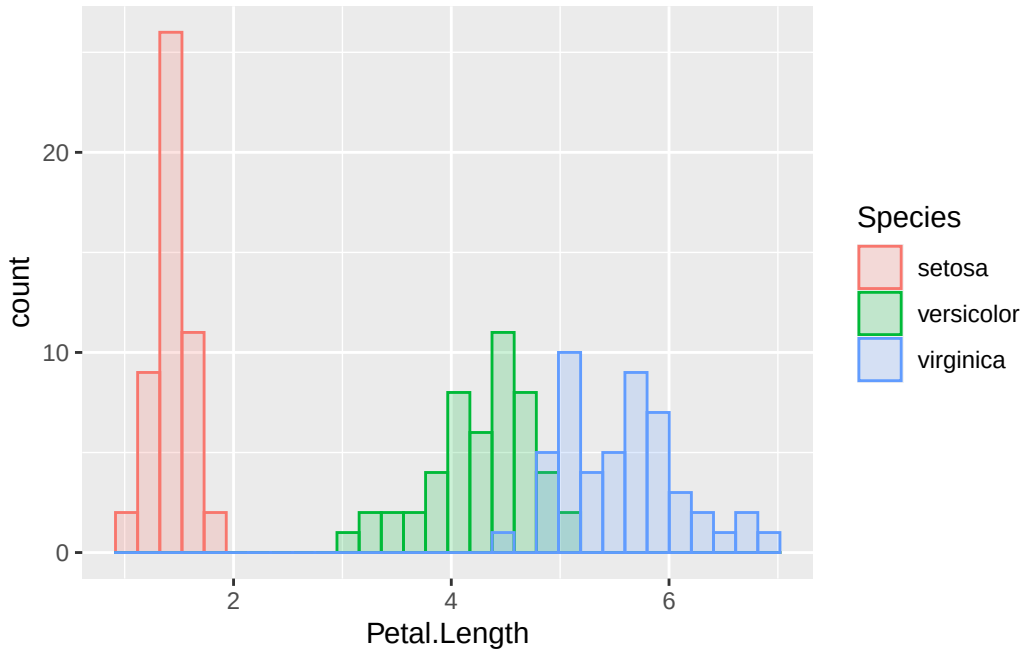
It may be easier to interpret the histogram if all bars start at the bottom, even if this means that the bars will now overlap to an extent. To achieve this overlap, one simply has to add the argument `position = "identity"` to `geom_histogram`:

```
iris %>%
  ggplot() +
  aes(x = Petal.Length, colour = Species, fill = Species) +
  geom_histogram(position = "identity")
```



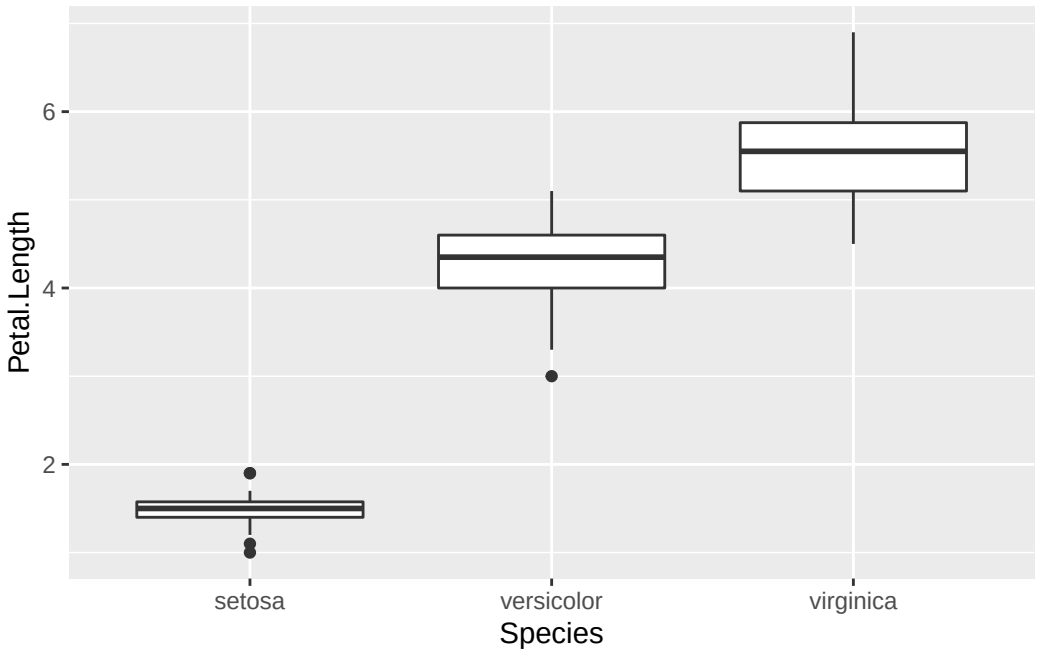
(There are other options as well, such as `position = "dodge"` – feel free to experiment. In general, you can learn what options each of the `geom_`s have by invoking the Help, or asking Google.) The figure above is almost perfect; the one remaining problem is that, due to the lack of transparency, the bars belonging to different species cover each other completely. So let us change the fill's *transparency*, which is called `alpha` in ggplot. The `alpha` property can take on any value between 0 (fully transparent object, to the point of invisibility) to 1 (fully solid object with no transparency, like above; this is the default). Setting it to 0.2 (say) will finally fully reveal the distribution of each species:

```
iris %>%
  ggplot() +
  aes(x = Petal.Length, colour = Species, fill = Species) +
  geom_histogram(position = "identity", alpha = 0.2)
```



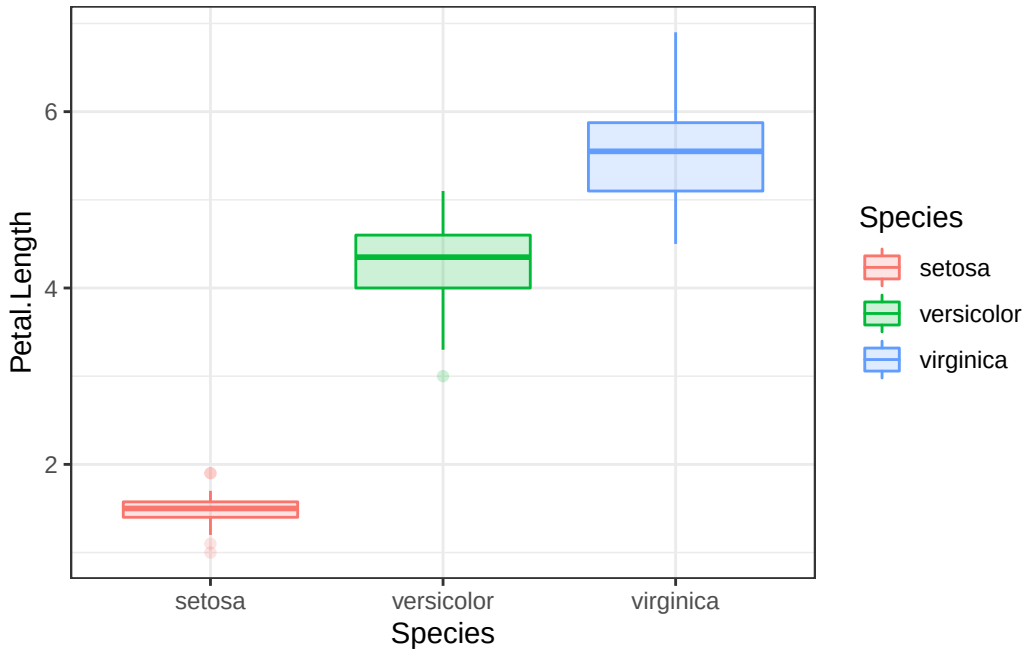
There are plenty of other `geom_`s as well, such as `geom_boxplot`, `geom_violin`, etc. Let us try `geom_boxplot` for instance. Let us create one box plot of the distribution of petal lengths for each species. Putting the species along the x-axis and petal length along the y-axis:

```
iris %>%
  ggplot() +
  aes(x = Species, y = Petal.Length) +
  geom_boxplot()
```



Although this is sufficient, one should feel free to make the plots prettier. For instance, one could use colors, like before. One can also use different themes. There are pre-defined themes such as `theme_classic()`, `theme_bw()`, and so on. For example, using `theme_bw()` gets rid of the default gray background and replaces it with a white one. The version of the graph below applies this theme, and also changes the color and fill to be based on species identity:

```
iris %>%
  ggplot() +
  aes(x = Species, y = Petal.Length, colour = Species, fill = Species) +
  geom_boxplot(alpha = 0.2) +
  theme_bw()
```

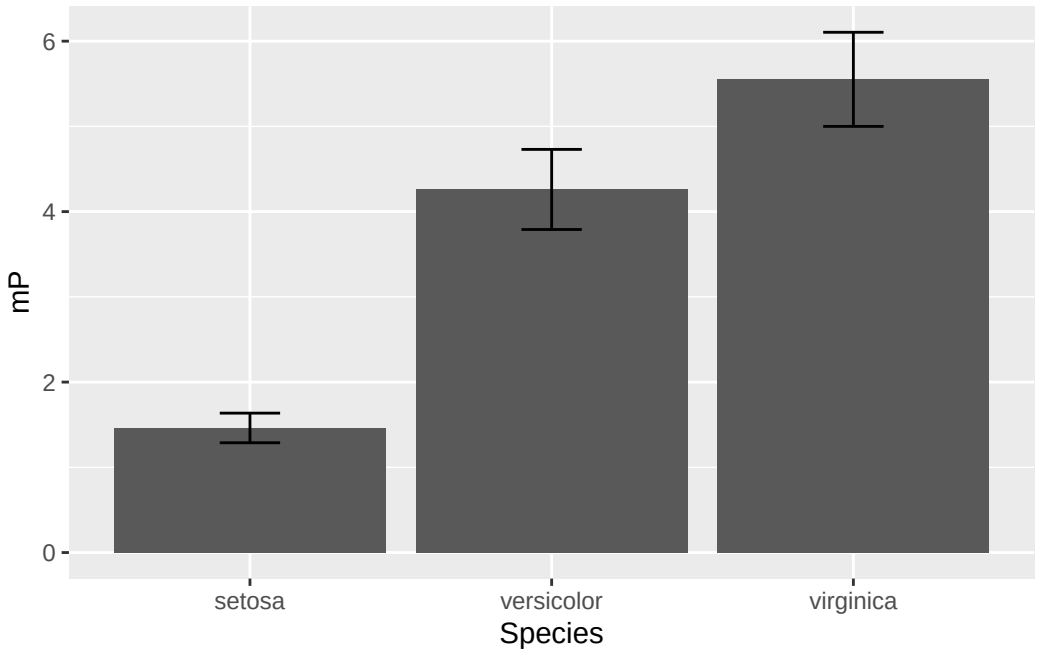


6.1 Summaries and confidence intervals

Providing appropriate information on experimental errors is a hallmark of any credible scientific graph. Choose a type of error based on the conclusion that you want the reader to draw. While the standard deviation (SD) represents the dispersion of the data, the standard error of the mean (SEM) and confidence intervals (CI) report the certainty of the estimate of a value (e.g., certainty in estimating the mean). Let us see examples of each.

