Introduction to coding

HTML Version: https://bradduthie.github.io/notes/R_intro_notes.html

Brad Duthie

19 OCT 2022

Contents

After reading through this, you should have a working understanding of what version control is and why it is useful. You should also be able to start using version control in your own work through a combination of GitHub and either GitKraken or the command line interface.

The purpose of this introduction

The purpose here is to get readers past the initial learning curve of coding as quickly as possible. If you want to start coding for yourself, particularly in R for data analysis, but are not sure how, then read on. By the end of these notes, you should be able to navigate through the basic graphical user interface of Rstudio and write some basic lines of code. The goal is not to develop proficiency in coding or R yet, but to help you get to the point at which it is possible to write and run code, make coding mistakes, and learn from other researcher's code.

Why use the R programming language?

The computer programming language R is a powerful and very widely-used tool among scientists for analysing data. You can use it to analyse and plot data, run computer simulations, or even write slides, papers, or books. The R programming language is completely free and open source, as is the popular Rstudio software for using it. It specialises in statistical computing, which is part of the reason for its popularity among scientists.

Another reason for its popularity is its versatility, and the ease with which new techniques can be shared. Imagine that you develop a new method for analysing data. If you want other researchers to be able to use your method in their research, then you could write your own software from scratch for them to install and use. But doing this would be very time consuming, and a lot of that time would likely be spent writing the graphical user interface and making sure that your program worked across platforms (e.g., on Windows and Mac). Worse, once written, there would be no easy way to make your program work with other statistical software should you need to integrate different analyses or visualisation tools (e.g., plotting data). To avoid all of this, you could instead just present your new method for data analysis and let other researchers write their own code for implementing it. But not all researchers will have the time or expertise to do this.

Instead, R allows researchers to write new tools for data analysis using simple coding scripts. These scripts are organised into R packages, which can be uploaded by authors to the Comprehensive R Archive Network (CRAN), then downloaded by users with a single command in R. This way, there is no need for completely different software to be used for different analyses – all analyses can be written and run in R.

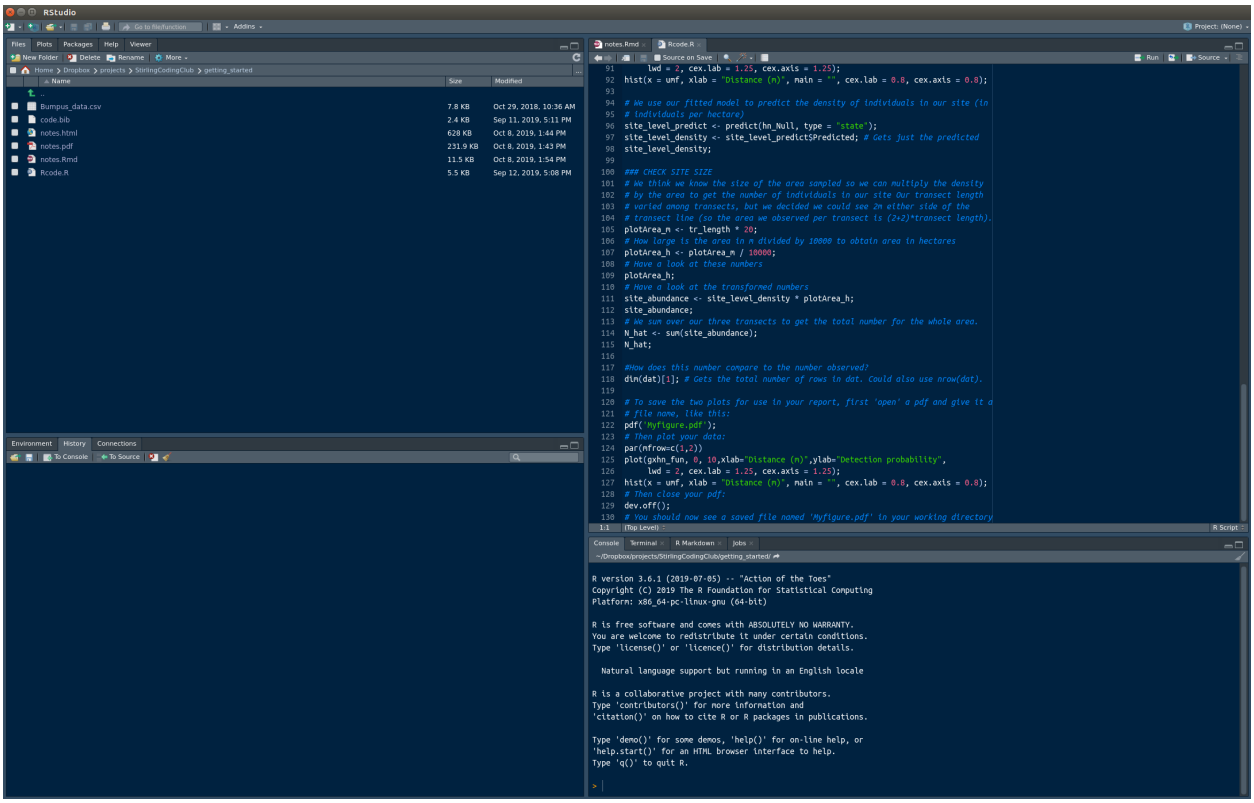
The downside to all of this is that learning R can be a bit daunting at first. Running analyses is not done by pointing and clicking on icons as in Excel, SigmaPlot, or JMP. You need to use code. Here we will start with the very basics and work our way up to some simple data analyses.

Getting started in R, and the basics

Installation. The first thing to do is download Rstudio if you have not already (but see below if you're eager to get started and want to skip this step). Note that R and Rstudio are not the same thing; R is a language for scientific computing, and can be used outside of Rstudio. Rstudio is a very useful tool for coding in the R language. As a very loose analogy, R is like a written language (e.g., English, Spanish) that can be used to write inside Rstudio (e.g., a word processor such as Microsoft Word, LibreOffice). Look carefully at the version of Rstudio that you download; different installers exist for Windows, Mac, and Linux. The most recent version of Rstudio requires a 64-bit operating system. Unless your computer is quite old (say, over seven years), you most likely have a 64-bit operating system rather than a 32-bit operating system, but if you are uncertain, then it is best to check.

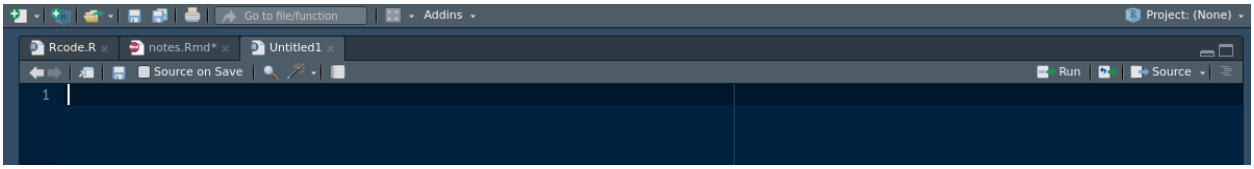
Bypassing installation with Rstudio Cloud. If you do not want to install R or Rstudio, or are having trouble doing so but want to get started in R quickly, then an alternative is to use R through the Rstudio cloud (https://rstudio.cloud). The Rstudio cloud allows you to run R right from your browser, and you can sign up for free. You can watch this five minute video to see how to sign up and get started.

Running Rstudio. When you first run Rstudio, you will see several windows open. It will look something like the below, except probably with a standard black on white theme (if you want, you can change this by selecting from the toolbar 'Tools > Global Options...', then selecting the 'Appearance' tab on the left).



This might look a bit intimidating at first. Unlike Microsoft Excel, SigmaPlot, or JMP, there is no spreadsheet that opens up for you. You interact with R mostly by typing lines of commands rather than using a mouse to point and click on different options. Eventually, this will feel liberating, but at first it will probably feel overwhelming. First, let us look at all of the four panes in the Figure above. Your panes might be organised a bit differently, but the important ones to start out with are the 'Source' and the 'Console'. These are shown in the right hand panes in the above Figure.

To make sure that the Source pane is available to you, open an new R script by selecting from the toolbar 'File > New File > Rscript' (shortcut: Shift+Ctrl+N). You should see a new Rscript open up that looks something like the below (again, the colour scheme might differ).



Think of this Source file like a Word document that you have just opened up – completely blank and ready for typing new lines of commands to read in data and run analyses. We will come back to this Source file, but for now just know that the Source file stores commands that we want R to intepret and use. The Source file does this by sending commands to the R console, which we will look at now.

Getting started in the R console

The R console should be located somewhere in Rstudio (I like to keep it directly underneath my R Source files). You can identify it by finding the standard R information printed off, which should look something like the below.

```
R version 4.2.2 (2022-10-31) -- "Innocent and Trusting"
Copyright (C) 2022 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

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

  Natural language support but running in an English locale

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

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

>
```

The console is where all of the code is run. To get started, you can actually ignore everything else and just focus on this pane. If you click to the right of the greater than sign `>` at the bottom, then you can start using R right in the console. To get a feel for running code in the console, you can use the R console as a standard calculator. Try typing something like the below to the right of the `>`, then hit 'Return' on your keyboard (note, all of my semi-colons are optional – you do not actually need to put them at the end of each line).

```
2 + 5;
```

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

Now try some other common mathematical operations, one line at a time.

```
4 * 4;
```

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

```
12 - 3;
```

```
## [1] 9
```

```
5^2;
```

```
## [1] 25
```

Notice that R does the calculation of each of the above mathematical operations and returns the correct value on the line below. If you are familiar with using Microsoft Excel, this is the equivalent to typing [`= 2 + 5`], or [`= 4 * 4`], etc., into a cell of an Excel spreadsheet. You might also be familiar with spreadsheet functions as well, such as the square root function, which you could use in Excel by typing, e.g., [`= sqrt(25)`] into a spreadsheet cell. This works in the R console too; the functions actually have the same syntax, so you could type the below into the console and hit 'Enter'.

```
sqrt(256);
```

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

The console returns the correct answer 16. Similar functions exist for logarithms (`log`) and trigonmetric functions (e.g., `sin`, `cos`), as they do in Microsoft Excel. But this is just the beginning in R. Functions can be used to do any number of tasks. Some of these functions are built into the base R language; others are written by researchers and distributed in R packages, but you can also learn to write your own R functions to do any number of customised tasks. You will need to use functions in nearly every line of code you write (technically, even the `+`, `-`, etc., are also functions), so it is good to know the basics of how to use them.

Most functions are called with open and closed parentheses, as in the `sqrt(256)` above. The `sqrt` is the function, while the `256` is a function argument. An argument is a specific input to a function, and functions can take any number of arguments. For the `sqrt` function, only one argument is needed, but many arguments will have more than one argument. For example, if we want to take the logarithm of some number using the `log` function, we might need to specify the base. In this case, we clarify the number for which we want to compute the log (`x`) and the logarithm base (`base`). Let us say that we want to compute the logarithm of 256 in base 10.

```
log(x = 256, base = 10);
```

```
## [1] 2.40824
```

Note that some arguments are required, while some are optional. In the case of `log`, the first argument is required, but the base is actually optional. If we do not specify a `base`, then the function simply defaults to calculating the natural logarithm (i.e., base e). Hence, the below also works (note that we get a different answer because the bases differ).

```
log(x = 256);
```

```
## [1] 5.545177
```

In fact, we do not even need to specify the `x` because only one argument is needed for the `log` function. Hence, if only one argument is specified, the function just assumes that this argument is `x`. Try the below.

```
log(256);
```

```
## [1] 5.545177
```

Note that functions can be nested inside other functions, though this can get messy. For example, if you wanted to get the logarithm of the logarithm of 256, then you could write the below.

```
log( log(256) );
```

```
## [1] 1.712929
```

Also note that functions do not need to be mathematical in R; they do not even need to operate on numbers. One very useful function is the `help` function, which provides documentation for other R functions. If, for example, we were not sure what `log` did, or what arguments it accepted, then we could run the code below.

```
help(topic = log);
```

Try running the above line of code in the R console. You should see a description of the `log` function, along with some examples of how it is used and the two arguments (`x` and `base`) that it accepts. Anytime you get stuck with a function, you should be able to use the `help` function for clarification. You can even use a shortcut that returns the same as the `help(topic = log);` above.

```
?log;
```

We now have looked at three functions, `sqrt`, `log`, and `help`. If you have previous experience with Microsoft Excel spreadsheets, you should now be able to make the conceptual connection between typing `=sqrt(25)` into a spreadsheet cell and `sqrt(25)` into the R console. You should also recognise that R functions serve a much broader set of purposes in R. Next, we will move onto assigning variables in the R console.

Assigning variables in the R console

In R, we can also assign values to variables using the characters `<-` to make an arrow. Say, for example, that we wanted to make `var_1` equal 10.

```
var_1 <- 10;
```

We can now use `var_1` in the console.

```
var_1 * 5; # Multiplying the variable by 5
```

```
## [1] 50
```

Note that the correct value of 50 is returned because `var_1` equals 10. Also note the comment left after the `#` key. In R, anything that comes after `#` on a line is a comment that R ignores. Comments are ways of explaining in plain words what the code is doing, or drawing attention to important notes about the code.

Note that we can assign multiple things to a single variable. Here is a vector of numbers created using the `c` function, which combines multiple arguments into a vector or list. Below, we combine six numbers to form a vector called `vector_1`.

```
vector_1 <- c(5, 1, 3, 5, 7, 11); # Six numbers
```

We can now print and perform operations on `vector_1`.

```
vector_1;      # Prints out the vector
```

```
## [1]  5  1  3  5  7 11
```

```
vector_1 + 10; # Prints the vector elements plus ten
```

```
## [1] 15 11 13 15 17 21
```

```
vector_1 * 2;  # Prints the vector elements times two
```

```
## [1] 10  2  6 10 14 22
```

```
vector_1 <- c(vector_1, 31, 100); # Appends the vector
vector_1;
```

```
## [1]   5   1   3   5   7  11  31 100
```

We can also assign lists, matrices, or other types of objects using the `list` function.

```
object_1 <- list(vector_1, 54, "string of words");
object_1;
```

```
## [[1]]
## [1]   5   1   3   5   7  11  31 100
##
## [[2]]
## [1] 54
##
## [[3]]
## [1] "string of words"
```

Play around a bit with R before moving on, and try to get comfortable using the console. When you have finished with the R console, continue reading to learn how to store lines of code using an R script.