Let us first obtain the mean and the standard deviation of the petal lengths for each species. We then would like to plot them. How to proceed? By creating a workflow which first uses `group_by` and `summarise` to obtain the required statistics (means and standard deviations), and then uses these data for plotting. One can include all these steps in a logical workflow:

```
iris %>%
  group_by(Species) %>% # Perform summary calculations for each species separately
  summarise(mP = mean(Petal.Length), sP = sd(Petal.Length)) %>% # Mean & sd petal length
  ungroup() %>% # Ungroup data
  ggplot() + # Start plotting
  aes(x = Species, y = mP, ymin = mP - sP, ymax = mP + sP) + # Means plus/minus std devs
  geom_col() + # This takes the y aesthetic, for plotting the mean mP
  geom_errorbar(width = 0.2) # Takes the ymin and ymax aesthetics
```

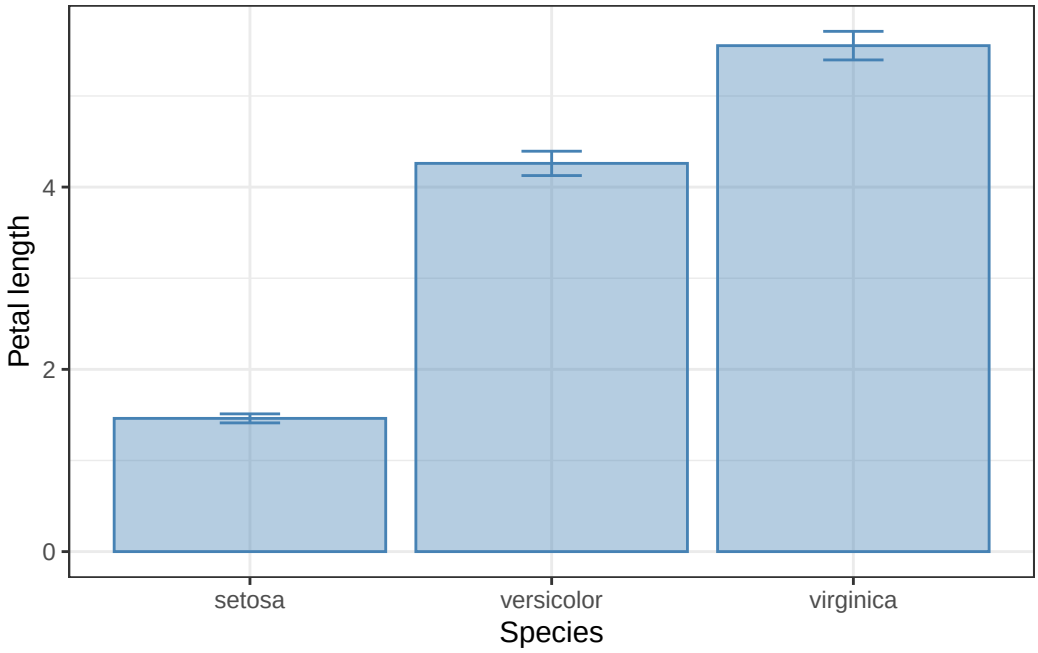


Note that the y-axis label is not very descriptive. This is because it inherited the name of the corresponding data column, `mP`. There are multiple ways to change it; the simplest is to add `ylab("Petal length")` to the plot. Another way of doing so is duscussed in Chapter 7.

In case we want to calculate the 95% confidence intervals of the mean values, we first obtain some necessary summary statistics: the number of observations (sample size) in each group; the standard error of the mean (standard deviation divided by the square root of the sample size); and finally, the confidence interval itself (read off from the t-distribution, with one fewer degrees of freedom than the sample size). We can then include these confidence intervals on top of the mean values:

```
iris %>%
  group_by(Species) %>%
  summarise(mP = mean(Petal.Length), # Mean petal length per species
            sP = sd(Petal.Length), # Standard deviation per species
            N = n(), # Sample size (number of observations) per species
            SEM = sP / sqrt(N), # Standard error of the mean
            CI = SEM * qt(0.975, N - 1)) %>% # Confidence interval, read off
                 # at 0.975 because plus/minus 2.5% adds up to 5%
  ggplot() +
  aes(x = Species, y = mP, ymin = mP - CI, ymax = mP + CI) +
  geom_col(alpha = 0.4, colour = "steelblue", fill = "steelblue") +
  geom_errorbar(width = 0.2, colour = "steelblue") +
```

```
    ylab("Petal length") +
    theme_bw()
```



6.2 Exercises

Fauchald et al. (2017)[2] tracked the population size of various herds of caribou in North America over time, and correlated population cycling with the amount of vegetation and sea-ice cover. The part of their data that we will use consists of two files (found on Lisam): `pop_size.tsv` (data on herd population sizes), and `sea_ice.tsv` (on sea levels of sea ice cover per year and month).

1. The file `sea_ice.tsv` is in human-readable, wide format. Note however that the rule "each set of observations is stored in its own row" is violated. For computation, we would like to organize the data in a tidy tibble with four columns: `Herd`, `Year`, `Month`, and `Cover`. To this end, apply the function `pivot_longer` to columns 3-14 in the tibble, gathering the names of the months in the new column `Month` and the values in the new column `Cover`.

[2]Fauchald, P. et al. (2017). Arctic greening from warming promotes declines in caribou populations. *Science Advances*, 3:e1601365.

2. Use `pop_size.tsv` to make a plot of herd sizes through time. Let the x-axis be Year, the y-axis be population size. Show different herds in different colors. For geometry, use points.

3. The previous plot is actually not that easy to see and interpret. To make it better, add a `line` geometry as well, which will connect the points with lines.

4. Make a histogram out of all population sizes in the data.

5. Make the same histogram, but break it down by herd, using a different color and fill for each herd.

6. Instead of a histogram, make a density plot with the same data and display (look up `geom_density` if needed).

7. Make box plots of the population size of each herd. Along the x-axis, each herd should be separately displayed; the y-axis should be population size. The box plots should summarize population sizes across all years.

8. Let us go back to `sea_ice.tsv`. Make the following plot. Along the x-axis, have Year. Along the y-axis, Month. Then, for each month-year pair, color the given part of the plot darker for lower ice cover and lighter for more. (Hint: look up `geom_tile` if needed.) Finally, make sure to do all this only for the herd with the i.d. `WAH` (filter the data before plotting).

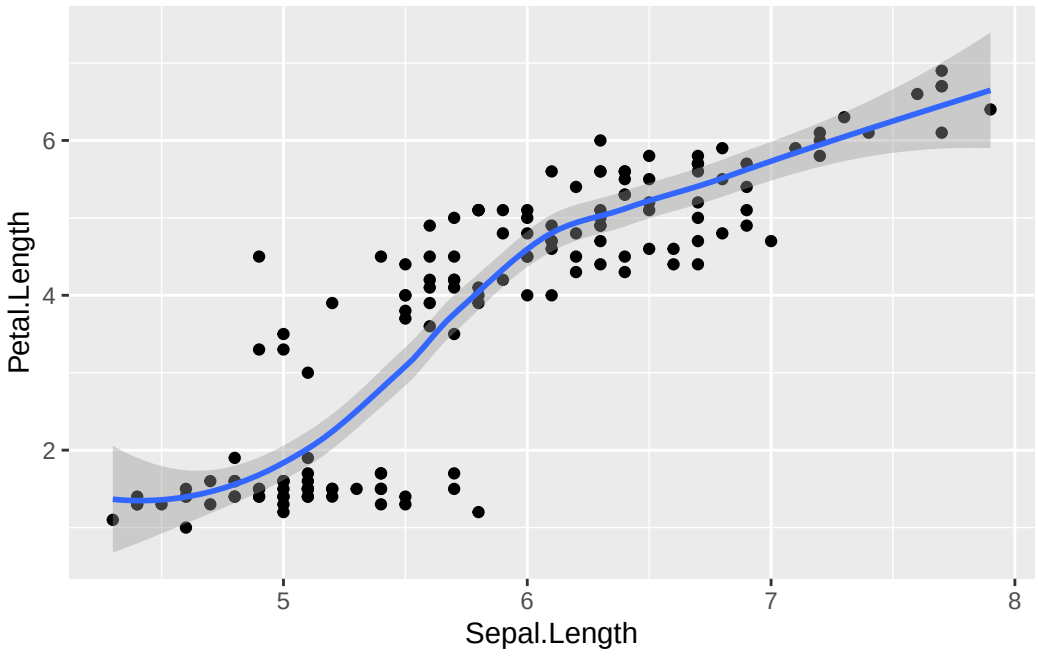
7 Some further plotting options

7.1 Smoothing and regression lines

Last time we learned about aesthetic mappings and various `geom_` options, such as `geom_point`, `geom_histogram`, and `geom_boxplot`. Let us explore another type of `geom_`, which approximates the trend of a set of data points with a line and an error bar that shows the confidence interval of the estimate at each point:

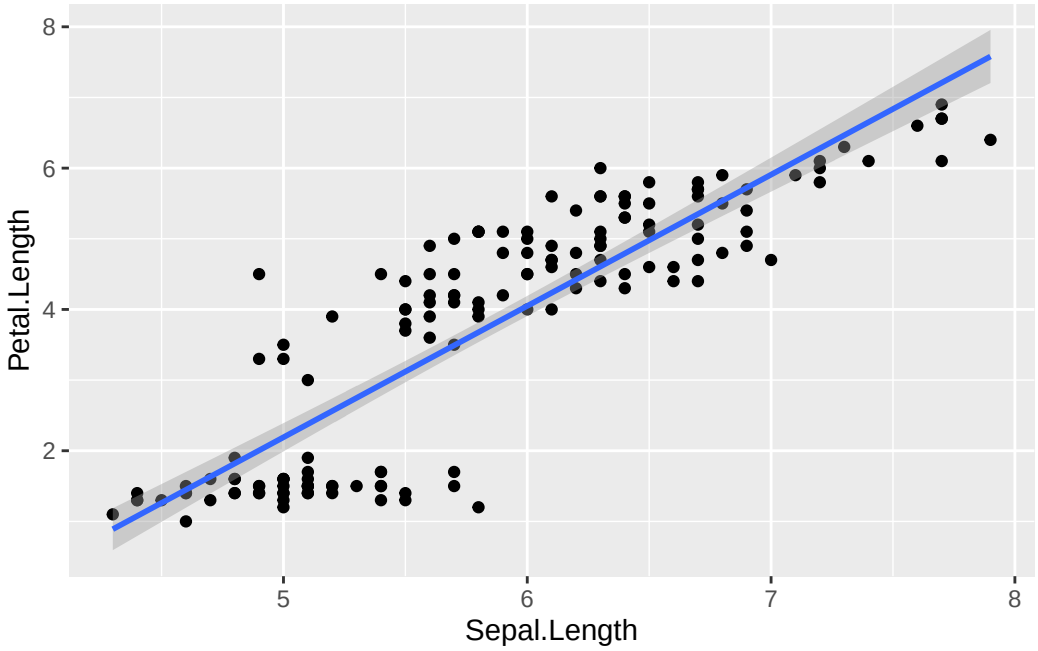
```
library(tidyverse)

ggplot(iris) +
  aes(x = Sepal.Length, y = Petal.Length) +
  geom_point() +
  geom_smooth()
```



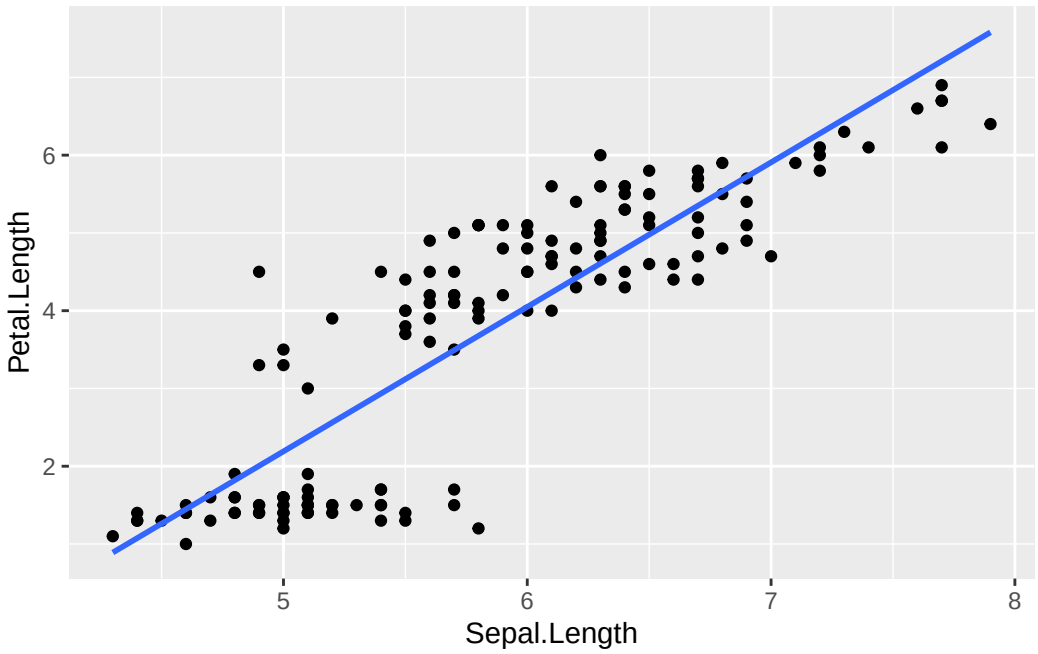
While such fits are occasionally useful, we often want a linear least-suqares regression on our data. To get such a linear fit, add the argument `method = "lm"` to `geom_smooth()` (`"lm"` stands for "linear model"):

```
ggplot(iris) +
  aes(x = Sepal.Length, y = Petal.Length) +
  geom_point() +
  geom_smooth(method = "lm")
```



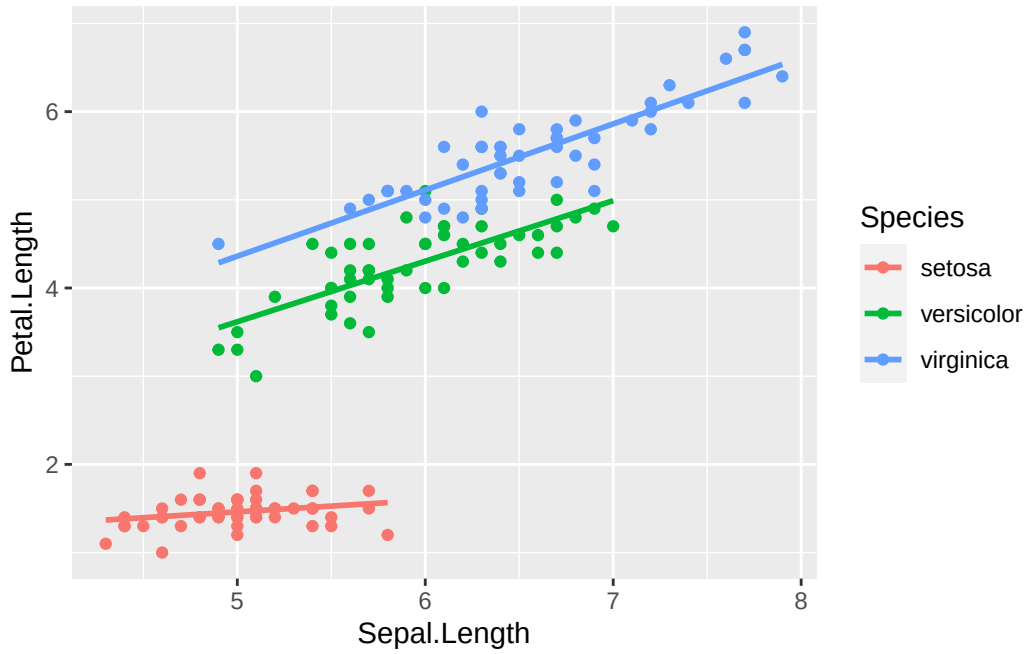
Linear regression lines are usually shown without the confidence intervals (the gray band around the regression line). To drop this, set `se = FALSE`:

```
ggplot(iris) +
  aes(x = Sepal.Length, y = Petal.Length) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
```



What happens if we color the data points by species? Let us add `colour = Species` to the list of aesthetic mappings:

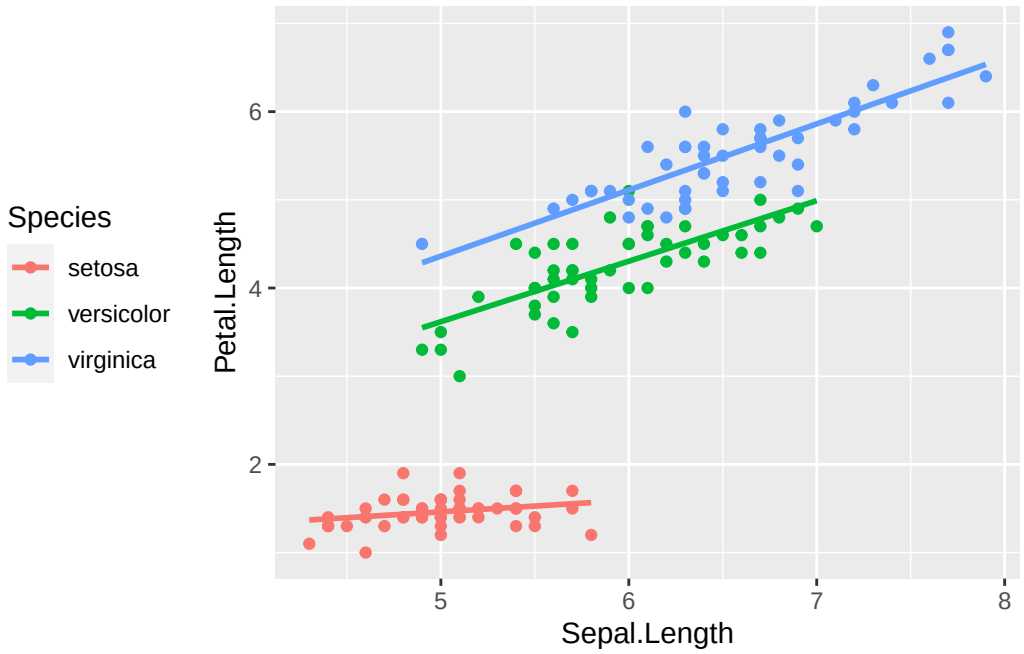
```
ggplot(iris) +
  aes(x = Sepal.Length, y = Petal.Length, colour = Species) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
```



Now the regression line is automatically fitted to the data within each of the groups separately—a highly useful behavior.

Notice also that a color legend was automatically created and put to the right of the graph. This is the default in `ggplot2`. You can move them to another position by specifying the `legend.position` option within the `theme` function that can be added onto the plot:

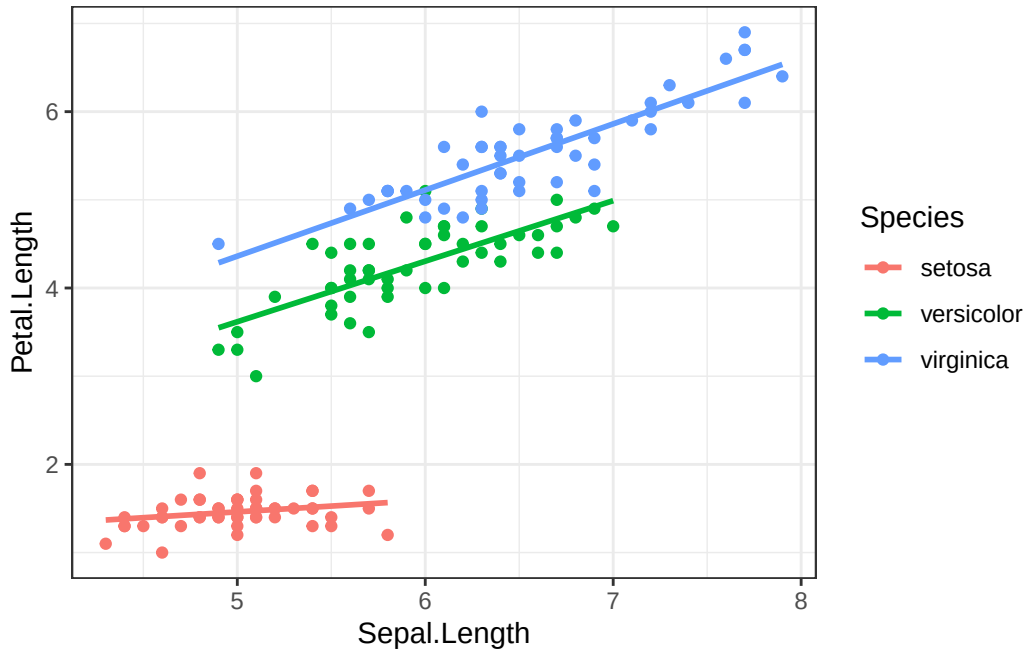
```
ggplot(iris) +
  aes(x = Sepal.Length, y = Petal.Length, colour = Species) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  theme(legend.position = "left") # "top", "bottom", "left", "right", or "none"
```



Specifying `legend.position = "none"` omits the legend altogether.

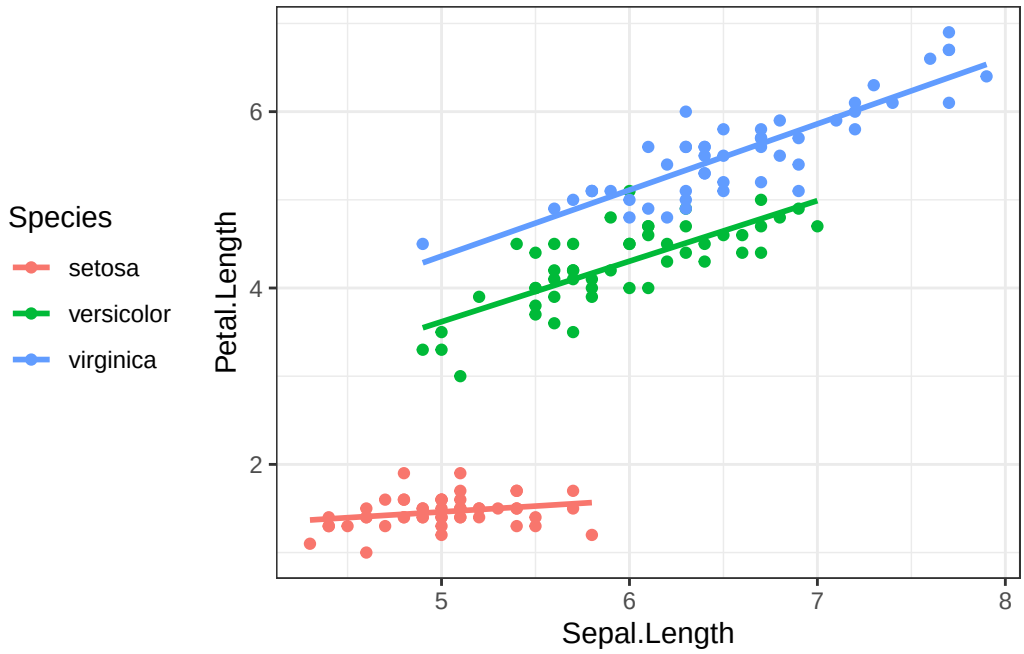
A word of caution: in case the legend positioning is matched with a generic theme such as `theme_bw()`, one should put the legend position *after* the main theme definition. The reason is that pre-defined themes like `theme_bw()` override any specific theme options you might specify. The rule of thumb is: any `theme()` component to your plot should be added only after the generic theme definition. Otherwise the `theme()` component will be overridden and will not take effect. For example, this does not work as intended:

```
ggplot(iris) +
  aes(x = Sepal.Length, y = Petal.Length, colour = Species) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  theme(legend.position = "left") + # Position legend at the left
  theme_bw() # This defines the general theme - and thus overrides the line above...
```



But this one does:

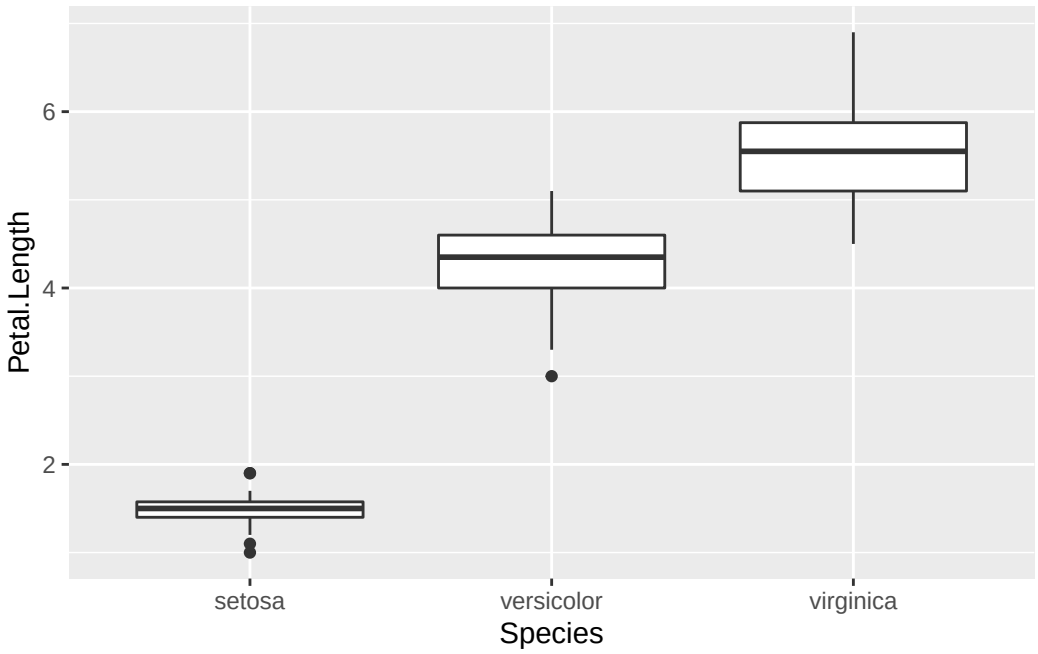
```
ggplot(iris) +
  aes(x = Sepal.Length, y = Petal.Length, colour = Species) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  theme_bw() + # This defines the general theme
  theme(legend.position = "left") # We now override the default legend positioning
```



7.2 Scales

The aesthetic mappings of a graph (x-axis, y-axis, color, fill, size, shape, alpha, …) are automatically rendered into the displayed plot, based on certain default settings within `ggplot2`. These defaults can be altered, however. Consider the following bare-bones plot:

```
iris %>%
  ggplot() +
  aes(x = Species, y = Petal.Length) +
  geom_boxplot()
```

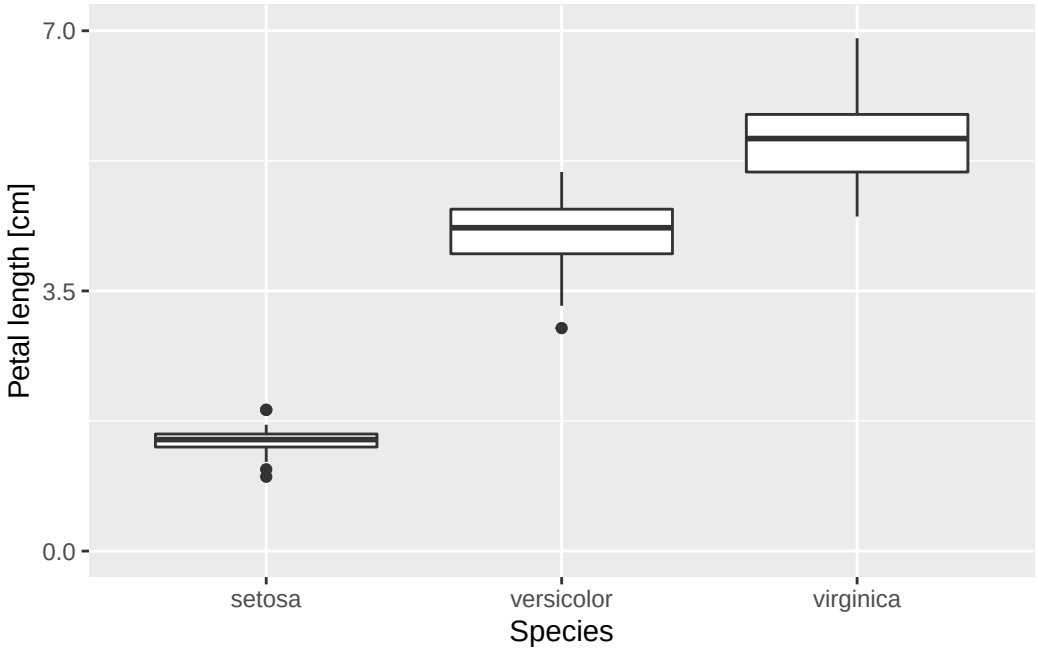


We can now change, for example, how the y-axis is displayed. The component to be added to the plot is `scale_y_continuous()`. Here `scale` means we are going to change the scaling of some aesthetic mapping, `y` refers to the y-axis (as expected, it can be replaced with `x`, `colour`, `fill`, etc.), and `continuous` means that the scaling of the axis is not via discrete values (e.g., either 1 *or* 2 *or* 3 but nothing in between), but continuous (every real number is permissible along the y-axis). The plot component `scale_y_continuous()` takes several arguments; take a look look at its `help` page to see all possible options. Here we mention a few of these. First, there is the `name` option, which is used to relabel the axis. The `limits` argument receives a vector of two values, containing the lower and upper limits of the plot. If any of them is set to `NA`, the corresponding limit will be determined automatically. Next, the `breaks` argument controls where the tick marks along the axis go. It is given as a vector, with its entries corresponding to the y-coordinates of the tick marks. Finally, `labels` determines what actually gets written on the axis at the tick mark points—it is therefore also a vector, its length matching that of `breaks`.