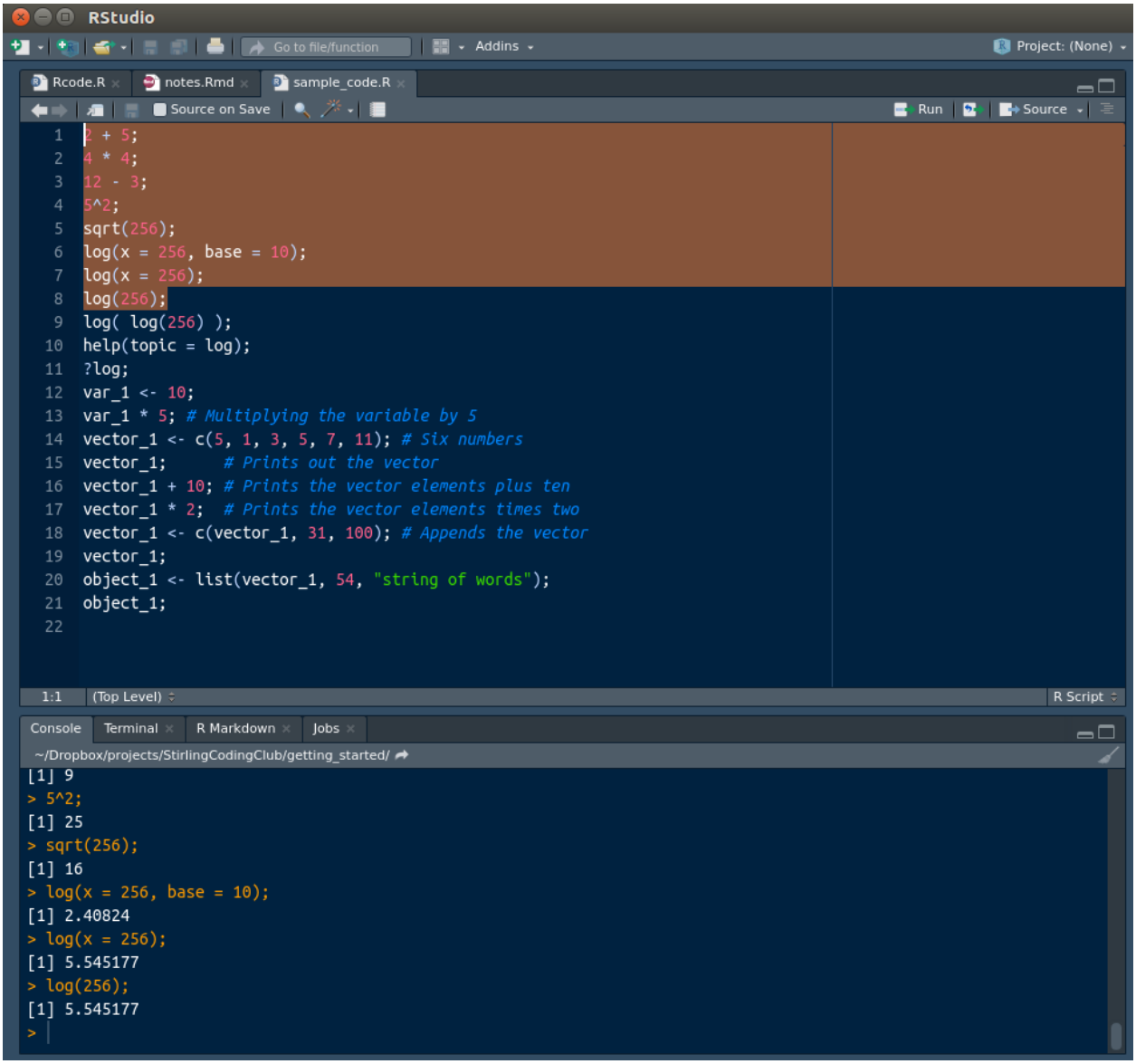
Using R script to save and run code

Up until now, we have focused on running code directly into the console. This works, but if you want to run multiple lines of code, or just save your code for later use, then you will need more than the console. R scripts are plain text files with a '.R' extension, which can be used to save R code. The R code itself is no different than what we have already run into the console. For example, we could save an R file with all of the code that we have read into the console up to this point; it would look like the below.

```
2 + 5;
4 * 4;
12 - 3;
5^2;
sqrt(256);
log(x = 256, base = 10);
```

```
log(x = 256);
log(256);
log( log(256) );
help(topic = log);
?log;
var_1 <- 10;
var_1 * 5; # Multiplying the variable by 5
vector_1 <- c(5, 1, 3, 5, 7, 11); # Six numbers
vector_1;      # Prints out the vector
vector_1 + 10; # Prints the vector elements plus ten
vector_1 * 2;  # Prints the vector elements times two
vector_1 <- c(vector_1, 31, 100); # Appends the vector
vector_1;
object_1 <- list(vector_1, 54, "string of words");
object_1;
```

You can find and download the code above in the file sample_code.R on GitHub. If you wanted to redo all of the calculations in this R script, you could open it and run each line one by one, or as a group. In the figure below, I have read sample_code.R into R by first saving it to my computer, then from the toolbar selecting 'File > Open File...' and opening it from the saved location.



There are a few things to note in the Figure above. First, the script sample_code.R now sits above the console; the position of the script might be different depending on your pane settings, but you should be able to see it appear somewhere after opening the file. Second, note how different parts of the text in the R script are coloured differently; this makes reading the text a bit easier. The variables, assignments, arguments, and functions, all appear in white. Numbers are shown in pink, strings of words are in green, and comments are in light blue (your colours might differ, but you should see some distinction among different types of text). Third, note the 'Run' button in the upper right corner above the script. This allows you to run the commands in the R script lines directly into the R console so that you do not have to retype them directly.

You can read the code from the Rscript into the console in multiple ways. The easiest is to simply click with your mouse on whatever line you want to run, then click the 'Run' button. Try clicking anywhere on line 1, for example, so that the cursor is blinking somewhere on the line. Then click 'Run'; you should see `> 2 + 5` appear in the R console, followed by the correct answer 7. After you have done this, R moves the cursor to the next line in anticipation of you wanting to run line 2. If you want to run line 2, then you could just hit 'Run' again, and repeat for line 3, 4, etc. Give this a try.

If you do not want to go through all of the code line by line, you could instead highlight a block of code, as I did above for lines 1-8. If you highlight these lines, then click 'Run', the R console will run every

line one after another, producing the output shown in the console of the Figure above. Try this as well to get a feel for running multiple lines of code at once. You now know the basics of getting started with coding in R. Next, we will move onto reading data into R for analysis, and doing a very simple correlation analysis on the classic Bumpus data set (Bumpus 1898; Johnston, Niles, and Rohwer 1972). Very briefly, the Bumpus data includes characteristics and morphological measurements from sparrows in North America following a severe storm (the specifics are not important for our purposes). You can download it here: https://bradduthie.github.io/data/Bumpus_data.csv.

Reading in data

Now we need to read in the Bumpus data. This is actually a challenging part because the data need to be in a correct format and location to be read into R successfully. The format is best read in as a CSV file, though other formats are also possible (TXT can work, or XLSX if you download and load the `openxlsx` R package and use the `read.xlsx` function). For now, I will use a CSV file with the Bumpus data set. Reading CSV files into R can be challenging at first, and I encourage you to first read in the example data set, then try reading in your own data sets into R. You can read your own dat sets into R by saving them as CSV files in Excel; as a general rule, it is good to avoid spaces in these files (replacing them, e.g., with an underscore '_'). Also make sure that all rows and columns are filled in; any empty values can be replaced with an `NA`, which R reads as unavailable data. See the Bumpus CSV file online to get an idea of what a data file looks like.

Two common errors arise at this point, which can be sources of frustration for getting started. First, the data might not be organised correctly for reading into R. Note that rows and columns should start in row '1' and column 'A' of Excel (i.e., don't leave empty rows and columns), and additional cells should not be used outside of rows and columns (if, e.g., you have a value in cell M4, when the last column in your table is K, then R will intepret this as column L being full of empty values). You should be fine if R includes some number of completely filled in rows and columns, with nothing filled in outside. If you want to, you can download the Bumpus CSV file from Dropbox and open it up in Excel for an example of what a good CSV file looks like. Note that there are no empty cells inside the table, and no values outside the table. This should therefore be read easily into R.

Second, you need to make sure that the file you are trying to read into R is located in the same place as your current working directory. You can see what your current working directory (i.e., 'folder') is using the command below.

```
getwd(); # No argument is needed here for the function
```

```
## [1] "/home/brad/Dropbox/teaching/workshops/intro_to_statistics"
```

The above function returns the current working directory. If this is the same as the CSV file that you want to read into R, then all is well. But if this is not the working directory where your CSV file is located, then you need to find it. You could do this from the R console, but the easiest way is to go to the toolbar and go to 'Session > Set Working Directory > Choose Directory...' and find the location where your CSV file is saved. The easiest way to do this is to save your data in the same place that you have saved your R script. If you do this, then you can simply go to 'Session > Set Working Directory > To Source File Location', and R will set the directory to the same file as your current R script. From there you can read in your CSV file with the `read.csv` function in R. Note that the first row of the file 'Bumpus_data.csv' is a header, which gives the column names, so we should specify the argument 'header = TRUE'.

```
dat <- read.csv(file = "Bumpus_data.csv", header = TRUE, sep = ",");
```

If you get an error message, double-check that the file name and the working directory are correct (if there is an error, this is the problem most of the time). Note that the **everything in R is case sensitive**. That means that if a letter is capitalised in the file name, but you do not capitalise it in the `file` argument above, then R will not recognise it. A lot of errors are caused by capitalisation issues in R.

Using data frames in R

Once you have succeeded in reading in a file without getting an error message, to make sure that everything looks correct, you can type `dat` in the console to see all of the data print out. I will use the 'head' function below to just print off the first six rows.

```
head(dat);
```

```
##    sex  surv totlen wingext  wgt head humer femur tibio skull stern
## 1 male alive    154     241 24.5 31.2 0.687 0.668 1.022 0.587 0.830
## 2 male alive    160     252 26.9 30.8 0.736 0.709 1.180 0.602 0.841
## 3 male alive    155     243 26.9 30.6 0.733 0.704 1.151 0.602 0.846
## 4 male alive    154     245 24.3 31.7 0.741 0.688 1.146 0.584 0.839
## 5 male alive    156     247 24.1 31.5 0.715 0.706 1.129 0.575 0.821
## 6 male alive    161     253 26.5 31.8 0.780 0.743 1.144 0.607 0.893
```

If the data appear to be read into R correctly, then you can move on to working with the data and performing analyses in R. What we are looking at above is a data frame. Visually, this is a two-dimensional table of data that includes columns of various types (in this case, words and numbers). It is completely fine to think about the data frame this way, but the way that R sees the data frame is as an ordered list of vectors, with each vector having the same number of elements. I will create a smaller one to demonstrate what I mean.

```
eg_list <- list(A = c("X", "Y", "Z"), B = c(1, 2, 3), C = c(0.3, 0.5, -0.2));
eg_list;
```

```
## $A
## [1] "X" "Y" "Z"
##
## $B
## [1] 1 2 3
##
## $C
## [1]  0.3  0.5 -0.2
```

Notice that the list that I created above includes three elements, which I have named 'A', 'B', and 'C'. Each of these elements is a vector of length three (e.g., the first element 'A' is a vector that includes "X", "Y", and "Z"). The whole list is called `eg_list`, but if I wanted to just pick out the first vector, I could do so with the `$` sign as below.

```
eg_list$A;
```

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

Notice how only the first list element (vector with "X", "Y", and "Z") is printed. Somewhat confusingly there are at least two other ways to get at this first list element. The notation for identifying list elements is to enclose them in two square brackets, so if I just wanted to pull the first element of `eg_list`, I could also have typed the below to get an identical result.

```
eg_list[[1]];
```

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

Since the first element of the list is named 'A', both `eg_list$A` and `eg_list[[1]]` give the same output. There is a third option, `eg_list[["A"]]`, which is a bit more to type, but is also more stable than the `$` because `$` allows for partial matching (e.g., if a list element was named 'December', then 'Dec' would work, but be careful if you have another list element that starts with 'Dec'!).

```
eg_list[["A"]];
```

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

If we want to get individual vector elements of our list elements, we use a single square bracket. That is, say we wanted to just pick out the second element "Y" in the list element "A". We could do so with the following code.

```
eg_list[["A"]][2];
```

```
## [1] "Y"
```

Note that `eg_list$A[2]` or `eg_list[[1]][2]` would also work just fine. Lists are very flexible in R, and you can even have lists within lists, which could make the notation quite messy (e.g., `eg_list[[1]][[1]][2]` for the second element of the first list in the first list – I don't generally recommend structuring data in this way unless you absolutely need to for some reason). In any case, it is helpful to know sometimes that when we are reading in and working with data frames, we are really just looking at lists of equal vector lengths. We can even turn our `eg_list` into a two-dimensional data frame that looks like the `dat` Bumpus data that we read in earlier.

```
as.data.frame(x = eg_list);
```

```
##   A B    C
## 1 X 1  0.3
## 2 Y 2  0.5
## 3 Z 3 -0.2
```

See in the above how each element name became the column name above. Let's take another look at the `dat` data frame again.

```
head(dat);
```

```
##    sex  surv totlen wingext  wgt head humer femur tibio skull stern
## 1 male alive    154     241 24.5 31.2 0.687 0.668 1.022 0.587 0.830
## 2 male alive    160     252 26.9 30.8 0.736 0.709 1.180 0.602 0.841
## 3 male alive    155     243 26.9 30.6 0.733 0.704 1.151 0.602 0.846
## 4 male alive    154     245 24.3 31.7 0.741 0.688 1.146 0.584 0.839
## 5 male alive    156     247 24.1 31.5 0.715 0.706 1.129 0.575 0.821
## 6 male alive    161     253 26.5 31.8 0.780 0.743 1.144 0.607 0.893
```

Note now that we can refer to each column in the same way that we referred to list elements (note, we could also put `dat` in list form with `as.list(dat)`). So if we just wanted to look at `wgt`, then we could type `dat$wgt`, `dat[["wgt"]]`, or `dat[[5]]`. Because R recognises the data frame as having two dimensions, we could also type the below to get all of the `wgt` values in the fifth column.

```
dat[,5];
```

```
##   [1] 24.5 26.9 26.9 24.3 24.1 26.5 24.6 24.2 23.6 26.2 26.2 24.8 25.4 23.7 25.7
##  [16] 25.7 26.5 26.7 23.9 24.7 28.0 27.9 25.9 25.7 26.6 23.2 25.7 26.3 24.3 26.7
##  [31] 24.9 23.8 25.6 27.0 24.7 26.5 26.1 25.6 25.9 25.5 27.6 25.8 24.9 26.0 26.5
##  [46] 26.0 27.1 25.1 26.0 25.6 25.0 24.6 25.0 26.0 28.3 24.6 27.5 31.0 28.3 24.6
##  [61] 25.5 24.8 26.3 24.4 23.3 26.7 26.4 26.9 24.3 27.0 26.8 24.9 26.1 26.6 23.3
##  [76] 24.2 26.8 23.5 26.9 28.6 24.7 27.3 25.7 29.0 25.0 27.5 26.0 25.3 22.6 25.1
##  [91] 23.2 24.4 25.1 24.6 24.0 24.2 24.9 24.1 24.0 26.0 24.9 25.5 23.4 25.9 24.2
## [106] 24.2 27.4 24.0 26.3 25.8 26.0 23.2 26.5 24.2 26.9 27.7 23.9 26.1 24.6 23.6
## [121] 26.0 25.0 24.8 22.8 24.8 24.6 30.5 24.8 23.9 24.7 26.9 22.6 26.1 24.8 26.2
## [136] 26.1
```

Note that the square brackets above identify the row and column to select in `dat`. Empty values are interpreted as 'select all', meaning that `dat[,]` is the same as writing `dat` – both select the entire data set. To select, e.g., only the first five columns of data, we could use `dat[,1:5]` (recall that `1:5` produces the sequence of integers, `1, 2, 3, 4, 5`). If instead we wanted to select the first three rows, then we could use `dat[1:3, ]`. And if we wanted only the first three rows and first five columns, we could use `dat[1:3, 1:5]`. The point is that the numbers listed within the square brackets refer to the rows and columns of the data frame, and we can use this to manipulate the data.

Here is a simple example. In this data set, the last five columns are measured in inches (all of the other length measurements are in mm). Assume that we wanted to put them into mm to match the 'totlen', 'wingext', and 'head' measurements. We just need to then multiply all of the values in the last five columns (columns 7-11) by 25.4 (1 inch equals 25.4 mm). We could do this column by column. For example, to multiply all of the values in the seventh column `humer`, we could use the following code.

```
dat[["humer"]] <- dat[["humer"]] * 25.4;
```

Verbally, what I have done above is to assign `dat[["humer"]]` to a new value of `dat[["humer"]] * 25.4` – that is, I have multiplied the column `dat[["humer"]]` by 25.4 and inserted the result back into the `dat` array. A short-cut for doing the rest of them all at once (columns 8-11) is below.

```
dat[,8:11] <- dat[,8:11] * 25.4;
```

I am mixing and matching the notation a bit just to get you used to seeing a couple different versions (as a side note, R allows us to assign in both directions, so we could have also typed `dat[,8:11] * 25.4 -> dat[,8:11];`, though it is very rare to do it this way). Now we should have a data set with the last five columns in mm rather than inches.

```
head(dat);
```

```
##    sex  surv totlen wingext  wgt head   humer   femur   tibio   skull   stern
## 1 male alive    154     241 24.5 31.2 17.4498 16.9672 25.9588 14.9098 21.0820
## 2 male alive    160     252 26.9 30.8 18.6944 18.0086 29.9720 15.2908 21.3614
## 3 male alive    155     243 26.9 30.6 18.6182 17.8816 29.2354 15.2908 21.4884
## 4 male alive    154     245 24.3 31.7 18.8214 17.4752 29.1084 14.8336 21.3106
## 5 male alive    156     247 24.1 31.5 18.1610 17.9324 28.6766 14.6050 20.8534
## 6 male alive    161     253 26.5 31.8 19.8120 18.8722 29.0576 15.4178 22.6822
```

Maybe we want to save this data set as a CSV file. To do so, we can use the `write.csv` function as below.

```
write.csv(x = dat, file = "Bumpus_data_mm.csv", row.names = FALSE);
```

There will now be a new file 'Bumpus_data_mm.csv' in the working directory with the changes that we made to it (converting inches to mm). Note my use of the argument `row.names = FALSE`. This is just because the `write.csv` function, somewhat annoyingly, will otherwise assume that we want to insert a first column of row names, which will show up as integer values (i.e., a new first column with the sequence, 1, 2, 3, . . . , 136).

Some useful base functions in R

Now that you have a handle on how to refer to rows, columns, and individual values of a data set, I will introduce some functions in R that might be useful for managing data. If at any time we want to look at what objects we have available in Rstudio, then we can use the `ls` function below.

```
ls();
```

```
## [1] "dat"      "eg_list"  "object_1" "var_1"    "vector_1"
```

The above output should make sense because we have assigned `dat` and `eg_list`. Now say we want to get rid of the `eg_list` that I assigned. We can remove it using the `rm` function.

```
rm(eg_list);
```

The `eg_list` should now be removed from R and not appear anymore if I type `ls()`.

```
ls();
```

```
## [1] "dat"      "object_1" "var_1"    "vector_1"
```

Now let's look more at `dat`. Let's say that I want to find out about the attributes of this object. I can use the `attributes` function to learn more.

```
attributes(dat);
```

```
## $names
##  [1] "sex"     "surv"    "totlen"  "wingext" "wgt"     "head"    "humer"
##  [8] "femur"   "tibio"   "skull"   "stern"
##
## $row.names
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
## [109] 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
## [127] 127 128 129 130 131 132 133 134 135 136
##
## $class
## [1] "data.frame"
```

You can now see the names (column names, which recall are also vector element names), row names, and the class of the object. Note that not all objects will have the same (or indeed any) attributes. But the attributes that we can see gives us a bit of information, and we can actually refer to the attributes themselves with the equivalently named `names`, `row.names`, and `class` functions. For example, if we wanted to get all of the names of dat, we could use the code below.

```
names(dat);
```

```
##  [1] "sex"     "surv"    "totlen"  "wingext" "wgt"     "head"    "humer"
##  [8] "femur"   "tibio"   "skull"   "stern"
```

The same would work for `row.names` and `class`. We can also pull out information for individual columns of data. If, for example, we wanted a quick count of the alive versus dead individual sparrows in the data set, we could use the `table` function.

```
table(dat$surv);
```

```
##
## alive  dead
##    72    64
```

If we wanted the set of unique `totlen` values, then we could use the `unique` function.

```
unique(dat$totlen);
```

```
##  [1] 154 160 155 156 161 157 159 158 162 166 163 165 153 164 167 152
```

Let's say that we wanted to know if a sparrow has total length greater than 160. We could use the code below to get a TRUE/FALSE vector for which 'TRUE' indicates a length over 160 and a 'FALSE' indicates a length of 160 or less.

```
dat$totlen > 160;
```

```
##   [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
##  [13]  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE
##  [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
##  [37] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE
##  [49]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE
##  [61] FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
##  [73] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
##  [85]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE
##  [97] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
## [109] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
## [121] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE
## [133]  TRUE FALSE  TRUE  TRUE
```

The above looks a bit messy, but it can be quite useful if we also know that R interprets `TRUE` as having a value of 1, and `FALSE` as having a value of `0`. We can actually get R to confirm this itself, e.g., by checking if `FALSE == 1`.

```
FALSE == 1;
```

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

It does not. But now we can check if `FALSE == 0`.

```
FALSE == 0;
```

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

We could do the same for `TRUE` and confirm that `TRUE == 1`. Since `TRUE` is equivalent to 1 and `FALSE` is equivalent to 0, a numeric version of the `TRUE/FALSE` vector above would look like the below.

```
as.numeric(dat$totlen > 160);
```

```
##   [1] 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 1 1 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0
##  [38] 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 0 1 0 1 1 1 0 0 1 1 0 0 1 1 1 1 0 0 0 0 0
##  [75] 0 0 0 0 1 1 0 1 1 1 1 1 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0
## [112] 0 0 0 0 1 0 1 1 0 0 0 1 0 0 0 1 1 0 0 1 0 1 0 1 1
```

Compare the above to the equivalent `TRUE/FALSE` vector to confirm that they are the same. We can use this to our advantage to count all of the sparrows whose length exceeds 160.

```
sum(dat$totlen > 160);
```

```
## [1] 55
```

Note that what we have done is produce that same `TRUE/FALSE` vector as above, but then summed up all the values. Because `TRUE` values are 1 and `FALSE` values are 0, the number of `TRUE` values get counted up. We can do multiple comparisons though too. Say that we wanted to find out the number of individuals that have a total length either greater than 160 mm or less than or equal to 155 mm. We could make this work using the 'or' operator, represented in R by a vertical line `|`.

```
sum(dat$totlen > 160 | dat$totlen <= 155);
```

```
## [1] 76
```

Notice how the `|` separates the two comparisons, and the `<=` sign is used to indicate 'less than or equals to' (a `>=` would indicate 'greater than or equal to, and `==` is simply 'equal to', as we saw above).

We could also try to pull out a subset of individuals who have a total length greater than 160 mm and are female. We could make this work using the 'and' operator, represented by an ampersand in R, `&`.

```
sum(dat$totlen > 160 & dat$sex == "female");
```

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

Notice how the `&` separates the two comparisons and "female" is placed in quotes (else R will look for an object called 'female' and come up empty with an error message – try this out!).

Now say we wanted to identify the row numbers of the individuals with a total length above 165 mm. We could first find these individuals and assign their rows to a new variable using the `which` function.

```
inds_gt_165 <- which(dat$totlen > 165);
inds_gt_165;
```

```
## [1] 18 57 59 79 83 84 86
```

Now that we have these rows of individuals with a total length above 165 mm, we could use these values in `inds_gt165` to view just these rows of the `dat` data frame.

```
dat[inds_gt_165,];
```

```
##     sex  surv totlen wingext  wgt head   humer   femur   tibio   skull   stern
## 18 male alive    166     253 26.7 32.5 19.4818 19.4310 31.2420 15.2400 22.3012
## 57 male  dead    166     251 27.5 31.5 18.2880 17.5514 28.3972 15.5448 21.5138
## 59 male  dead    166     250 28.3 32.4 19.1516 18.2372 29.9466 15.4178 23.2664
## 79 male  dead    166     245 26.9 31.7 18.1610 17.6530 28.1178 15.2654 21.5138
## 83 male  dead    166     256 25.7 31.7 19.1008 19.0754 30.1498 15.1130 21.7932
## 84 male  dead    167     255 29.0 32.2 19.4310 18.9230 30.4038 16.2052 21.7170
## 86 male  dead    166     254 27.5 31.4 19.3040 18.8468 28.5496 15.3416 23.2156
```

Notice how the values of `totlen` are all above 165, and the row numbers to the left match the values in `inds_gt_165`. One more quick trick – say that we wanted to check if a living individual was in this subset (obviously we can see that the first one is alive, but pretend for a moment that the data frame was much larger). We could use the `%in%` operator to check.

```
"alive" %in% dat$surv;
```

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

Again, note how 'alive' is placed in quotes. We could also check to see if numeric values are in the data set. For example, we could ask if the value '250' appears anywhere in `dat$wingext`.

```
250 %in% dat$wingext;
```

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

More advanced base R functions

Next, I want to demonstrate three useful functions in R, `tapply`, `apply`, and `lapply`. All of these functions essentially apply some other function across an array, table, or list. Say that we want to find the means of females and males in the data set. We could do this with the `tapply` function.

```
tapply(X = dat$totlen, INDEX = dat$sex, FUN = mean);
```

```
##   female     male
## 157.9796 160.4253
```

Note that in the above, the argument `X` is what we want to do the calculations across (total length), `INDEX` does the calculation for each unique element in the vector (i.e., 'female' and 'male' in `dat$sex`), and `FUN` indicates the function ('mean' in this case, but we could do 'sd', 'length', 'sum', or any other calculation that we want). We could use all of this to calculate, e.g., the standard error of females and males.

```
N_dat  <- tapply(X = dat$totlen, INDEX = dat$sex, FUN = length);
SD_dat <- tapply(X = dat$totlen, INDEX = dat$sex, FUN = sd);
SE_dat <- SD_dat / sqrt(N_dat);
SE_dat;
```

```
##    female      male
## 0.5220396 0.3435868
```

The function `lapply` works similarly, but over lists. We can remake that example list from earlier.

```
eg_list <- list(A = c("X", "Y", "Z"), B = c(1, 2, 3), C = c(0.3, 0.5, -0.2));
eg_list;
```

```
## $A
## [1] "X" "Y" "Z"
##
## $B
## [1] 1 2 3
##
## $C
## [1]  0.3  0.5 -0.2
```

If we wanted to confirm the length of each of the three list elements, we could do so with `lapply`.

```
lapply(X = eg_list, FUN = length);
```

```
## $A
## [1] 3
##
## $B
## [1] 3
##
## $C
## [1] 3
```

We can try to use a function like 'mean' too, but note that we cannot get the mean of "X", "Y", and "Z", as this does not make any sense. If we try to do it, then R will give us an answer of `NA` for the first element with a warning, then calculate the means of the numeric elements.

```
lapply(X = eg_list, FUN = mean);
```

```
## Warning in mean.default(X[[i]], ...): argument is not numeric or logical:
## returning NA
```

```
## $A
## [1] NA
##
## $B
## [1] 2
##
## $C
## [1] 0.2
```

Finally, the function `apply` works similarly on arrays of numbers. That is, we need to have an object with two (or more) dimensions in which every element is a number. To get an example, we can just use the last nine columns of `dat` (i.e., everything except `sex` and `surv`). We can define this below in a new object.

```
dat_array <- dat[,3:11];
head(dat_array);
```

```
##   totlen wingext  wgt head   humer   femur   tibio   skull   stern
## 1    154     241 24.5 31.2 17.4498 16.9672 25.9588 14.9098 21.0820
## 2    160     252 26.9 30.8 18.6944 18.0086 29.9720 15.2908 21.3614
## 3    155     243 26.9 30.6 18.6182 17.8816 29.2354 15.2908 21.4884
## 4    154     245 24.3 31.7 18.8214 17.4752 29.1084 14.8336 21.3106
## 5    156     247 24.1 31.5 18.1610 17.9324 28.6766 14.6050 20.8534
## 6    161     253 26.5 31.8 19.8120 18.8722 29.0576 15.4178 22.6822
```

Now say that we wanted to get the mean value of every column. We could go through individually and find `mean(dat_array$totlen)`, `mean(dat_array$wingext)`, etc., or we could use `apply`.

```
apply(X = dat_array, MARGIN = 2, FUN = mean);
```

```
##    totlen   wingext       wgt      head     humer     femur     tibio     skull
## 159.54412 245.25735  25.52500  31.57279  18.59691  18.10852  28.79258  15.30126
##     stern
##  21.33432
```

The `MARGIN` argument is the only different one from `lapply` and `tapply`, and it is a bit confusing at first. It states the dimension over which the function will be applied, with 1 representing rows and 2 representing columns. This makes more sense when you consider that numeric arrays (unlike data frames) can have any number of dimensions. For example, consider this three dimensional array.

```
eg_array <- array(data = 1:64, dim = c(4, 4, 4));
eg_array;
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   17   21   25   29
## [2,]   18   22   26   30
## [3,]   19   23   27   31
## [4,]   20   24   28   32
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
```

```
## [1,]   33   37   41   45
## [2,]   34   38   42   46
## [3,]   35   39   43   47
## [4,]   36   40   44   48
##
## , , 4
##
##      [,1] [,2] [,3] [,4]
## [1,]   49   53   57   61
## [2,]   50   54   58   62
## [3,]   51   55   59   63
## [4,]   52   56   60   64
```

See how the array has four rows, four columns, and four different layers. Now if we wanted to pull out, e.g., the first row and second column of the fourth layer, we could do so with square brackets as below.

```
eg_array[1, 2, 4];
```

```
## [1] 53
```

We could also use `apply` across the third dimension (which I've been calling 'layer') to get the mean of each.

```
apply(X = eg_array, MARGIN = 3, FUN = mean);
```

```
## [1]  8.5 24.5 40.5 56.5
```

We could even get the mean column value *for each layer* with the somewhat confusing notation below.

```
apply(X = eg_array, MARGIN = c(1, 3), FUN = mean);
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    7   23   39   55
## [2,]    8   24   40   56
## [3,]    9   25   41   57
## [4,]   10   26   42   58
```

You will probably never need to actually do this, but it is possible. To read the above, note that the each row represents a layer in the original `eg_array`, and each column represents a row of that layer. So, e.g., the mean of 1, 5, 9, and 13 is 7; the mean of 2, 6, 10, and 14 is 8, and so forth. Moving onto the next layer, the mean of 17, 21, 25, and 29 is 23. Again, you will almost never need to do this, but it can be useful if you are working with multidimensional arrays.