As an example, let us scale the y-axis of the previous graph in the following way. The axis label should read "Petal length [cm]", instead of the current "Petal.Length". It should go from 0 to 7, with a break at those two values and also halfway in between at 3.5. Here is how to do this:

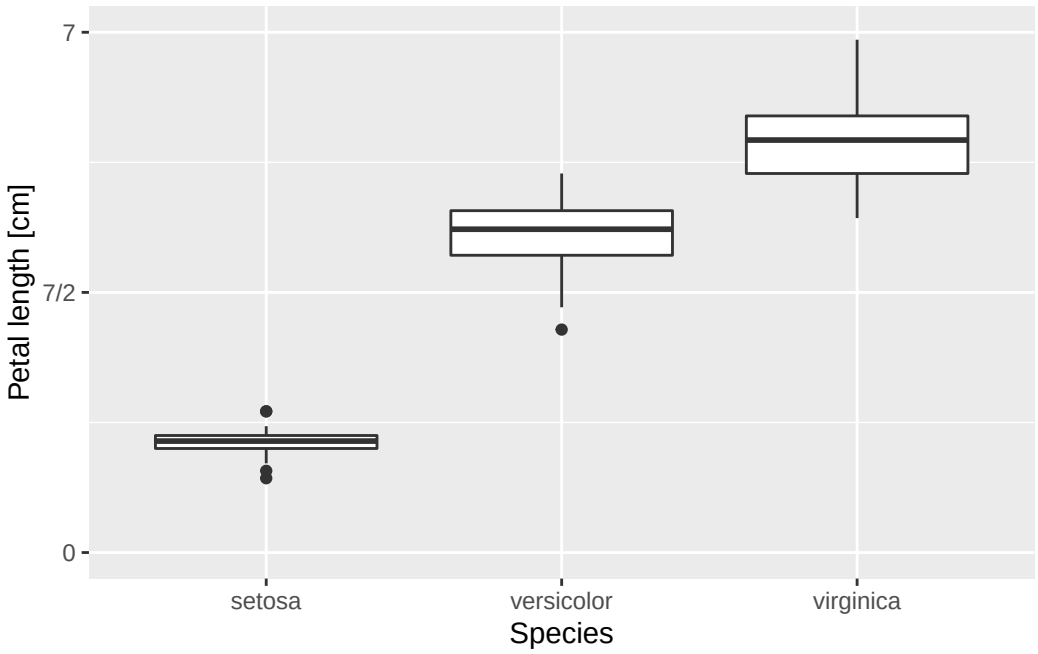
```
iris %>%
  ggplot() +
  aes(x = Species, y = Petal.Length) +
```

```
  geom_boxplot() +
  scale_y_continuous(name = "Petal length [cm]",
                     limits = c(0, 7),
                     breaks = c(0, 3.5, 7))
```



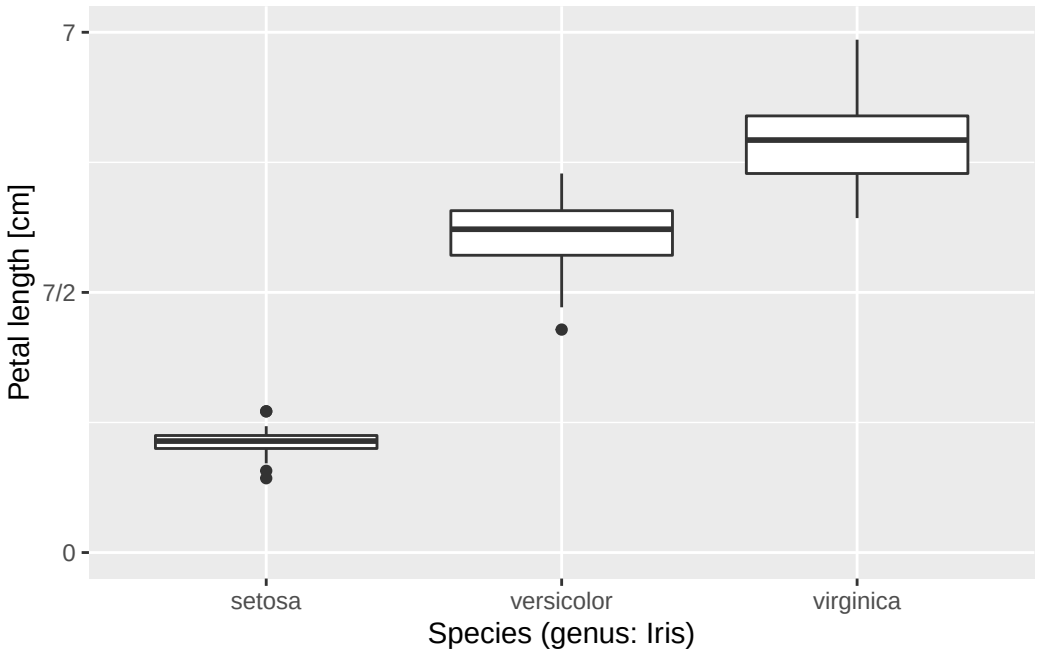
What should we do if, for some reason, we would additionally like the "3.5" in the middle to be displayed as "7/2" instead (an exact value)? In that case, we can add an appropriate `labels` option as an argument to `scale_y_continuous`:

```
iris %>%
  ggplot() +
  aes(x = Species, y = Petal.Length) +
  geom_boxplot() +
  scale_y_continuous(name = "Petal length [cm]",
                     limits = c(0, 7),
                     breaks = c(0, 3.5, 7),
                     labels = c("0", "7/2", "7"))
```



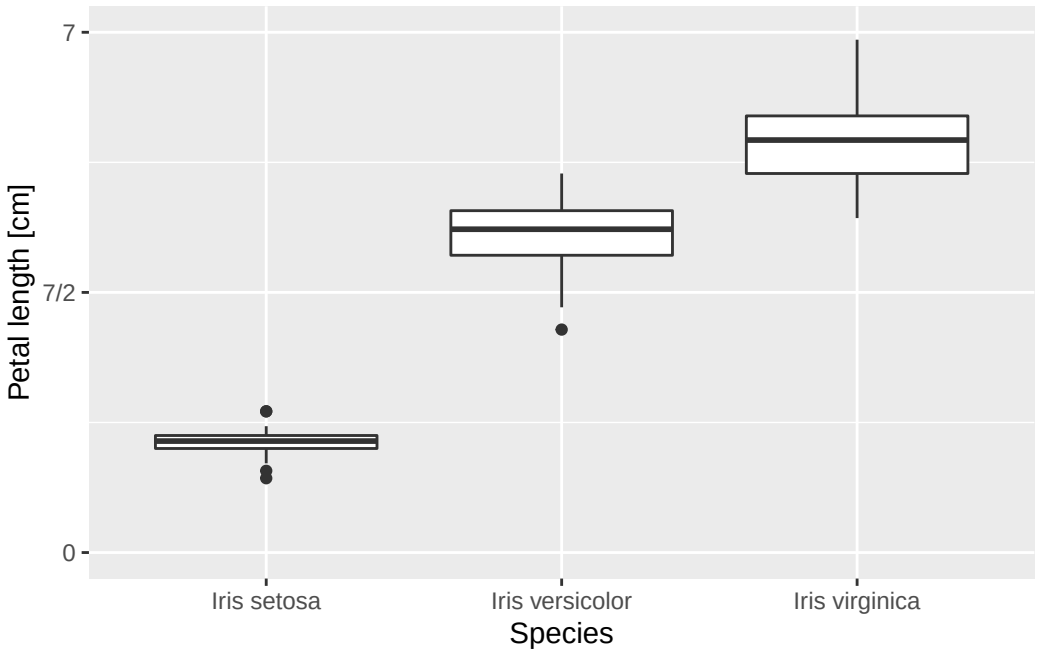
The x-axis can be scaled similarly. One important difference though is that here, the x-axis has a *discrete* scale. Since we are displaying the species along it, any value must be either *setosa* or *versicolor* or *virginica*; it makes no sense to talk about what is "halfway in between *setosa* and *versicolor*". Therefore, one should use `scale_x_discrete()`. Its options are similar to those of `scale_x_continuous()`. For instance, let us override the axis label, spelling out that the three species belong to the genus *Iris*:

```
iris %>%
  ggplot() +
  aes(x = Species, y = Petal.Length) +
  geom_boxplot() +
  scale_y_continuous(name = "Petal length [cm]",
                     limits = c(0, 7),
                     breaks = c(0, 3.5, 7),
                     labels = c("0", "7/2", "7")) +
  scale_x_discrete(name = "Species (genus: Iris)")
```



Alternatively, one could also redefine the labels and get an equally good graph:

```
iris %>%
  ggplot() +
  aes(x = Species, y = Petal.Length) +
  geom_boxplot() +
  scale_y_continuous(name = "Petal length [cm]",
                     limits = c(0, 7),
                     breaks = c(0, 3.5, 7),
                     labels = c("0", "7/2", "7")) +
  scale_x_discrete(labels = c("Iris setosa", "Iris versicolor", "Iris virginica"))
```