Indexing values in R

Note that the Bumpus data (`dat`) is a big table that is now read into R. While we do not necessarily see the entire table at once, as we would in Excel, we can pull out any of the information in that we want. For example, if we want to see how many rows and columns are in `dat`, we can use the following functions.

```
nrow(dat);
```

```
## [1] 136
```

```
ncol(dat);
```

```
## [1] 11
```

We could also just use the function `dim` to get the dimensions of `dat` (note that this would work for an array of any number of dimensions).

```
dim(dat);
```

```
## [1] 136  11
```

So we know that our table `dat`, which contains the Bumpus data, includes 136 rows and 11 columns. Having read this table into R successfully, we can now perform any number of statistical analyses on the contents. The different ways to analyse theses data are beyond the scope of these notes, but there are a few useful things to know. First, the row and columns in `dat` can be indexed using square brackets. If, for example, we wanted to just look at the value of the fourth row and sixth column, we could type the following.

```
dat[4, 6]; # First row, second column
```

```
## [1] 31.7
```

The first position within the brackets is the row (`4`), and the second position is the column (`6`). Note that R is not restricted to two dimensions; it is possible to have three or more dimensions of an array, in which case we might refer to an array element as `dat[x_dim, y_dim, z_dim]` for a `dat` of three dimensions. Note that we can also store any particular value in `dat` as a variable, if we want. We could, for example store the above as `dat_point_1` using the code below.

```
dat_point_1 <- dat[4, 6];
```

We could then use `dat_point_1` in place of `dat[4, 6]`. We can also define entire rows or columns. For example, if we wanted to return all of the values of row 4, then we could leaven the second index blank, as below.

```
dat[4, ]; # Note the empty space where a column was previously
```

```
##    sex  surv totlen wingext  wgt head   humer   femur   tibio   skull   stern
## 4 male alive    154     245 24.3 31.7 18.8214 17.4752 29.1084 14.8336 21.3106
```

In the Bumpus data set, this gives us all the information of measurements for sparrow number 4. We can do the same for columns. Note that column 5 holds the mass of each sparrow (in grams). We could look at all of the sparrow masses using the code below.

```
dat[, 5]; # Note the empty space is now where a row used to be.
```

```
##   [1] 24.5 26.9 26.9 24.3 24.1 26.5 24.6 24.2 23.6 26.2 26.2 24.8 25.4 23.7 25.7
##  [16] 25.7 26.5 26.7 23.9 24.7 28.0 27.9 25.9 25.7 26.6 23.2 25.7 26.3 24.3 26.7
##  [31] 24.9 23.8 25.6 27.0 24.7 26.5 26.1 25.6 25.9 25.5 27.6 25.8 24.9 26.0 26.5
##  [46] 26.0 27.1 25.1 26.0 25.6 25.0 24.6 25.0 26.0 28.3 24.6 27.5 31.0 28.3 24.6
##  [61] 25.5 24.8 26.3 24.4 23.3 26.7 26.4 26.9 24.3 27.0 26.8 24.9 26.1 26.6 23.3
##  [76] 24.2 26.8 23.5 26.9 28.6 24.7 27.3 25.7 29.0 25.0 27.5 26.0 25.3 22.6 25.1
##  [91] 23.2 24.4 25.1 24.6 24.0 24.2 24.9 24.1 24.0 26.0 24.9 25.5 23.4 25.9 24.2
## [106] 24.2 27.4 24.0 26.3 25.8 26.0 23.2 26.5 24.2 26.9 27.7 23.9 26.1 24.6 23.6
## [121] 26.0 25.0 24.8 22.8 24.8 24.6 30.5 24.8 23.9 24.7 26.9 22.6 26.1 24.8 26.2
## [136] 26.1
```

Note that this now returns the masses of all 136 sparrows. Since our table has headers, and the header for column 5 is wgt, we could also use the code below.

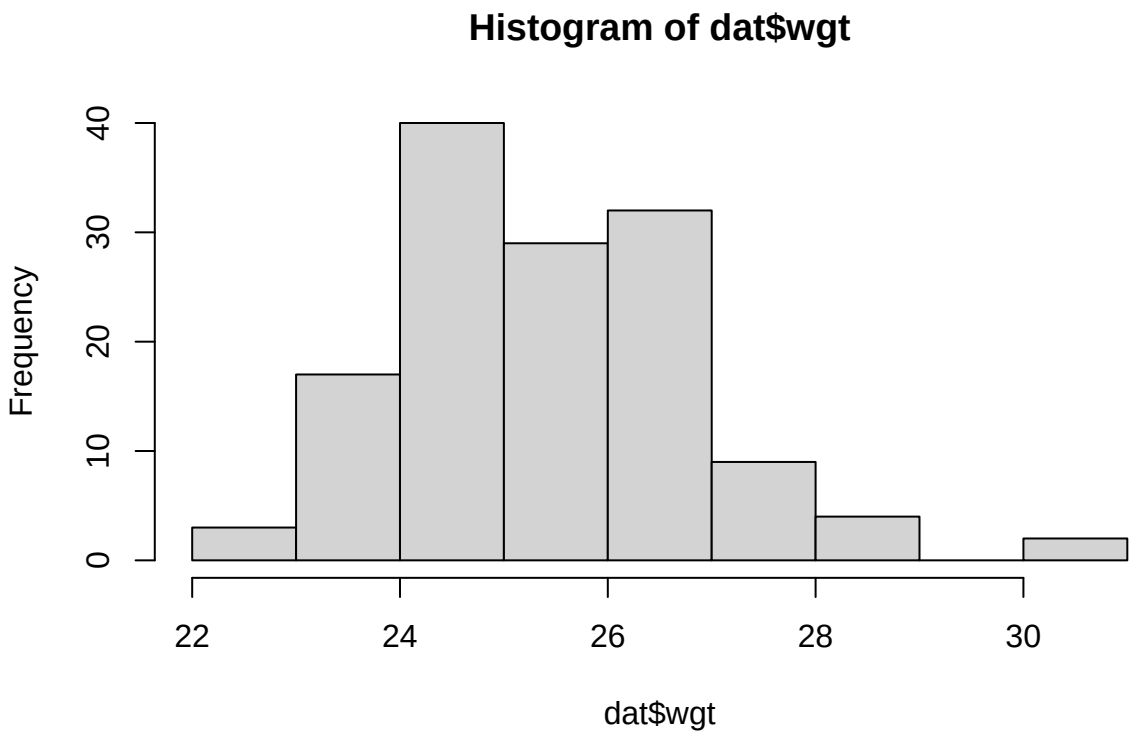
```
dat$wgt; # R sees the column header and returns column 5; same as above
```

```
##   [1] 24.5 26.9 26.9 24.3 24.1 26.5 24.6 24.2 23.6 26.2 26.2 24.8 25.4 23.7 25.7
##  [16] 25.7 26.5 26.7 23.9 24.7 28.0 27.9 25.9 25.7 26.6 23.2 25.7 26.3 24.3 26.7
##  [31] 24.9 23.8 25.6 27.0 24.7 26.5 26.1 25.6 25.9 25.5 27.6 25.8 24.9 26.0 26.5
##  [46] 26.0 27.1 25.1 26.0 25.6 25.0 24.6 25.0 26.0 28.3 24.6 27.5 31.0 28.3 24.6
##  [61] 25.5 24.8 26.3 24.4 23.3 26.7 26.4 26.9 24.3 27.0 26.8 24.9 26.1 26.6 23.3
##  [76] 24.2 26.8 23.5 26.9 28.6 24.7 27.3 25.7 29.0 25.0 27.5 26.0 25.3 22.6 25.1
##  [91] 23.2 24.4 25.1 24.6 24.0 24.2 24.9 24.1 24.0 26.0 24.9 25.5 23.4 25.9 24.2
## [106] 24.2 27.4 24.0 26.3 25.8 26.0 23.2 26.5 24.2 26.9 27.7 23.9 26.1 24.6 23.6
## [121] 26.0 25.0 24.8 22.8 24.8 24.6 30.5 24.8 23.9 24.7 26.9 22.6 26.1 24.8 26.2
## [136] 26.1
```

Basic plotting in R

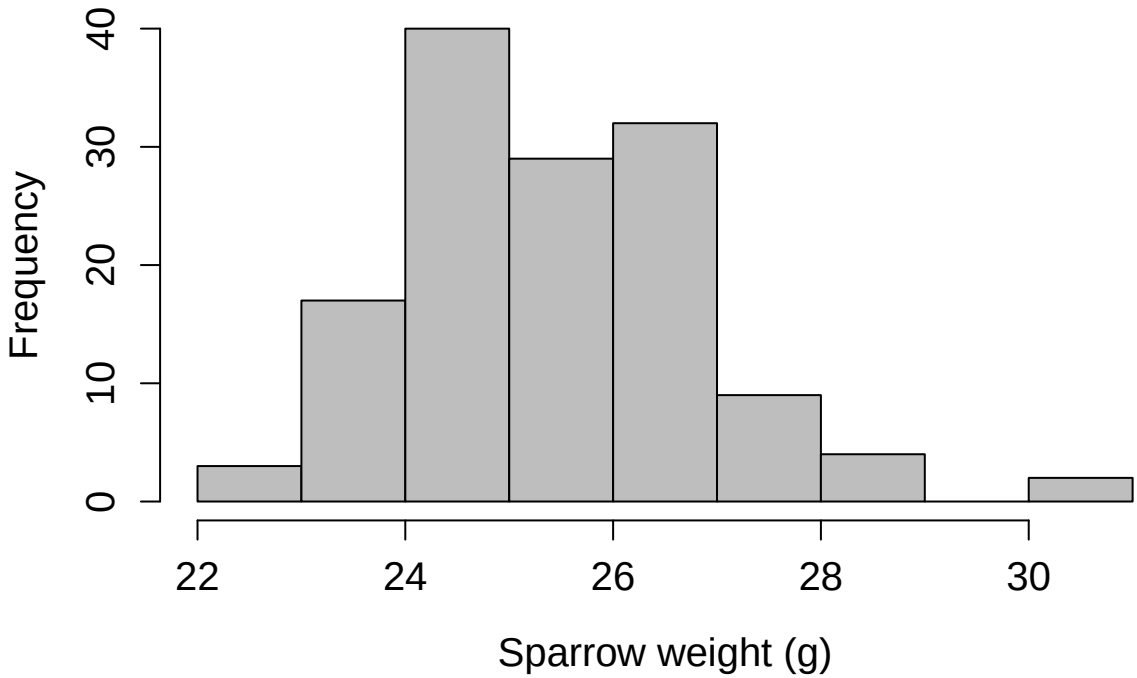
We can plot a histogram of sparrow weights using the built-in function `hist` in R.

```
hist(x = dat$wgt);
```



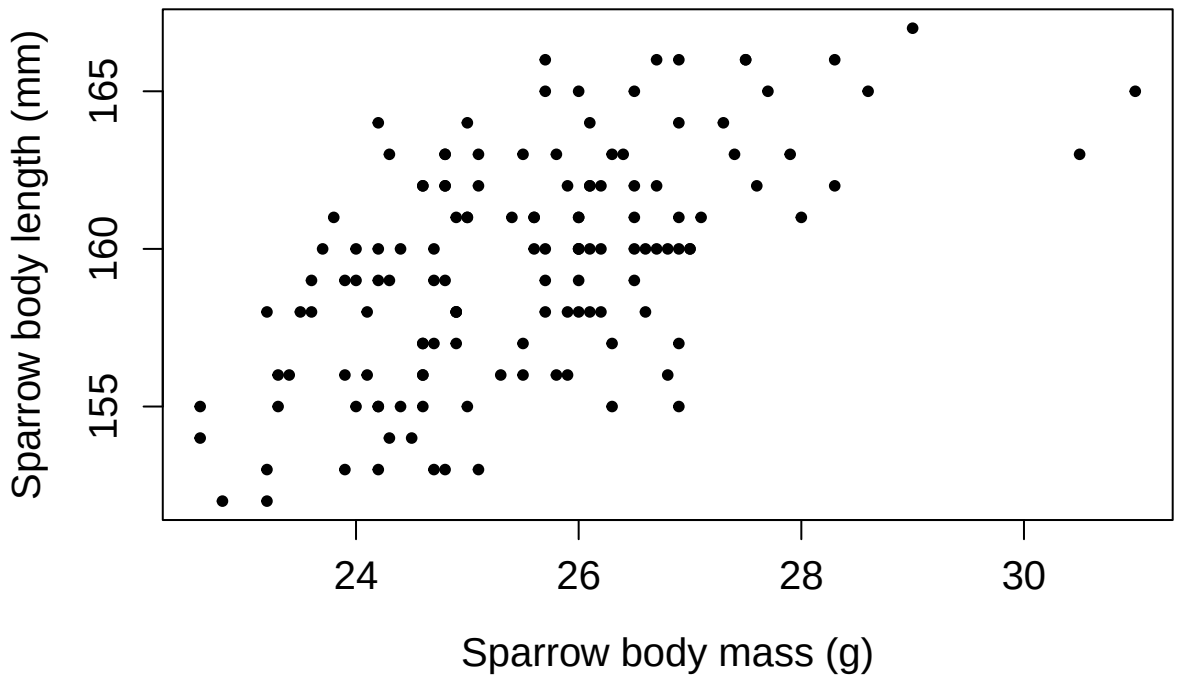
This looks a bit rubbish; the main title is unnecessary, and the axis labels are not terribly informative. We can tweak the axis labels and colours using the following arguments.

```
hist(x = dat$wgt, main = "", xlab = "Sparrow weight (g)", ylab = "Frequency",
     cex.lab = 1.25, cex.axis = 1.25, col = "grey");
```



We can also make a scatterplot using the `plot` function in R. The scatterplot below shows body mass versus length.

```
plot(x = dat$wgt, y = dat$totlen, xlab = "Sparrow body mass (g)",
     ylab = "Sparrow body length (mm)", cex.lab = 1.25, cex.axis = 1.25,
     pch = 20); # Note: cex.lab, cex.axis, and pch are purely cosmetic
```



You can try various arguments to `plot` to change the axes (`xlim` and `ylim`), the points (`pch`), the colour (`col`), or several other options.

Installing packages in R

One of the most useful attributes of R is the ability to make and use packages. Packages allow custom code, data, and documentation to be published and used by anyone. Up until now, we have been looking at functions (e.g., `hist` or `plot`) that are available in base R (i.e., they are part of the R language, and anyone who installs R can use these functions). But by downloading and installing a package, it is possible to use functions written by anyone in the R community. As an example, we can download the swirl package (https://swirlstats.com/). The swirl package is a useful package for learning R step-by-step. You can get started by running the code below.

```
install.packages("swirl");
library("swirl");
swirl();
```

```
##
## | Hi! I see that you have some variables saved in your workspace. To keep
## | things running smoothly, I recommend you clean up before starting swirl.
##
## | Type ls() to see a list of the variables in your workspace. Then, type
## | rm(list=ls()) to clear your workspace.
##
## | Type swirl() when you are ready to begin.
```

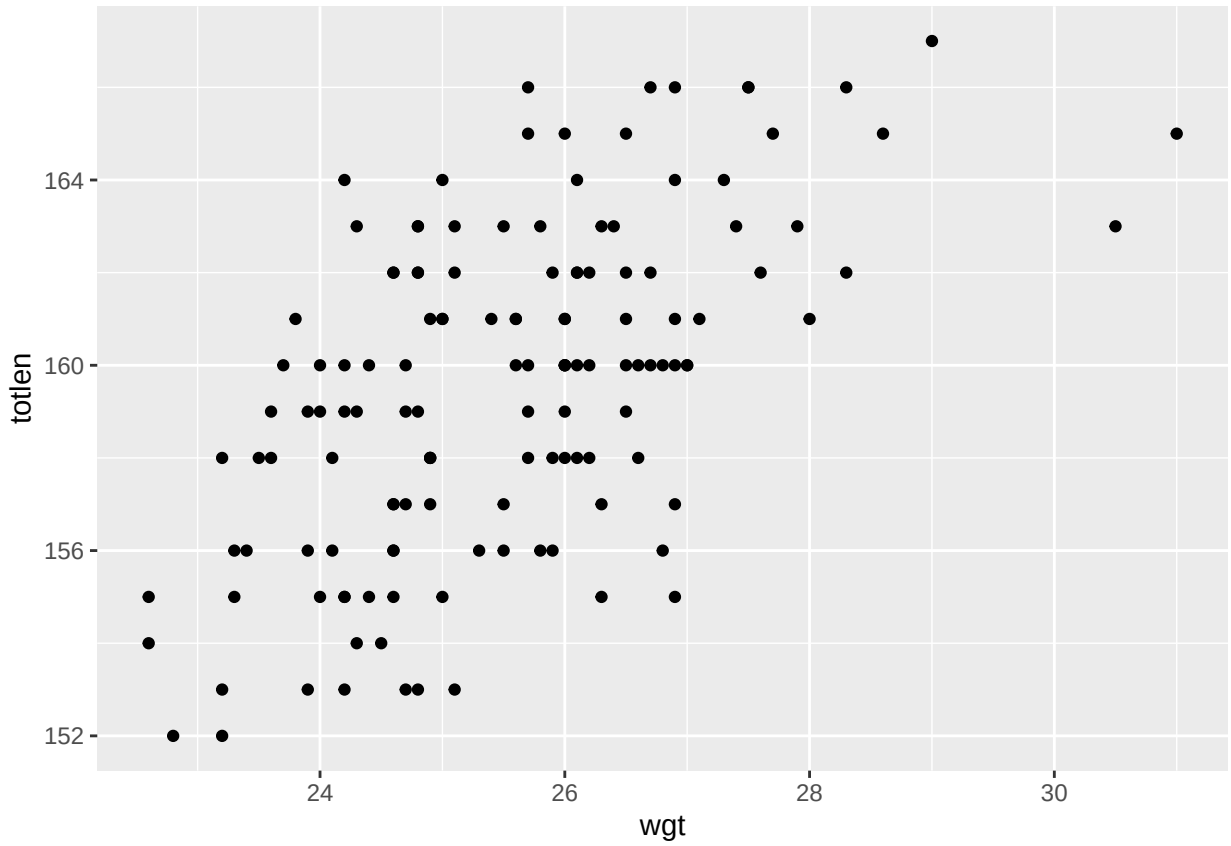
The `install.packages` command is all that you need to install a package directly from the Comprehensive R Archive Network (CRAN; https://cran.r-project.org/). After R installs the package, the `library` function is used to deploy the package functions into the console. There are 18000+ packages in R, all free to download and use. If you use one of these packages, you can cite it with the `citation()` function in base R.

```
citation("swirl");
```

```
##
## To cite package 'swirl' in publications use:
##
##   Kross S, Carchedi N, Bauer B, Grdina G (2020). _swirl: Learn R, in
##   R_. R package version 2.4.5,
##   <https://CRAN.R-project.org/package=swirl>.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {swirl: Learn R, in R},
##     author = {Sean Kross and Nick Carchedi and Bill Bauer and Gina Grdina},
##     year = {2020},
##     note = {R package version 2.4.5},
##     url = {https://CRAN.R-project.org/package=swirl},
##   }
```

As one more example, a popular package is `ggplot2`, which some people prefer to use in plotting instead of the available base R functions. If we wanted to plot sparrow weight versus total length using a slightly different plotting style from `plot` (see above), then we could download `ggplot2` and run the `ggplot` function.

```
install.packages("ggplot2");
library("ggplot2");
ggplot(data = dat, mapping = aes(x = wgt, y = totlen)) +
    geom_point();
```



There are ways to make the plot above look much nicer, but I am not an expert in `ggplot`. For your own data, I recommend exploring and trying new tools and new code. Most of what you try will not work out they way you want it to, but you will almost always learn something new and get practice.

Custom functions in R

Functions can be written in R, then used in ways that make data wrangling, analysis, and all kinds of other tasks much more easy and efficient. Being able to write your own functions is a highly useful skill. For example, if you collect data in a particular way more than once, and need to reorganise those data to a different format than you collected for statistical analysis, then it might be useful to write a function that automates this so that you can simply run the function on your collected dataset. To write a function, you use the `function` function, which specifies the function name, the arguments that your custom function will take (along with any defaults), what the function does, and what the function returns. Below is a very simple example that converts a Fahrenheit value to Celsius.

```
F_to_C <- function(F_temp = 70){
    C_temp <- (F_temp - 32) * 5/9;
    return(C_temp);
}
```

The function is assigned to the name `F_to_C`. The function takes one argument `F_temp`, which has a default value of 70. It then calculates `(F_temp - 32) * 5/9` and assigns it to the variable `C_temp`. It then returns `C_temp`. To input this function into R, we just need to highlight the whole function and run it. After it is run, then we can then use `F_to_C` just as we would any other function. This is demonstrate below with a Fahrenheit temperature of 100.

```
F_to_C(F_temp = 100);
```

```
## [1] 37.77778
```

Note that we do not need to actually write out `F_temp` as an argument. R recognises the order of arguments, or in this case, that there is only one argument in the `F_To_C` function, so it must be `F_temp`. We can therefore do the same thing as above without specifying the argument.

```
F_to_C(100);
```

```
## [1] 37.77778
```

Also note that because we gave the argument `F_temp` a default value of 70, if we do not specify any argument, then it runs the function with this default `F_temp = 70`.

```
F_to_C();
```

```
## [1] 21.11111
```

If we realy want to, we can also set the function without `return`. In this case, R simply returns the last variable assigned.

```
F_to_C <- function(F_temp = 70){
    C_temp <- (F_temp - 32) * 5/9;
}
```

The above leaves out the return argument, but it stil returns `C_temp` because this is the last variable assigned.

```
F_to_C(100);
```

We can also create custom functions that use custom functions. For example, assume that we wanted to convert from temperature Fahrenheit to temperature Kelvin. Temperature Kelvin is simply Celsius plus 273.15, $K = C + 273.15$. We could convert from Fahrenheit to Kelvin by calculating `((F_temp - 32) * 5/9) + 273.15`, or we could make use of the `F_to_C` function within a new function `F_to_K`.

```
F_to_K <- function(F_temp){
  K_temp <- F_to_C(F_temp = F_temp) + 273.15;
  return(K_temp);
}
```

What is going on above? We are defining the new function `F_to_K`, which takes an argument `F_temp` and converts it to Kelvin. It does this by passing this `F_temp` into the previously written function `F_to_C`. It calculates the output of `F_to_C(F_temp)` and then adds 273.15 and assigns the value to `K_temp`. Then it returns the calculated `K_temp`.

```
F_to_K(F_temp = 100);
```

```
## [1] 310.9278
```

Note that I have not given `F_temp` a default value in the `F_to_K` function above, so we cannot just call it without defining a value. If we do, then we get an error message.

```
F_to_K();
```

```
## Error in F_to_C(F_temp = F_temp): argument "F_temp" is missing, with no default
```

We can try something else. Maybe we want to write a function that converts degrees Fahrenheit to *either* Celsius or Kelvin. We can do this with a new function that includes two arguments, one specifying the degrees Fahrenheit, and the second specifying whether the function should convert to Celsius or Fahrenheit.

```
F_convert <- function(F_temp = 70, conversion = "Celsius"){
  if(conversion == "Celsius"){
    converted <- F_to_C(F_temp = F_temp);
  }
  if(conversion == "Kelvin"){
    converted <- F_to_K(F_temp = F_temp);
  }
  return(converted);
}
```

The below will get the job done, provided we use it correctly.

```
F_convert(F_temp = 70, conversion = "Kelvin");
```

```
## [1] 294.2611
```

But the function could be written better. If, for example, we spell "Kelvin" incorrectly, or even forget to capitalise it, then we will get an uniformative error.

```
F_convert(F_temp = 70, conversion = "kelvin");
```

```
## Error in F_convert(F_temp = 70, conversion = "kelvin"): object 'converted' not found
```

To improve the function, we can redefine it to make sure it gives an informative error message. We can check to make sure that `conversion` is either "Celsius" or "Kelvin", and that "F_temp" is a numeric input. If this is not the case, then the function should `STOP` with an error message.

```
F_convert <- function(F_temp = 70, conversion = "Celsius"){
  if(conversion != "Celsius" & conversion != "Kelvin"){
    stop("conversion argument must be 'Celsius' or 'Kelvin'.")
  }
  if(is.numeric(F_temp) == FALSE){
    stop("F_temp argument must be numeric");
  }
  if(conversion == "Celsius"){
    converted <- F_to_C(F_temp = F_temp);
  }else{
    converted <- F_to_K(F_temp = F_temp);
  }
  return(converted);
}
```

Note that the first `if` statement in the function above checks to see if the argument `conversion` does *not* equal "Celsius" (`conversion != "Celsius"`) **and** (`&`) does *not* equal "Kelvin" (`conversion != "Kelvin"`). If this is `TRUE`, then `conversion` must be incorrectly specified because it does not equal one of the two permissible options. We therefore need to return an error message. Note that since we have established that `conversion` must be either "Celsius" or "Kelvin" (returning an error message if not), then we can just use the `else` at the end of the `if(conversion == "Celsius")` statement. We can try the incorrect specification of `conversion = "kelvin"` again.

```
F_convert(F_temp = 70, conversion = "kelvin");
```

```
## Error in F_convert(F_temp = 70, conversion = "kelvin"): conversion argument must be 'Celsius' or 'Kel
```

Similarly, if we specify a value of `F_temp` that is not numeric, then `is.numeric(F_temp)` will be `FALSE`, causing the function to `stop` with a relevant error message.

```
F_convert(F_temp = "seventy", conversion = "Kelvin");
```

```
## Error in F_convert(F_temp = "seventy", conversion = "Kelvin"): F_temp argument must be numeric
```

We can run the function correctly now.

```
F_convert(F_temp = 70, conversion = "Kelvin");
```

```
## [1] 294.2611
```

Note that we could also assign the output of the above to a new variable.

```
new_variable <- F_convert(F_temp = 70, conversion = "Kelvin");
print(new_variable);
```

```
## [1] 294.2611
```

It is often useful to write error messages like this into functions even if you know that you will be the only person who uses them. Next, try to write a function that converts from Celsius to either Fahrenheit or Kelvin. The 'Details' pulldown below gives one potential way to do it.

```
C_convert <- function(C_temp, conversion = "Kelvin"){
  if(conversion != "Fahrenheit" & conversion != "Kelvin"){
    stop("conversion argument must be 'Fahrenheit' or 'Kelvin'.")
  }
  if(is.numeric(C_temp) == FALSE){
    stop("C_temp argument must be numeric");
  }
  if(conversion == "Celsius"){
    converted <- C_temp + 273.15;
  }else{
    converted <- (C_temp * 9/5) + 32;
  }
  return(converted);
}
```

Try creating some other functions that might be useful for your own purposes, or that do something fun.

There is one more trick that we can do with functions called *recursion*. Recursion occurs when a function calls *itself* until some kind of stopping condition is met. This is almost never necessary to use once you know how to use a loop (which we will learn in the next session). Since learning to code, there has been exactly one time I can recall in which I basically **needed** to use recursive programming, and this was programming in C, not R. Nevertheless, it is a bit of a cool trick to know.