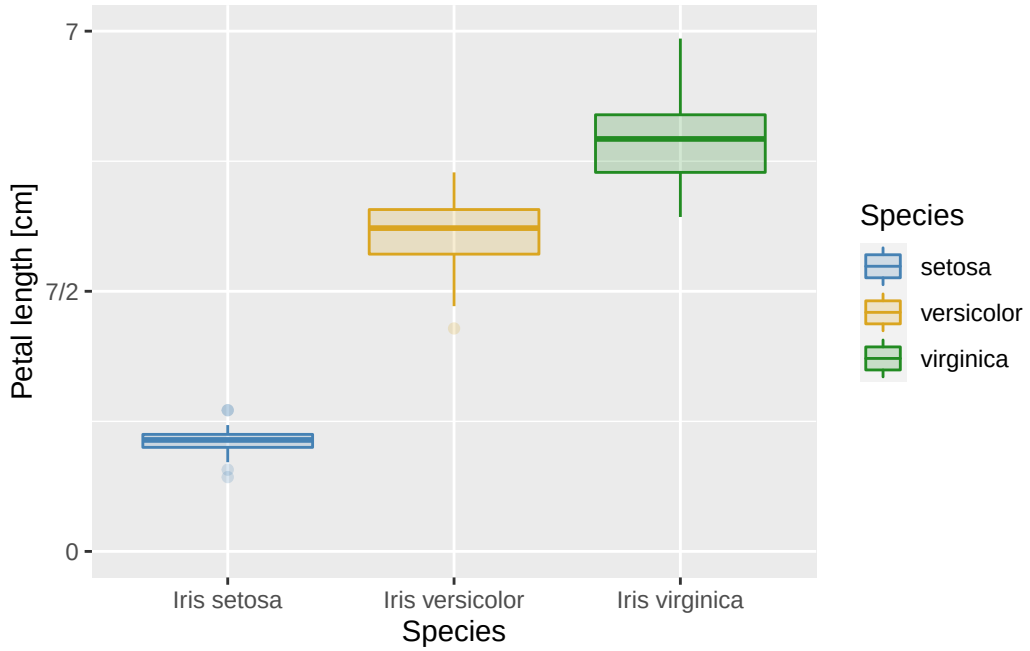


> **Note**
>
> In case you would like to display the species names in *italics*, as is standard requirement when writing binomial nomenclature, feel free to add `theme(axis.text.x = element_text(face = "italic"))` to the end of the plot. We will not be going into more detail on tweaking themes, but feel free to explore the possibilities by looking at the `help` pages or Googling them.

Other aesthetic mappings can also be adjusted, such as `colour`, `fill`, `size`, or `alpha`. One useful way to do it is through `scale_colour_manual()`, `scale_fill_manual()`, and so on. These are like `scale_colour_discrete()`, `scale_fill_discrete()` etc., except that they allow one to specify a discrete set of values by hand. Let us do this for color and fill:

```
iris %>%
  ggplot() +
  aes(x = Species, y = Petal.Length, colour = Species, fill = Species) +
  geom_boxplot(alpha = 0.2) +
  scale_y_continuous(name = "Petal length [cm]",
                     limits = c(0, 7),
                     breaks = c(0, 3.5, 7),
                     labels = c("0", "7/2", "7")) +
  scale_x_discrete(labels = c("Iris setosa", "Iris versicolor", "Iris virginica")) +
```

```
  scale_colour_manual(values = c("steelblue", "goldenrod", "forestgreen")) +
  scale_fill_manual(values = c("steelblue", "goldenrod", "forestgreen"))
```

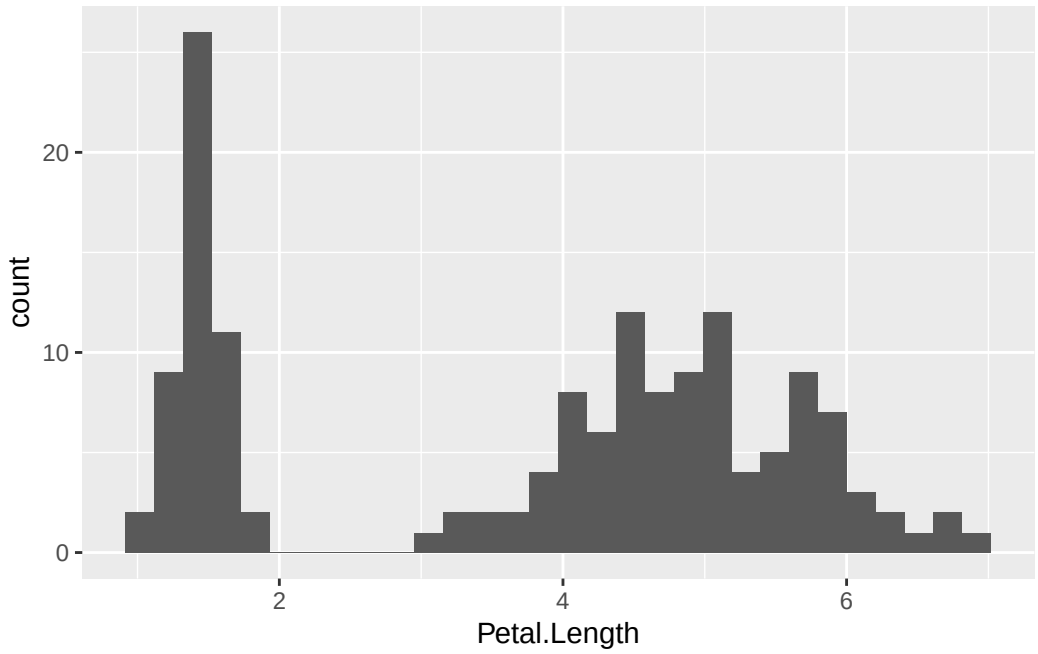


We used the built-in color names `"steelblue"`, `"goldenrod"`, and `"forestgreen"` above. A full R color cheat sheet can be found here, for more options and built-in colors.

7.3 Facets

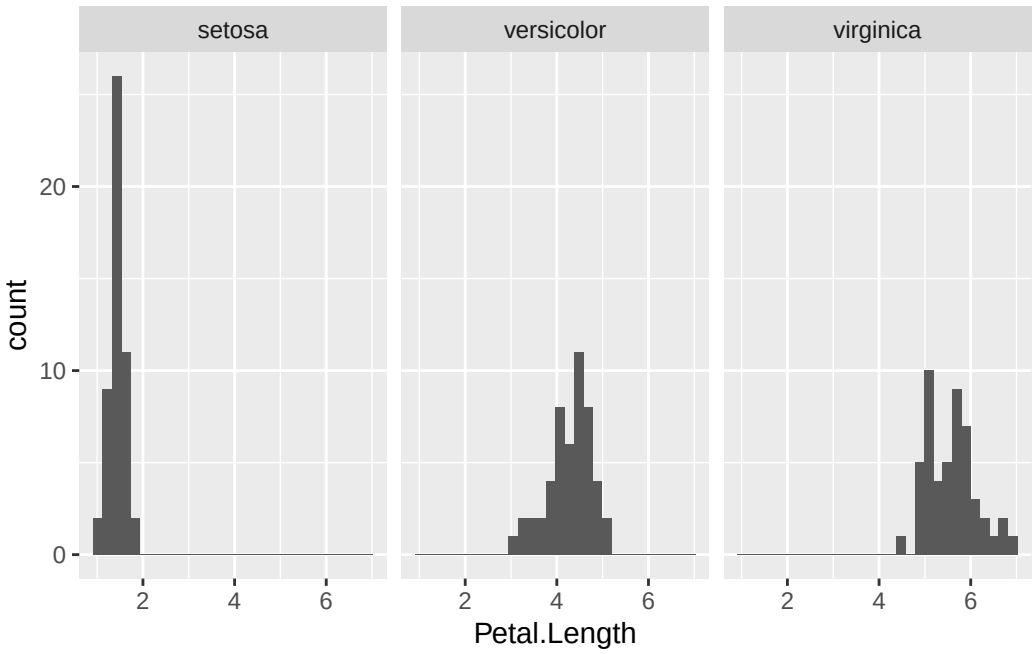
Plots can be *faceted* (subplots created and arranged in a grid layout) based on some variable or variables. For instance, let us create histograms of petal lengths in the `iris` dataset, like we did last time:

```
iris %>%
  ggplot() +
  aes(x = Petal.Length) +
  geom_histogram()
```



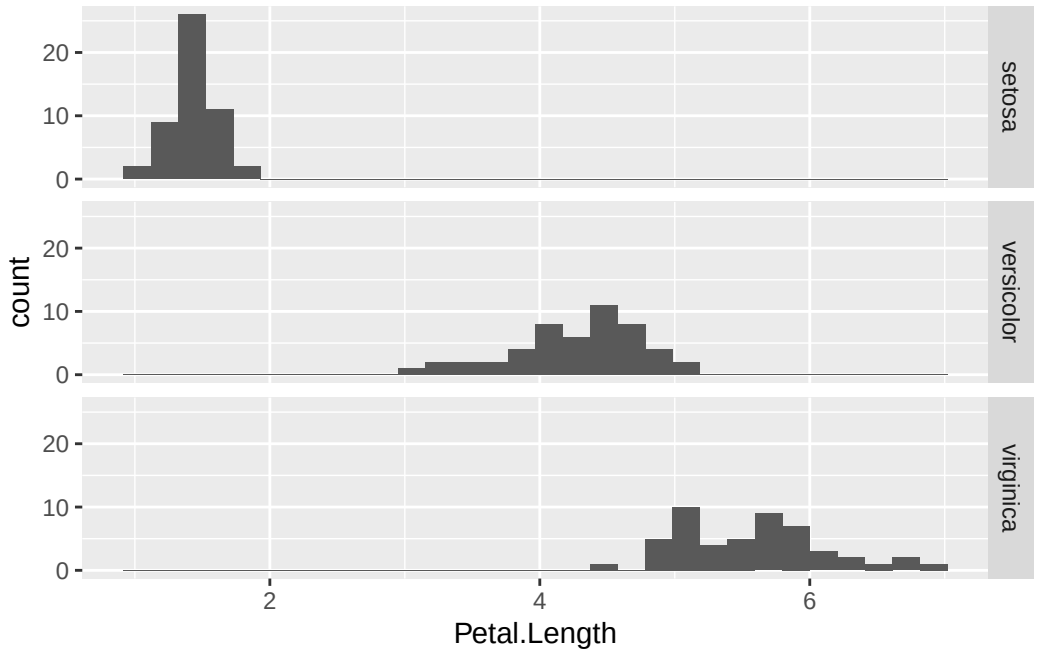
This way, one cannot see which part of the histogram belongs to which species. One fix to this is to color the histogram by species—this is what we have done before. Another is to separate the plot into three facets, each displaying data for one of the species only:

```
iris %>%
  ggplot() +
  aes(x = Petal.Length) +
  geom_histogram() +
  facet_grid(. ~ Species)
```



The component `facet_grid(x ~ y)` means that the data will be grouped based on columns `x` and `y`, with the distinct values of column `x` making up the rows and those of column `y` the columns of the grid of plots. If one of them is replaced with a dot (as above), then that variable is ignored, and only the other variable is used in creating a row (or column) of subplots. So, to display the same data but with the facets arranged in one column instead of one row, we simply replace `facet_grid(. ~ Species)` with `facet_grid(Species ~ .)`:

```
iris %>%
  ggplot() +
  aes(x = Petal.Length) +
  geom_histogram() +
  facet_grid(Species ~ .)
```



In this particular case, the above graph is preferable to the previous one, because the three subplots now share the same x-axis. This makes it easier to compare the distribution of petal lengths across the species.