Suppose we wanted to create a function that samples from a range of values `interval` iteratively until it finds the value that makes it stop (`stop_number`). Further suppose that we want to record all of the values tried until sampling the `stop_number`. There are many ways that we could do this in R, but the program below shows how to do it with recursive programming.

```
recursive_function <- function(stop_number, interval = 1:10, rand_tries = NULL){
  rand_number <- sample(x = interval, size = 1);
  rand_tries  <- c(rand_tries, rand_number);
  if(rand_number == stop_number){
    return(rand_tries);
  }else{
    new_try    <- recursive_function(stop_number = stop_number,
                                     interval    = interval,
                                     rand_tries  = rand_tries);
  }
}
```

Verbally, the program first randomly samples a number `rand_number` (by default the random number is from 1 to 10). The number is added to a growing list called `rand_tries`, which starts out as `NULL` with the first call to the function. If the `rand_number` equals the `stop_number`, then the function returns the list `rand_tries`. But if it does not equal `stop_number`, then the function runs again, so a new `rand_number` is selected and added to `rand_tries`. This continues until the `rand_number == stop_number` condition is met, at which point the function returns the list `rand_tries`. This is a bit confusing and counter-intuitive at first, and, as already mentioned, there almost never any need for it once you know how to use a loop. Nevertheless, we can run the function to see the output.

```
new_rec1 <- recursive_function(stop_number = 4, interval = 1:10);
return(new_rec1);
```

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

We can try it again.

```
new_rec2 <- recursive_function(stop_number = 4, interval = 1:10);
return(new_rec2);
```

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

One more time.

```
new_rec3 <- recursive_function(stop_number = 4, interval = 1:10);
return(new_rec3);
```

```
## [1] 9 6 8 2 4
```

Note that we get a different answer each time, with a vector of varying lengths.

Important function considerations

There are a few additional things to keep in mind when writing functions (note, these are useful points from Thiago that I would have probably otherwise forgotten to include).

Function environment

The code run within an argument executes within its own local environment. What this means is that any objects assigned in a function cannot be used unless they are returned by the function. Here is a quick example.

```
sum_three_vals <- function(starting_val){
  aa <- starting_val + 0;
  bb <- starting_val + 2;
  cc <- starting_val + 4;
  dd <- aa + bb + cc;
  return(dd);
}
```

The function `sum_three_vals` defined above takes one argument `starting_val` and uses it to assign a value to `aa`, `bb`, and `cc` in the function. It then sums `aa`, `bb`, and `cc` and returns the summed value.

```
sum_three_vals(starting_val = 1);
```

```
## [1] 9
```

We get the answer of 9 as expected, but note that the 'global' environment outside of the function does not retain `aa`, `bb`, or `cc`. If we try to print one of these values, R cannot find it.

```
print(aa);
```

```
## Error in print(aa): object 'aa' not found
```

This is not true in the other direction, going from the global to the local. For example, say that we first define `xx`, `yy`, and `zz`, then a function summing them up.

```
xx         <- 1;
yy         <- 2;
zz         <- 3;
sum_xxyyzz <- function(){
  dd <- xx + yy + zz;
  return(dd);
}
```

We can now see that the function does technically work.

```
sum_xxyyzz()
```

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

Nevertheless, this is *really* not good practice. The `sum_xxyyzz` function is not portable. You could not actually use it unless the three `xx`, `yy`, and `zz` were already calculated, and if these values change, then the function could return unexpected values. It is important to always make functions self contained, so we can rewrite the above.

```
sum_xxyyzz <- function(xx, yy, zz){
  dd <- xx + yy + zz;
  return(dd);
}
```

Now we specify all of the arguments that the function takes, and the funciton is not reliant upon any other objects in the global environment. Note that the function no longer works by itself as before.

```
sum_xxyyzz();
```

```
## Error in sum_xxyyzz(): argument "xx" is missing, with no default
```

This is because the function is looking for `xx`, `yy`, and `zz` to be assigned as arguments, and with no defaults, it will (correctly and helpfully) produce an error message.

Order of arguments

Note that if arguments are not specified, then R assumes that any values input are defined in the order of the function arguments. We can consider a function that just adds two values, `val1` and two times `val2`.

```
add_two_eg <- function(val1, val2){
  out_val <- val1 + (2 * val2);
  return(out_val);
}
```

We can specify what we want the values to be in any order. Consider `val1 = 1` and `val2 = 2`.

```
add_two_eg(val1 = 1, val2 = 2);
```

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

We can write this with the order of arguments flipped and get the same answer.

```
add_two_eg(val2 = 2, val1 = 1);
```

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

But if we do not specify the arguments explicitly, then R just assumes that they are in the order of the original function `val1, val2`. We can see this below, where we get a different answer if we set `2` first.

```
add_two_eg(2, 1);
```

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

In the above, R just assumes that we meant `add_two_eg(val1 = 2, val2 = 1);`.

Function returns

Note that all of the examples that we have used return a single value. This was done for simplicity, but it is not necessary. For example, we might want to return a list of values instead.

```
func_list <- function(name = "A name", value = 3, vector = 1:10){
  new_list <- list(name, value * 2, vector);
  return(new_list);
}
```

The above function is a very simple example. It simply returns the three arguments `name`, `value` (times 2), and `vector` as a list.

```
func_list(name = "Grace Hopper", value = 85, vector = 1906:1992);
```

```
## [[1]]
## [1] "Grace Hopper"
##
## [[2]]
## [1] 170
##
## [[3]]
##  [1] 1906 1907 1908 1909 1910 1911 1912 1913 1914 1915 1916 1917 1918 1919 1920
## [16] 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935
## [31] 1936 1937 1938 1939 1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950
## [46] 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965
## [61] 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980
## [76] 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992
```

Do call

Lastly, we can use the function `do.call` to read arguments into a function as a list. Here is what that looks like, using the `func_list` example from above.

```
my_args <- list("Grace Hopper", 85, vector = 1906:1992)
do.call(what = func_list, args = my_args);
```

```
## [[1]]
## [1] "Grace Hopper"
##
## [[2]]
## [1] 170
##
## [[3]]
##  [1] 1906 1907 1908 1909 1910 1911 1912 1913 1914 1915 1916 1917 1918 1919 1920
## [16] 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935
## [31] 1936 1937 1938 1939 1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950
## [46] 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965
## [61] 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980
## [76] 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992
```

The `do.call` function seems a bit unnecessary, bit it is actually quite useful in some context. I have used it when writing R packages that include functions with many arguments, and a need for flexibility given user input. But it is rarely needed, in my experience, for data analysis.

Writing Loops in R

Being able to use loops is a critical skill in programming and working with large arrays of data. Loops make it possible to repeat a set of instructions (i.e., code) `for` a particular set of conditions (e.g., for a range of numbers from 1 to 1000), or `while` a set of conditions still applies (e.g., while a value is still greater than zero). Hence, the use of for loops and while loops are fundamental for writing and running code efficiently (note that other types of loops also exist, but I will not focus on them now). Here I will introduce the key concepts of programming with loops, with particular emphasis on getting started with some practical uses of loops in the R programming language for scientific researchers.

The R programming language includes many base level functions to perform tasks that would otherwise require loops (e.g., functions such as apply and tapply, which effectively repeat a set of instructions to summarise values in an array). Tens of thousands of downloadable packages (code, data, and documentation bundled together, written by and for R users) are available in R, most of which include their own functions that can perform specific tasks required in scientific research (e.g., vegan, dplyr, and shiny). Hence, successful data analysis in R can often just be a matter of finding and using an appropriate and reliable package for a given task. Nevertheless, unique data sets and models often require unique code, so base and package functions cannot always be found to do custom tasks. In many situations, the ability to use loops to repeat tasks will make it possible to quickly and confidently develop reproducible code in data analysis. In the long term, this will likely save time; in the short term, it creates an opportunity to develop and practice coding skills.

Below I will demonstrate how to use for loops and while loops.

- A **for loop** iterates a set of instructions *for* a predetermined set of conditions (i.e., when you know how many times you need to iterate the same set of instructions)
- A **while loop** iterates a set of instructions *while* some condition(s) remains satisfied (i.e., when you might not know how many times you need to iterate the same set of instructions, but you do know when the iterations need to stop)

It is not important to completely understand the two definitions above before getting started, just as it is not important to understand a verbal definition of 'multiplication' before learning to multiply two numbers. Seeing examples of loops in practice will make it clear how they work and when to use them. In the next section, I will therefore start with some key examples to show how for loops can be used in R. I will then do the same with while loops. Finally, I will provide some practice problems and additional resources for using loops in programming.

The for loop in R: getting started

Different languages have differ syntaxes for writing for loops. In R, the syntax is as follows:

```
for(index in set_of_conditions){
  # Code that gets repeated for each condition
}
```

In the above, anything within the bracketed `{ }` gets repeated with `index` being substituted, sequentially, for all possible conditions `in` the `set_of_conditions`. A more concrete example will help:

```
for(i in 1:10){
  print(i);
}
```

We can interpret the line `for(i in 1:10)` verbally to explain what is going on in plain English:

Substitute the index `i` for each value from 1 to 10 (i.e., 1, 2, 3, . . . , 8, 9, 10), and run the following bracketed code with each value in sequence. In other words, run the bracketed code first with `i = 1`, then with `i = 2`, and so forth until finishing the loop with `i = 10`.

Given the above explanation, we can predict what will happen with the example code above, which I now run below.

```
for(i in 1:10){
  print(i);
}
```

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

The code has printed integers from 1 to 10. This is obviously a very simple example of using a loop, but it highlights the basic idea. **There are three key points that I want to note with this example before moving on**

1. the above for loop is effectively doing the same as the code below

```
print(1);
print(2);
print(3);
print(4);
print(5);
print(6);
print(7);
print(8);
print(9);
print(10);
```

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

In the above, I have unrolled the for loop that printed off integers from 1 to 10. But the result is the same. The advantage of using the loop is that it avoids the need to repeat the same code more times than is necessary (consider if we wanted to print values from 1 to 10000 – much less needs to be changed when using

the for loop, and much fewer lines of code need to be written). There is no hard rule for when to use a loop versus when to repeat the same line(s) of code multiple times; it is generally best to use whichever method is most readable to (future) you. In practice, when getting started, it might help to think about what the loops would look like if unrolled – or even write them both ways to confirm that a loop is working as intended.

2. There is nothing special about `i`

In a lot of books and online examples introducing loops, the index `i` is used as it is in my above example. But there is nothing special about `i`, just as there is nothing special about the variable `x` in algebra. The index `i` just serves as a placeholder for whatever actual value is going to be substituted from the set of conditions (i.e., values from 1 to 10 in the above example). We get the exact same result with the following code:

```
for(value_to_be_printed in 1:10){
  print(value_to_be_printed);
}
```

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

The above use of the long `value_to_be_printed` instead of the shorter `i` seems unnecessary at first, but it is actually often helpful to give indices specific names like this to make code more readable. Doing so becomes especially helpful when working with multiple indices and loops within loops (an example of this later), or when the number of lines between the starting `{` and ending `}` brackets becomes larger than can be viewed on a computer screen.

3. There is nothing special about `1:10`

A lot of introductions to for loops in R will show the set of values to be iterated in this way, but there are equally acceptable ways to write it. For example, consider the below:

```
for(i in c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)){
  print(i);
}
```

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

Or even the below, where I first define the set of values with its own variable named `the_set`:

```
the_set <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
for(i in the_set){
  print(i);
}
```

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

The order of numbers **does** matter. For example, we could reverse the order in which numbers are printed by reversing the set of values:

```
for(i in 10:1){
  print(i);
}
```

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

We could even print off the numbers in a random order with the below:

```
random_vec <- sample(x = 1:10, size = 10);
for(i in random_vec){
  print(i);
}
```

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

It is unlikely that there would ever be a need to reverse the order of a set, and for most for loops, the simple `1:N` format will usually be all that that is needed. The point is that there is no reason to feel *constrained* to using this format when writing loops.

The for loop in R: a real example

The examples above were intended only to introduce the general idea of using for loops, and their specific syntax in R. In coding, there is rarely such a need to use for loops to simply print off values. More often, for loops are used to repeat the same set of (often complex) instructions for a set of values. As a real example, I will use use the `nhtemp` data set available in R.

```
data(nhtemp); # Reads in the data set
```

The `nhtemp` data set includes a vector that stores the mean annual temperature in degrees Fahrenheit in New Haven, Connecticut (USA), from 1912 to 1971.

```
print(nhtemp);
```

```
## Time Series:
## Start = 1912
## End = 1971
## Frequency = 1
##  [1] 49.9 52.3 49.4 51.1 49.4 47.9 49.8 50.9 49.3 51.9 50.8 49.6 49.3 50.6 48.4
## [16] 50.7 50.9 50.6 51.5 52.8 51.8 51.1 49.8 50.2 50.4 51.6 51.8 50.9 48.8 51.7
## [31] 51.0 50.6 51.7 51.5 52.1 51.3 51.0 54.0 51.4 52.7 53.1 54.6 52.0 52.0 50.9
## [46] 52.6 50.2 52.6 51.6 51.9 50.5 50.9 51.7 51.4 51.7 50.8 51.9 51.8 51.9 53.0
```

In the above data, the first value 49.9 is therefore the mean temperature from 1912, and the last value 53 is the mean temperature from 1971. We can print these off by using the indices `nhtemp[1]` and `nhtemp[60]` (as there are `length(nhtemp)` $= 60$ temperature years in `nhtemp`).

```
print(nhtemp[1]);
```

```
## [1] 49.9
```

```
print(nhtemp[60]);
```

```
## [1] 53
```

We might want to use these data to analyse how the temperature in New Haven has changed over the years from 1912-1972. The first task would likely be to convert the temperatures from Fahrenheit to Celsius. We can make use of the temperature conversion function `F_convert()` we have already defined. The code below shows how we can use the function in a loop.

```
T_Celsius <- NULL;
for(year in 1:length(nhtemp)){
  T_Celsius[year] <- F_convert(F_temp = nhtemp[year], conversion = "Celsius");
}
print(T_Celsius);
```

```
##  [1]  9.944444 11.277778  9.666667 10.611111  9.666667  8.833333  9.888889
##  [8] 10.500000  9.611111 11.055556 10.444444  9.777778  9.611111 10.333333
## [15]  9.111111 10.388889 10.500000 10.333333 10.833333 11.555556 11.000000
## [22] 10.611111  9.888889 10.111111 10.222222 10.888889 11.000000 10.500000
## [29]  9.333333 10.944444 10.555556 10.333333 10.944444 10.833333 11.166667
## [36] 10.722222 10.555556 12.222222 10.777778 11.500000 11.722222 12.555556
## [43] 11.111111 11.111111 10.500000 11.444444 10.111111 11.444444 10.888889
## [50] 11.055556 10.277778 10.500000 10.944444 10.777778 10.944444 10.444444
## [57] 11.055556 11.000000 11.055556 11.666667
```

There are a few things to note here.

- In the first line of the above, I have defined `T_Celsius` to be a null variable.
- I have used the index `year` rather than `i` to make it easier to remember what I am looping over.
- The loop goes from 1 to the length of `nhtemp` (`length(nhtemp)` = 60). I could have instead just written `1:60`, but the above has the advantage that if the length of `nhtemp` changes for some reason, the loop will still work. As a exercise, try running the above code for `1:40` or `1:80` to see what happens when the number of years to loop over does not match the number of years in `nhtemp`.

Now say that we actually want to know the *change* in temperature from one year to the next, and to make a new vector `Temp_ch` that stores the difference in degrees Celsius from `year` to `year - 1`. While there are ways to make such a vector in R without a loop, using a for loop is probably most intuitive way to do it.

```
T_Celsius <- NULL;
Temp_ch   <- NULL;
for(year in 1:length(nhtemp)){
  T_Celsius[year] <- F_convert(F_temp = nhtemp[year], conversion = "Celsius");
  Temp_ch[year]   <- T_Celsius[year] - T_Celsius[year - 1];
}
print(Temp_ch);
```

```
##  [1]          NA  1.33333333 -1.61111111  0.94444444 -0.94444444 -0.83333333
##  [7]  1.05555556  0.61111111 -0.88888889  1.44444444 -0.61111111 -0.66666667
## [13] -0.16666667  0.72222222 -1.22222222  1.27777778  0.11111111 -0.16666667
## [19]  0.50000000  0.72222222 -0.55555556 -0.38888889 -0.72222222  0.22222222
## [25]  0.11111111  0.66666667  0.11111111 -0.50000000 -1.16666667  1.61111111
## [31] -0.38888889 -0.22222222  0.61111111 -0.11111111  0.33333333 -0.44444444
## [37] -0.16666667  1.66666667 -1.44444444  0.72222222  0.22222222  0.83333333
## [43] -1.44444444  0.00000000 -0.61111111  0.94444444 -1.33333333  1.33333333
## [49] -0.55555556  0.16666667 -0.77777778  0.22222222  0.44444444 -0.16666667
## [55]  0.16666667 -0.50000000  0.61111111 -0.05555556  0.05555556  0.61111111
```

In the new line of code added within the above loop, the temperature in degrees Celsius from the previous year `T_Celsius[year - 1]` is subtracted from the temperature from the current year `T_Celsius[year]`. The value of this difference is then stored in `Temp_ch[year]`, so at the end of the loop, each element of `Temp_ch` stores the difference between the current year's temperature and the last year's temperature.

Note that this newly added line within the for loop is a bit dangerous because `T_Celsius[year - 1]` does not exist when `year = 1` (i.e., at the start of the loop). Since there is no value at `T_Celsius[1 - 1]`, R returns an `NA`, so the first value of `Temp_ch = NA`. This is what we want, but if we are not careful, trusting R to fill things in appropriately might cause us problems later. It is usually better to err on the side of caution and think carefully about what each line is doing, using comments to help the readability. The example below makes everything a bit cleaner and clearer.

```
total_years <- length(nhtemp); # Total years in the data set
# Make vectors with an NA element for each year
T_Celsius   <- rep(x = NA, times = total_years);
Temp_ch     <- rep(x = NA, times = total_years);
for(year in 1:total_years){ # For each year in the data set
  # First calculate the temperature in degrees Celsius
  T_Celsius[year] <- F_convert(F_temp = nhtemp[year], conversion = "Celsius");
  if(year > 1){ # Condition in which difference exists
      Temp_ch[year] <- T_Celsius[year] - T_Celsius[year - 1];
  } # Now T_Celsius[0] will not be attempted
}
print(Temp_ch);
```

```
##  [1]          NA  1.33333333 -1.61111111  0.94444444 -0.94444444 -0.83333333
##  [7]  1.05555556  0.61111111 -0.88888889  1.44444444 -0.61111111 -0.66666667
## [13] -0.16666667  0.72222222 -1.22222222  1.27777778  0.11111111 -0.16666667
## [19]  0.50000000  0.72222222 -0.55555556 -0.38888889 -0.72222222  0.22222222
## [25]  0.11111111  0.66666667  0.11111111 -0.50000000 -1.16666667  1.61111111
## [31] -0.38888889 -0.22222222  0.61111111 -0.11111111  0.33333333 -0.44444444
## [37] -0.16666667  1.66666667 -1.44444444  0.72222222  0.22222222  0.83333333
## [43] -1.44444444  0.00000000 -0.61111111  0.94444444 -1.33333333  1.33333333
## [49] -0.55555556  0.16666667 -0.77777778  0.22222222  0.44444444 -0.16666667
## [55]  0.16666667 -0.50000000  0.61111111 -0.05555556  0.05555556  0.61111111
```

The use of the `if` above is technically unnecessary, but it serves as a nice reminder that it only makes sense to take the difference between temperatures starting in year 2. Now with `T_Celsius` and `Temp_ch` calculated, we can make a nice table of years and temperature values and changes.

```
years <- 1912:1971;
dat   <- cbind(years, nhtemp, T_Celsius, Temp_ch);
```

years	nhtemp	T_Celsius	Temp_ch
1912	49.9	9.944444	NA
1913	52.3	11.277778	1.33333333
1914	49.4	9.666667	-1.61111111
1915	51.1	10.611111	0.94444444
1916	49.4	9.666667	-0.94444444
1917	47.9	8.833333	-0.83333333
1918	49.8	9.888889	1.05555556
1919	50.9	10.500000	0.61111111
1920	49.3	9.611111	-0.88888889
1921	51.9	11.055556	1.44444444
1922	50.8	10.444444	-0.61111111
1923	49.6	9.777778	-0.66666667
1924	49.3	9.611111	-0.16666667
1925	50.6	10.333333	0.72222222
1926	48.4	9.111111	-1.22222222
1927	50.7	10.388889	1.27777778
1928	50.9	10.500000	0.11111111
1929	50.6	10.333333	-0.16666667
1930	51.5	10.833333	0.50000000

years	nhtemp	T_Celsius	Temp_ch
1931	52.8	11.555556	0.72222222
1932	51.8	11.000000	-0.55555556
1933	51.1	10.611111	-0.38888889
1934	49.8	9.888889	-0.72222222
1935	50.2	10.111111	0.22222222
1936	50.4	10.222222	0.11111111
1937	51.6	10.888889	0.66666667
1938	51.8	11.000000	0.11111111
1939	50.9	10.500000	-0.50000000
1940	48.8	9.333333	-1.16666667
1941	51.7	10.944444	1.61111111
1942	51.0	10.555556	-0.38888889
1943	50.6	10.333333	-0.22222222
1944	51.7	10.944444	0.61111111
1945	51.5	10.833333	-0.11111111
1946	52.1	11.166667	0.33333333
1947	51.3	10.722222	-0.44444444
1948	51.0	10.555556	-0.16666667
1949	54.0	12.222222	1.66666667
1950	51.4	10.777778	-1.44444444
1951	52.7	11.500000	0.72222222
1952	53.1	11.722222	0.22222222
1953	54.6	12.555556	0.83333333
1954	52.0	11.111111	-1.44444444
1955	52.0	11.111111	0.00000000
1956	50.9	10.500000	-0.61111111
1957	52.6	11.444444	0.94444444
1958	50.2	10.111111	-1.33333333
1959	52.6	11.444444	1.33333333
1960	51.6	10.888889	-0.55555556
1961	51.9	11.055556	0.16666667
1962	50.5	10.277778	-0.77777778
1963	50.9	10.500000	0.22222222
1964	51.7	10.944444	0.44444444
1965	51.4	10.777778	-0.16666667
1966	51.7	10.944444	0.16666667
1967	50.8	10.444444	-0.50000000
1968	51.9	11.055556	0.61111111
1969	51.8	11.000000	-0.05555556
1970	51.9	11.055556	0.05555556
1971	53.0	11.666667	0.61111111

In the next section, I will move on to consider a more complicated example using nested for loops (i.e., a for loop inside of another for loop).

The for loop in R: nested loops

Loops can be nested inside one another, such that the inner loop is run one time for each iteration of the outer loop. A common example of when nested loops are useful is in working with two dimensional arrays (e.g., tables or matrices). I will share a quick example from community ecology theory, in which species interactions within a community are often represented by square matrices like the one below,

$$M = \begin{bmatrix} -1 & -0.2 \\ -0.3 & -1 \end{bmatrix}.$$

Community ecology theory is not the focus here, so it is fine to skip a couple paragraphs to just move along to the coding problem. For more context though, each element in the above matrix defines how a slight increase in the density of one species affects the density of another species when species densities are at some equilibrium state. Each row and column in M represents a single species, so there are two species in the above matrix. Where rows and column numbers are identical, we have the diagonal of the matrix; this defines how a species affects its own density (i.e., self-regulation). The off-diagonals define how a slight increase in one species' density affects a different species; in the above example, both species decrease each others densities because each has a negative affect on the other (the species in row 1 is negatively affected by species 2 by a magnitude of $M_{1,2} = -0.2$, and the species in row 2 is negatively affected by species 1 by a magnitude of $M_{2,1} = -0.3$). If one of these two off-diagonal elements were positive and the other were negative (e.g., $M_{1,2} = 0.2$, $M_{2,1} = -0.3$), we could interpret this as a predator-prey interaction. If both off-diagonal elements were positive (e.g., $M_{1,2} = 0.2$, $M_{2,1} = 0.3$), we could interpret this as a mutualistic interaction.

To investigate community stability, theoreticians use random matrix theory to test how likely it is that communities with specific properties will return to equilibrium species densities when perturbed (e.g., Allesina and Tang 2015, 2012). Developing this theory sometimes requires generating many large M matrices with random interaction strengths (off-diagonal elements) but uniform interaction types (competitor, predator-prey, or mutualist) and self-regulation (diagonal elements). If we take the case of large M matrices in which all interactions are predator-prey (e.g., a big food web), all pairs of row-column elements need to have opposite signs. In other words, if $M_{i,j}$ is positive, then $M_{j,i}$ needs to be negative. To generate a random matrix with this property, we need go through the elements of M and change the signs of values where appropriate.

The **coding** issue is therefore to build a large matrix in which the sign of $M_{i,j}$ is the opposite of $M_{j,i}$. We also want the absolute values of the off-diagonal elements to be random numbers, and the diagonal elements to be -1. These latter two properties can be made with the following lines of code, which will make a 10×10 matrix as printed off below:

```
M_vals <- rnorm(n = 100, mean = 0, sd = 1);
M_vals <- round(M_vals, digits = 2);
M_mat  <- matrix(data = M_vals, nrow = 10);
diag(M_mat) <- -1; # Adds -1 values to diagonal
print(M_mat);
```

```
##        [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9] [,10]
##  [1,] -1.00  1.10  0.12 -0.38  0.98 -0.20 -0.41  0.43  0.91 -0.81
##  [2,]  0.52 -1.00 -2.27 -0.80  0.25  1.31 -0.47  0.54 -0.10  0.03
##  [3,]  0.54 -1.30 -1.00  1.25 -1.32 -1.12  0.47  1.62 -0.71 -0.61
##  [4,]  0.99 -1.45 -0.45 -1.00 -0.72 -0.66 -1.77  0.69 -0.60  0.25
##  [5,] -3.55 -0.43 -1.14 -0.14 -1.00  0.13  0.19 -0.33 -0.26 -1.48
##  [6,]  1.31  1.76 -0.46  0.54 -0.51 -1.00  0.92 -1.38 -1.47 -1.21
##  [7,] -0.12  1.35 -0.08  0.47  1.54  0.07 -1.00  0.25  0.64 -0.36
##  [8,]  0.12 -0.02  0.53 -0.16  1.05  0.24 -1.14 -1.00 -1.02  1.12
##  [9,] -0.37  0.14 -0.23 -0.24  0.42 -1.87  0.08 -0.40 -1.00  0.97
## [10,]  0.83 -0.53  0.09  1.10 -1.57 -0.46  0.27  1.73 -0.95 -1.00
```

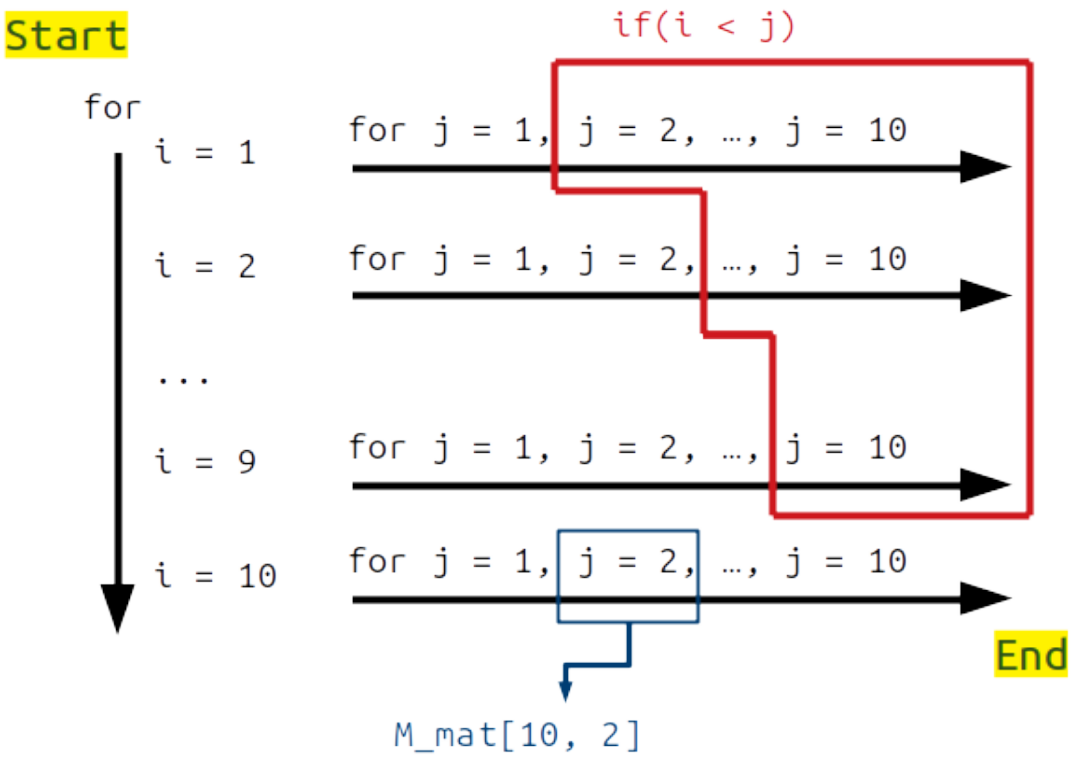
The above random matrix has diagonal elements all equal to -1, and off-diagonal elements independently drawn from a standard normal distribution $\mathcal{N}(0, 1)$. **The task is now to to make sure that pairs of off-diagonal elements $M_{i,j}$ and $M_{j,i}$ have opposite signs**. In other words, if `M_mat[1, 3]` is positive, then `M_mat[3, 1]` should be negative (recall that R indices in brackets refer first to the row, then the column