To illustrate how to make a two-dimensional grid of facets, let us normalize the `iris` dataset using `pivot_longer()`:

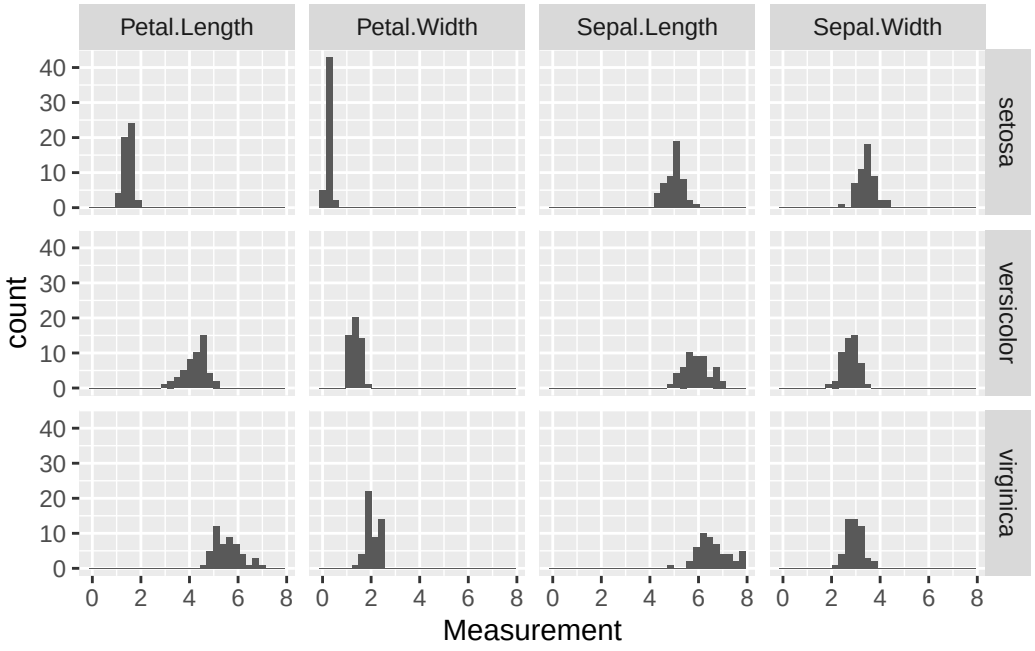
```
as_tibble(iris) %>%
  pivot_longer(cols = c(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width),
               names_to = "Trait",
               values_to = "Measurement")
```

```
# A tibble: 600 x 3
   Species Trait        Measurement
   <fct>   <chr>              <dbl>
 1 setosa  Sepal.Length         5.1
 2 setosa  Sepal.Width          3.5
 3 setosa  Petal.Length         1.4
 4 setosa  Petal.Width          0.2
 5 setosa  Sepal.Length         4.9
 6 setosa  Sepal.Width          3
 7 setosa  Petal.Length         1.4
 8 setosa  Petal.Width          0.2
```

```
 9 setosa  Sepal.Length         4.7
10 setosa  Sepal.Width          3.2
# ... with 590 more rows
```

As seen, now the `Measurement` in every row is characterized by two other variables: `Species` and `Trait` (i.e., whether the given value refers to the sepal length, petal width etc. of the given species). We can create a histogram of each measured trait for each species now, in a remarkably simple way:

```
as_tibble(iris) %>%
  pivot_longer(cols = c(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width),
               names_to = "Trait",
               values_to = "Measurement") %>%
  ggplot() +
  aes(x = Measurement) +
  geom_histogram() +
  facet_grid(Species ~ Trait)
```



7.4 Saving plots

To save the most recently created ggplot figure, simply type

```r
ggsave(filename = "graph.pdf", width = 4, height = 3)
```

Here `filename` is the name (with path and extension) of the file you want to save the figure into. The extension is important: by having specified `.pdf`, the system automatically saves the figure in PDF format. To use, say, PNG instead:

```r
ggsave(filename = "graph.png", width = 4, height = 3)
```

Since PDF is a vectorized file format (i.e., the file contains the instructions for generating the plot elements instead of a pixel representation), it is arbitrarily scalable, and is therefore the preferred way of saving and handling scientific graphs.

The `width` and `height` parameters specify, in inches, the dimensions of the saved plot. Note that this also scales some other plot elements, such as the size of the axis labels and plot legends. This means you can play with the `width` and `height` parameters to save the figure at a size where the labels are clearly visible without being too large.

In case you would like to save a figure that is not the last one that was generated, you can specify the `plot` argument to `ggsave()`. to do so, first you should assign a plot to a variable. For example:

```r
p <- ggplot(iris) + # Assign the ggplot object to the variable p
  aes(x = Petal.Length) +
  geom_histogram()
```

and then

```r
ggsave(filename = "graph.pdf", plot = p, width = 4, height = 3)
```

8 Joining data

8.1 Merging two related tables into one

So far, we have been working with a single table of data at a time. Often however, information about the same thing is scattered across multiple tables and files. In such cases, we sometimes want to *join* those separate tables into a single one. To illustrate how this can be done, let us create two simple tables. The first will contain the names of students, along with their chosen subject:

```
library(tidyverse)

studies <- tibble(name    = c("Sacha", "Gabe", "Alex"),
                  subject = c("Physics", "Chemistry", "Biology"))
print(studies)
```

```
# A tibble: 3 x 2
  name  subject
  <chr> <chr>
1 Sacha Physics
2 Gabe  Chemistry
3 Alex  Biology
```

The second table contains slightly different information: it holds which year a given student is currently into their studies.

```
stage <- tibble(name = c("Sacha", "Alex", "Jamie"),
                year = c(3, 1, 2))
print(stage)
```

```
# A tibble: 3 x 2
  name   year
  <chr> <dbl>
1 Sacha     3
```

```
2 Alex      1
3 Jamie     2
```

Notice that, while Sacha and Alex appear in both tables, Gabe is only included in `studies` and Jamie only in `stage`. While in such tiny datasets this might seem like an avoidable oversight, such non-perfect alignment of data can be the norm when working with data spanning hundreds, thousands, or more rows. Here, for the purposes of illustration, we use small tables, but the principles we learn here apply in a broader context as well.

There are four commonly used ways of joining these tables into one single dataset. All of them follow the same general pattern: the arguments are two tibbles (or data frames) to be joined, plus a `by =` argument which lists the name(s) of the column(s) based on which the tables should be joined. The output is always a single tibble (data frame), containing some type of joining of the data. Let us now look at each joining method in detail.

8.1.1 `left_join`

The `left_join` function keeps only those rows that appear in the *first* of the two tables to be joined:

```
left_join(studies, stage, by = "name")
```

```
# A tibble: 3 x 3
  name  subject    year
  <chr> <chr>     <dbl>
1 Sacha Physics       3
2 Gabe  Chemistry    NA
3 Alex  Biology       1
```

There are two things to notice. First, Jamie is missing from the `name` column above, even though s/he did appear in the `stage` tibble. This is exactly the point of `left_join`: if a row entry in the joining column (specified in `by =`) does not appear in the *first* table listed in the arguments (here, the `name` column of `studies`), then it is omitted. Second, the `year` entry for Gabe is `NA`. This is because Gabe is absent from the `stage` table, and therefore has no associated year of study. Rather than make up nonsense, R fills out such missing data with `NA` values.

8.1.2 `right_join`

This function works just like `left_join`, except only those rows are retained which appear in the *second* of the two tables to be joined:

```
right_join(studies, stage, by = "name")
```

```
# A tibble: 3 x 3
  name  subject  year
  <chr> <chr>   <dbl>
1 Sacha Physics     3
2 Alex  Biology     1
3 Jamie <NA>        2
```

In other words, this is exactly the same as calling `left_join` with its first two arguments reversed:

```
left_join(stage, studies, by = "name")
```

```
# A tibble: 3 x 3
  name   year subject
  <chr> <dbl> <chr>
1 Sacha     3 Physics
2 Alex      1 Biology
3 Jamie     2 <NA>
```

The only difference is in the ordering of the columns, but the data contained in the tables are identical.

In this case, the columns `subject` is `NA` for Jamie. The reason is the same as it was before: since the `studies` table has no `name` entry for Jamie, the corresponding subject area is filled in with a missing value `NA`.

8.1.3 `inner_join`

This function retains only those rows which appear in *both* tables to be joined. For our example, since Gabe only appears in `studies` and Jamie only in `stage`, they will be dropped by `inner_join` and only Sacha and Alex are retained (since they appear in both tables):

```
inner_join(studies, stage, by = "name")
```

```
# A tibble: 2 x 3
  name  subject  year
  <chr> <chr>   <dbl>
1 Sacha Physics     3
2 Alex  Biology     1
```

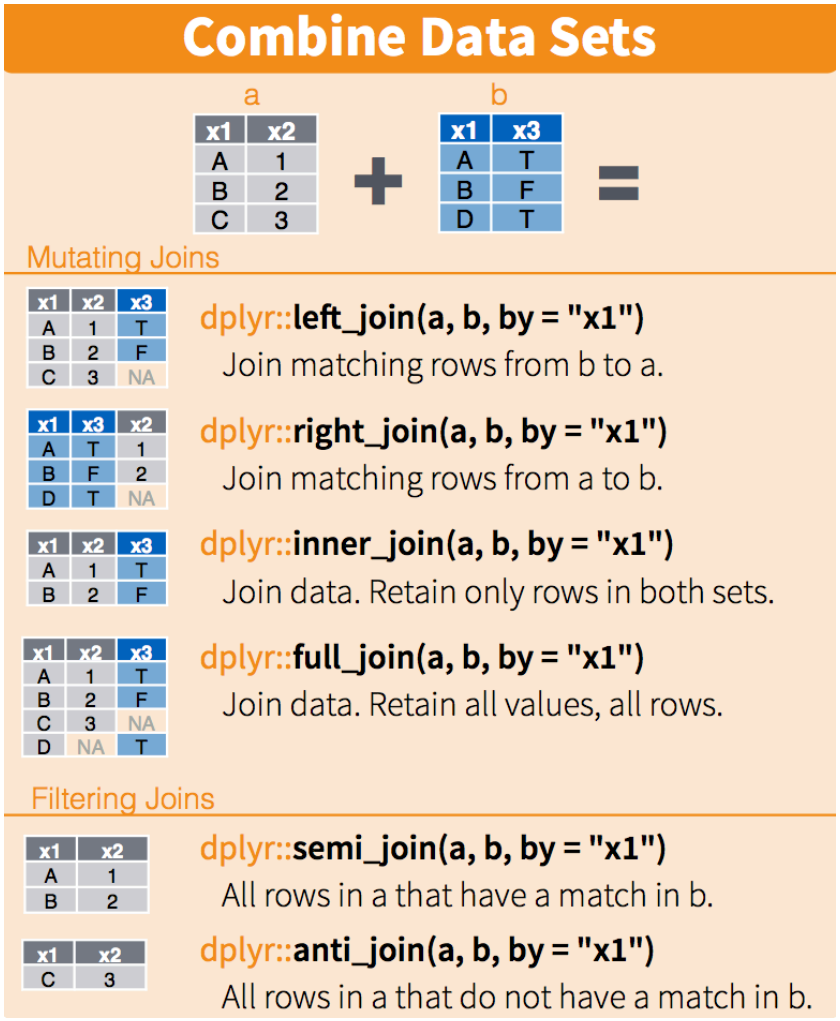
8.1.4 `full_join`

The complement to `inner_join`, this function retains all rows in all tables, filling in missing values with `NA`s everywhere:

```
full_join(studies, stage, by = "name")
```

```
# A tibble: 4 x 3
  name  subject    year
  <chr> <chr>     <dbl>
1 Sacha Physics       3
2 Gabe  Chemistry    NA
3 Alex  Biology       1
4 Jamie <NA>          2
```

A useful table summarizing these options, taken from a more comprehensive (though slightly out-of-date) cheat sheet, is below:



(As you see, apart from the four so-called *mutating joins* we have learned about, there are also two *filtering joins* included in this cheat sheet as well. We will not be covering those here, but feel free to check out their help pages.)

8.1.5 Joining by multiple columns

It is also possible to use the above joining functions specifying multiple columns to join data by. To illustrate how to do this and what this means, imagine that we slightly modify the student data. The first table will contain the name, study area, and year of study for each student. The second table will contain the name and study area of each student, plus whether they have passed their most recent exam:

```r
program  <- tibble(name     = c("Sacha", "Gabe", "Alex"),
                   subject  = c("Physics", "Chemistry", "Biology"),
                   year     = c(1, 3, 2))
print(program)
```

```
# A tibble: 3 x 3
  name  subject    year
  <chr> <chr>     <dbl>
1 Sacha Physics       1
2 Gabe  Chemistry     3
3 Alex  Biology       2
```

```r
progress <- tibble(name     = c("Sacha", "Gabe", "Jamie"),
                   subject  = c("Physics", "Chemistry", "Biology"),
                   examPass = c(TRUE, FALSE, TRUE))
print(progress)
```

```
# A tibble: 3 x 3
  name  subject   examPass
  <chr> <chr>     <lgl>
1 Sacha Physics   TRUE
2 Gabe  Chemistry FALSE
3 Jamie Biology   TRUE
```

And now, since the tables share not just one but two columns, it makes sense to join them using both. This can be done by specifying each column inside a vector in the `by =` argument. For example, left-joining `program` and `progress` by both `name` and `subject` leads to a joint table in which all unique `name-subject` combinations found in `program` are retained, but those found only in `progress` are discarded:

```r
left_join(program, progress, by = c("name", "subject"))
```

```
# A tibble: 3 x 4
  name  subject    year examPass
  <chr> <chr>     <dbl> <lgl>
1 Sacha Physics       1 TRUE
2 Gabe  Chemistry     3 FALSE
3 Alex  Biology       2 NA
```

> ⚠ Warning
>
> As mentioned, the columns by which one joins the tables must be included in a vector. Forgetting to do this will either fail or produce faulty output:

```
left_join(program, progress, by = "name", "subject")
```

```
# A tibble: 3 x 5
  name  subject.x  year subject.y examPass
  <chr> <chr>     <dbl> <chr>     <lgl>
1 Sacha Physics       1 Physics   TRUE
2 Gabe  Chemistry     3 Chemistry FALSE
3 Alex  Biology       2 <NA>      NA
```

> The problem is that the comma after `name` is interpreted as the start of the next argument to `left_join`, instead of being a part of the `by =` argument. Again, the solution is simple: enclose multiple column names in a vector.

The other joining functions also work as expected:

```
right_join(program, progress, by = c("name", "subject"))
```

```
# A tibble: 3 x 4
  name  subject    year examPass
  <chr> <chr>     <dbl> <lgl>
1 Sacha Physics       1 TRUE
2 Gabe  Chemistry     3 FALSE
3 Jamie Biology      NA TRUE
```

```
inner_join(program, progress, by = c("name", "subject"))
```

```
# A tibble: 2 x 4
  name  subject    year examPass
  <chr> <chr>     <dbl> <lgl>
1 Sacha Physics       1 TRUE
2 Gabe  Chemistry     3 FALSE
```

```
full_join(program, progress, by = c("name", "subject"))
```

```
# A tibble: 4 x 4
  name  subject    year examPass
  <chr> <chr>     <dbl> <lgl>
1 Sacha Physics       1 TRUE
2 Gabe  Chemistry     3 FALSE
3 Alex  Biology       2 NA
4 Jamie Biology      NA TRUE
```

8.2 Binding rows and columns to a table

Occasionally, a simpler problem presents itself: there is a single dataset, but its rows are contained across separate tables. For example, a table containing student names and subject areas might be spread across two tables, like this:

```
studies1 <- tibble(name    = c("Sacha", "Gabe", "Alex"),
                   subject = c("Physics", "Chemistry", "Biology"))
print(studies1)
```

```
# A tibble: 3 x 2
  name  subject
  <chr> <chr>
1 Sacha Physics
2 Gabe  Chemistry
3 Alex  Biology
```

```
studies2 <- tibble(name    = c("Jamie", "Ashley", "Dallas", "Jordan"),
                   subject = c("Geology", "Mathematics", "Philosophy", "Physics"))
print(studies2)
```

```
# A tibble: 4 x 2
  name   subject
  <chr>  <chr>
1 Jamie  Geology
2 Ashley Mathematics
3 Dallas Philosophy
4 Jordan Physics
```

The tables have the exact same structure, in that the column names and types are identical. It's just that the rows are, for some reason, disparate. To combine them together, we could recourse to full-joining the tables by both their columns:

```
full_join(studies1, studies2, by = c("name", "subject"))
```

```
# A tibble: 7 x 2
  name   subject
  <chr>  <chr>
1 Sacha  Physics
2 Gabe   Chemistry
3 Alex   Biology
4 Jamie  Geology
5 Ashley Mathematics
6 Dallas Philosophy
7 Jordan Physics
```

This, however, is not necessary. Whenever all we need to do is take two tables and stick their rows together, there is the simpler `bind_rows`:

```
bind_rows(studies1, studies2)
```

```
# A tibble: 7 x 2
  name   subject
  <chr>  <chr>
1 Sacha  Physics
2 Gabe   Chemistry
3 Alex   Biology
4 Jamie  Geology
5 Ashley Mathematics
6 Dallas Philosophy
7 Jordan Physics
```

Similarly, in case two tables have the same number of rows but different columns, one can stick their columns together using `bind_cols`. For example, suppose we have

```
studies <- tibble(name    = c("Sacha", "Gabe", "Alex"),
                  subject = c("Physics", "Chemistry", "Biology"))
print(studies)
```

```
# A tibble: 3 x 2
  name  subject
  <chr> <chr>
1 Sacha Physics
2 Gabe  Chemistry
3 Alex  Biology
```

as well as a table with year of study and result of last exam only:

```
yearExam <- tibble(year     = c(3, 1, 2),
                   examPass = c(FALSE, TRUE, TRUE))
print(yearExam)
```

```
# A tibble: 3 x 2
   year examPass
  <dbl> <lgl>
1     3 FALSE
2     1 TRUE
3     2 TRUE
```

We can now join these using `bind_cols`:

```
bind_cols(studies, yearExam)
```

```
# A tibble: 3 x 4
  name  subject    year examPass
  <chr> <chr>     <dbl> <lgl>
1 Sacha Physics       3 FALSE
2 Gabe  Chemistry     1 TRUE
3 Alex  Biology       2 TRUE
```

8.3 Exercises

We have used the data of Fauchald et al. (2017)[1] before in other exercises. As a reminder, they tracked the population size of various herds of caribou in North America over time, and correlated population cycling with the amount of vegetation and sea-ice cover. Two files from

[1]Fauchald, P. et al. (2017). Arctic greening from warming promotes declines in caribou populations. *Science Advances*, 3:e1601365.

their data are on Lisam: `pop_size.tsv` (herd population sizes), and `sea_ice.tsv` (sea ice cover per year and month).

1. Load these two datasets into two variables. They could be called `pop` and `ice`, for instance. Look at the data to familiarize yourself with them. How many rows and columns are in each?
2. Before doing anything else: how many rows will there be in the table that is the left join of `pop` and `ice`, based on the two columns `Herd` and `Year`? Perform the left join to see if you were correct. Where do you see `NA`s in the table, and why?
3. Now do the same with right-joining, inner-joining, and full-joining `pop` and `ice`.