of a matrix: `M_mat[row, column]`). Unlike the previous problems in these notes, it is difficult to see how to create such a matrix without using loops (or editing the values by hand). We need to iterate over `M_mat` `for` each row and `for` each column, reversing the signs of off-diagonal elements whenever necessary. To do this, we can use a `for` loop within another `for` loop – the outer loop iterates over rows, and the inner loop iterates over columns. Whenever a pair of elements `M_mat[i, j]` and `M_mat[j, i]` are found to have the same sign, `M_mat[i, j]` is multiplied by -1.

```
N_species <- dim(M_mat)[1]; # Get total row & col number
for(i in 1:N_species){ # For each row in the matrix
    for(j in 1:N_species){ # For each column in the row
        if(i < j){ # Only need to look at upper triangle
            elem_sign <- M_mat[i, j] * M_mat[j, i];
            if(elem_sign > 0){
                M_mat[i, j] <- -1 * M_mat[i, j];
            }
        }
    } # Finish all columns in the row
} # Finish all rows
print(M_mat);
```

```
##         [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9] [,10]
##  [1,] -1.00 -1.10 -0.12 -0.38  0.98 -0.20  0.41 -0.43  0.91 -0.81
##  [2,]  0.52 -1.00  2.27  0.80  0.25 -1.31 -0.47  0.54 -0.10  0.03
##  [3,]  0.54 -1.30 -1.00  1.25  1.32  1.12  0.47 -1.62  0.71 -0.61
##  [4,]  0.99 -1.45 -0.45 -1.00  0.72 -0.66 -1.77  0.69  0.60 -0.25
##  [5,] -3.55 -0.43 -1.14 -0.14 -1.00  0.13 -0.19 -0.33 -0.26  1.48
##  [6,]  1.31  1.76 -0.46  0.54 -0.51 -1.00 -0.92 -1.38  1.47  1.21
##  [7,] -0.12  1.35 -0.08  0.47  1.54  0.07 -1.00  0.25 -0.64 -0.36
##  [8,]  0.12 -0.02  0.53 -0.16  1.05  0.24 -1.14 -1.00  1.02 -1.12
##  [9,] -0.37  0.14 -0.23 -0.24  0.42 -1.87  0.08 -0.40 -1.00  0.97
## [10,]  0.83 -0.53  0.09  1.10 -1.57 -0.46  0.27  1.73 -0.95 -1.00
```

Note that in the matrix `M_mat` modified above, all pairs of off-diagonal elements `M_mat[i, j]` and `M_mat[j, i]` have opposite signs. Why did that work? We can start with the loops, the outer of which (`for(i in 1:N_species)`) started going through rows starting with row `i = 1`. While `i = 1`, the inner loop (`for(j in 1:N_species)`) went through all columns from 1 to 10 in row 1. Each unique combination of row `i` and column `j` identified a unique matrix element `M_mat[i, j]`, and the code then checked to see if any action needed to be taken in two ways. First, the code checked to see `if(i < j)` – if not, then the whole bracketed `if` statement is skipped and we move on to the next column `j`. This `if` statement prevents the code from unnecessarily checking the same `i` and `j` pair twice, and prevents it from changing the diagonal where `i == j`. Second, the code assigning `elem_sign` checks to see if `M_mat[i, j]` and `M_mat[j, i]` have opposing signs by multiplying the two values together (two positives or two negatives multiplied together will equal a positive value for `elem_sign`; one positive and one negative will equal a negative value). If `elem_sign > 0`, then we know that `M_mat[i, j]` and `M_mat[j, i]` are either both positive or both negative, so we fix this by changing the sign of `M_mat[i, j]` (multiplying by -1). The figure below gives a visual representation of what is happening.



In the figure above, we start with the case in which `i = 1` (i.e., the first row), and move through each value of `j` from `j = 1` to `j = 10`. If `M_mat[i, j]` is in the upper right triangle of the matrix (i.e., `i < j`; shown in red), then the code checks to see if both `M_mat[i, j]` and `M_mat[j, i]` have the same sign. The loop ends when it has moved through all values of `i` from 1 to 10, hence looking at each element `M_mat[i, j]` in the matrix.

Once the logic of the nested loop makes sense, the rest comes down to remembering the syntax for for coding loops in R correctly. This comes with practice, so I have included some practice problems using loops below. Next, I will have a (briefer) look at the `while` loop in R. The general idea of iterating the same task many times will be the same, but the conditions under which the task is iterated will change slightly.

The while loop in R

The general idea of a `while loop` is the same as that of a `for loop`. In both cases, we are repeating the same task multiple times. But whereas we could specify the full range of values `in` which to substitute some value (e.g., `i`) within a `for` loop, in a `while` loop, we only specify the conditions under which to continue iterating. The printing of numbers from 1 to 10, as done with the `for` loop above, can also be done with the `while` loop below.

```
i <- 1;
while(i <= 10){
    print(i);
    i <- i + 1;
}
```

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

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

There are few important things to note.

1. We need to specify an initial value of `i = 1` (else `print(i)` will not return anything).
2. The loop will continue as long as `i` is less than or equal to 10 (`while(i <= 10)`).
3. It is critical to increment `i` within the loop (i.e., add a value of 1 at the end so `i <- i + 1`).

If we had forgotten number 3, then the value of `i` would always be 1. Aside from only printing the number 1 many times (rather than 1-10), notice that the loop would never actually terminate because `i` would *always* stay less than or equal to 10. This situation is called an 'infinite loop', and will result in a situation where it is necessary to terminate the loop manually (i.e., tell R to stop it, either using the red stop sign in the Rstudio console, or by holding down 'CTRL + C'). This is almost always to be avoided, but to see what happens, remove the line `i <- i + 1;` and run the rest of code above.

Sometimes the use of `for` versus `while` loops is a matter of personal preference, such as with the simple example of printing a set of numbers from 1 to 10. In some cases, however, use of a `while` loop will make coding easier.

Consider a situation in which data need to be randomly sampled from a subset of 100 out of 1000 total different entities (the details do not matter – these entities could be different lochs, fields, or trees in a park). If we just need to sample 100 values out of 1000 without replacement, we can do this with a single line of R code:

```
subset <- sample(x = 1:1000, size = 100, replace = FALSE);
print(subset);
```

```
##   [1] 444 727 151 397 854 344 125 798 777 117 562 624 347 771 380 698 377 612
##  [19] 860 651 661 823 358 454 983 939 671 501 201 842 183 285 956 573 571 328
##  [37] 127 388 530 241 323 550 582 653 108 497  65 620 154 521 907 369 411 214
##  [55] 361 791  27 905 508 488 165 667 472 268 672 827  28 576 548 808 505 220
##  [73] 880 988 387   6 889 104 663 292 417 157 409 692 322  55 269 296 641 963
##  [91] 368 429  60 236 549 307 242 166 381 121
```

This is easy enough, but what if, having already chosen these 100 entities, we decide that we need *another* 100, for a total of 200 unique samples (without replacement). We could find a creative way of using `sample` again in R (give this a try), but there is a logical way to do this with a `while` loop. The idea is to sample a single value from `1:1000`, then check to see if that value is already in the `subset`. If it is in the `subset`, then throw it out and keep going. If it is not in the `subset`, add it. Continue until the size of `subset` is 200.

```
while(length(subset) <= 200){
    samp <- sample(x = 1:1000, size = 1);
    if(samp %in% subset == FALSE){
        subset <- c(subset, samp);
    }
}
print(subset);
```

```
##   [1] 444 727 151 397 854 344 125 798 777 117 562 624 347 771 380 698 377 612
##  [19] 860 651 661 823 358 454 983 939 671 501 201 842 183 285 956 573 571 328
##  [37] 127 388 530 241 323 550 582 653 108 497  65 620 154 521 907 369 411 214
##  [55] 361 791  27 905 508 488 165 667 472 268 672 827  28 576 548 808 505 220
##  [73] 880 988 387   6 889 104 663 292 417 157 409 692 322  55 269 296 641 963
##  [91] 368 429  60 236 549 307 242 166 381 121   7 732 204 336 230 299 266 252
## [109] 708 473 235 886 647 295 746 351 849 638 415 598 525 273 238  82 279 498
## [127] 674 751 888 681 585 902 782 962 989 447 375 492  99 757 846 217 434 688
## [145] 739 728 617 494 468 179 209 222 198 654 515 305  31 753 499 406 990 393
## [163] 229 947 313 383 659 541  51 435   8 881 202 670 330 257 215 465  95 360
## [181] 719 588 352 646 342 882 537 345 745 290 318 247  23 931 341 762 392 424
## [199] 675 446 761
```

The `while` loop above will continue as long as `subset` contains less than 200 numbers. If a randomly selected number from 1 to 1000 is **not** in the `subset`, then it is immediately added to make a bigger `subset` with the new number appended to it. The end result is that the above code has added 100 new unique values to the previous sample of 100.

Practice problems

Below are some practice problems for working with loops. **To see the answers**, click on the '**Details**' arrows to the left at the bottom of each question. Note that your answers might differ from mine – there is more than one way to solve each problem!

1. Using a `for` or `while` loop, print all of the numbers from 1 to 1000 that are multiples of 17. (*Hint: The mod operator `%%` returns the remainder after division. For example, `14 %% 4` would return a value of 2 because 14/4 = 3 with a remainder of 2.*)

```
for(i in 1:1000){
    if(i %% 17 == 0){
        print(i);
    }
}
```

```
## [1] 17
## [1] 34
## [1] 51
## [1] 68
## [1] 85
## [1] 102
## [1] 119
## [1] 136
## [1] 153
## [1] 170
## [1] 187
## [1] 204
## [1] 221
## [1] 238
## [1] 255
## [1] 272
## [1] 289
## [1] 306
```

```
## [1] 323
## [1] 340
## [1] 357
## [1] 374
## [1] 391
## [1] 408
## [1] 425
## [1] 442
## [1] 459
## [1] 476
## [1] 493
## [1] 510
## [1] 527
## [1] 544
## [1] 561
## [1] 578
## [1] 595
## [1] 612
## [1] 629
## [1] 646
## [1] 663
## [1] 680
## [1] 697
## [1] 714
## [1] 731
## [1] 748
## [1] 765
## [1] 782
## [1] 799
## [1] 816
## [1] 833
## [1] 850
## [1] 867
## [1] 884
## [1] 901
## [1] 918
## [1] 935
## [1] 952
## [1] 969
## [1] 986
```

2. In the `nhtemp`, write a loop to add up the temperatures *for all of the even numbered years*, then divide by the total number of even numbered years to get the average.

```
Y <- 1912:1971; # Years
N <- length(nhtemp); # Total temps
A <- 0; # Added temp
C <- 0; # Count
for(i in 1:N){
    if(Y[i] %% 2 == 0){
        A <- A + nhtemp[i];
        C <- C + 1;
    }
}
```

```
avg_A <- A/C;
print(avg_A);
```

```
## [1] 50.8
```

3. Using a `while` loop, calculate the sum of the series, $Y = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \ldots$ to at least 10000 terms. What does the value Y appear to approach as more terms are added? (*Hint: Use `if(){}`, or an `if(){}else{}` to switch from `+` to `-`*)

```
val  <- 0;
deno <- 1;
iter <- 1;
sign <- 1;
while(iter < 1000000){
    if(sign < 0){
        val  <- val - (4/deno);
    }
    if(sign > 0){
        val <- val + (4/deno);
    }
    sign <- -1 * sign;
    deno <- deno + 2;
    iter <- iter + 1;
}
print(val);
```

```
## [1] 3.141594
```

4. From here, write a while loop that prints out standard random normal numbers (use rnorm()) but stops (breaks) if you get a number bigger than 1.

```
i <- 0;
while(i <= 1){
    i <- rnorm(n = 1);
    print(i);
}
```

```
## [1] -0.7975713
## [1] 0.6788758
## [1] -0.5078375
## [1] -0.2595222
## [1] -0.4843359
## [1] 0.9570355
## [1] 0.2276619
## [1] -0.9008369
## [1] 1.001197
```

5. Create an 8×8 matrix `mat` with diagonal values of 1 and off-diagonal values randomly selected from a standard normal distribution $\mathcal{N}(0, 1)$ (using `rnorm`). Using nested `for` loops as in the above notes, swap elements `mat[i, j]` with `mat[j, i]` **only** if `mat[i, j] < mat[j, i]` (so that the higher number is in the lower triangle).

```
mat_v     <- rnorm(n = 64, mean = 0, sd = 1);
mat_v     <- round(mat_v, digits = 2);
mat       <- matrix(data = mat_v, nrow = 8);
diag(mat) <- 1;
N <- dim(mat)[1];
for(i in 1:N){
    for(j in 1:N){
        if(mat[i, j] < mat[j, i]){
            temp_val  <- mat[i, j];
            mat[i, j] <- mat[j, i];
            mat[j, i] <- temp_val;
        }
    }
}
print(mat);
```

```
##       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]
## [1,]  1.00 -1.00 -0.44 -1.05 -0.55  0.68  0.12  0.19
## [2,]  0.24  1.00 -0.93  0.62 -0.43 -0.40  0.66 -0.75
## [3,]  0.26 -0.56  1.00  0.13  0.34  0.95  0.35 -0.98
## [4,] -0.40  0.66  0.72  1.00 -0.14 -1.06  0.91 -0.92
## [5,] -0.06  0.20  0.82  0.31  1.00 -0.31 -2.26 -1.27
## [6,]  0.69 -0.04  1.15 -0.30  1.83  1.00 -1.64 -1.22
## [7,]  1.87  1.48  2.09  1.20 -0.71 -0.12  1.00 -0.39
## [8,]  1.00  0.03  0.58  0.32 -0.81 -1.08  1.31  1.00
```

References

Allesina, Stefano, and Si Tang. 2012. "Stability criteria for complex ecosystems." *Nature* 483 (7388): 205–8. https://doi.org/10.1038/nature10832.

———. 2015. "The stability–complexity relationship at age 40: a random matrix perspective." *Population Ecology*, 63–75. https://doi.org/10.1007/s10144-014-0471-0.

Bumpus, Hermon C. 1898. "Eleventh lecture. The elimination of the unfit as illustrated by the introduced sparrow, *Passer domesticus*. (A fourth contribution to the study of variation.)." *Biological Lectures: Woods Hole Marine Biological Laboratory*, 209–25.

Johnston, R F, D M Niles, and S A Rohwer. 1972. "Hermon Bumpus and natural selection in the House Sparrow *Passer domesticus* ." *Evolution* 26: 20–31.