Data Science for Biologists - 5023Y

Philip Leftwich

2021-09-26

Contents

Chapter 1

Introduction

1.1 Approach and style

This book is designed to accompany the module BIO-5023Y for those new to R looking for best practices and tips. So it must be both accessible and succinct. The approach here is to provide just enough text explanation that someone very new to R can apply the code and follow what the code is doing. It is not a comprehensive textbook.

A few other points:

This is a code reference book accompanied by relatively brief examples - not a thorough textbook on R or data science

This is intended to be a living document - optimal R packages for a given task change often and we welcome discussion about which to emphasize in this handbook

Top tips for the course:

DON'T worry if you don't understand everything

DO ask lots of questions!

1.2 Teaching

1.2.1 Lectures

1.2.2 Workshops

1.3 Introduction to R

R is the name of the programming language itself and RStudio is a convenient interface.], which we will be using throughout the course in order to learn how to organise data, produce accurate data analyses & data visualisations.

Eventually we will also add extra tools like GitHub and RMarkdown for data reproducibility and collaborative programming, check out this short (and very cheesy) intro video.], which are collaboration and version control systems that we will be using throughout the course. More on this in future weeks.

By the end of this module I hope you will have the tools to confidently analyze real data, make informative and beautiful data visuals, and be able to analyse lots of different types of data.

The taught content this autumn will be given to you in several **worksheets**, these will be added to this dynamic webpage each week.

1.4 Getting around on RStudio

1.5 Error!

Chapter 2

Week One

Go to RStudio Cloud and enter the Project labelled `Week One` - this will clone the project and provide you with your own workspace.

Follow the instructions below to get used to the R command line, and how R works as a language.

2.1 Your first R command

In the RStudio pane, navigate to the console (bottom left) and `type or copy` the below it should appear at the >

Hit Enter on your keyboard.

```
10 + 20
```

You should now be looking at the below:

```
> 10 + 20
[1] 30
```

The first line shows the request you made to R, the next line is R's response

You didn't type the `>` symbol: that's just the R command prompt and isn't part of the actual command.

It's important to understand how the output is formatted. Obviously, the correct answer to the sum `10 + 20` is `30`, and not surprisingly R has printed that out as part of its response. But it's also printed out this `[1]` part, which probably doesn't make a lot of sense to you right now. You're going to see that a lot. You can think of `[1] 30` as if R were saying "the answer to the 1st question you asked is 30".

2.1.1 Typos

Before we go on to talk about other types of calculations that we can do with R, there's a few other things I want to point out. The first thing is that, while R is good software, it's still software. It's pretty stupid, and because it's stupid it can't handle typos. It takes it on faith that you meant to type *exactly* what you did type. For example, suppose that you forgot to hit the shift key when trying to type `+`, and as a result your command ended up being `10 = 20` rather than `10 + 20`. Try it for yourself and replicate this error message:

```
10 = 20
```

```
## Error in 10 = 20: invalid (do_set) left-hand side to assignment
```

What's happened here is that R has attempted to interpret `10 = 20` as a command, and spits out an error message because the command doesn't make any sense to it. When a *human* looks at this, and then looks down at his or her keyboard and sees that `+` and `=` are on the same key, it's pretty obvious that the command was a typo. But R doesn't know this, so it gets upset. And, if you look at it from its perspective, this makes sense. All that R "knows" is that `10` is a legitimate number, `20` is a legitimate number, and `=` is a legitimate part of the language too. In other words, from its perspective this really does look like the user meant to type `10 = 20`, since all the individual parts of that statement are legitimate and it's too stupid to realise that this is probably a typo. Therefore, R takes it on faith that this is exactly what you meant... it only "discovers" that the command is nonsense when it tries to follow your instructions, typo and all. And then it whinges, and spits out an error.

Even more subtle is the fact that some typos won't produce errors at all, because they happen to correspond to "well-formed" R commands. For instance, suppose that not only did I forget to hit the shift key when trying to type `10 + 20`, I also managed to press the key next to one I meant do. The resulting typo would produce the command `10 - 20`. Clearly, R has no way of knowing that you meant to *add* 20 to 10, not *subtract* 20 from 10, so what happens this time is this:

```
10 - 20
```

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

In this case, R produces the right answer, but to the the wrong question.

2.1.2 More simple arithmetic

One of the best ways to get to know R is to play with it, it's pretty difficult to break it so don't worry too much. Type whatever you want into to the console and see what happens.

If the last line of your console looks like this

```
> 10+
+
```

and there's a blinking cursor next to the plus sign. This means is that R is still waiting for you to finish. It "thinks" you're still typing your command, so it hasn't tried to execute it yet. In other words, this plus sign is actually another command prompt. It's different from the usual one (i.e., the `>` symbol) to remind you that R is going to "add" whatever you type now to what you typed last time. For example, type `20` and hit enter, then it finishes the command:

```
> 10 +
+ 20
[1] 30
```

Alternatively hit escape, and R will forget what you were trying to do and return to a blank line.

2.1.3 Try some maths

```
1+7
```

```
13-10
```

```
4*6
```

```
12/3
```

Raise a number to the power of another

```
5^4
```

As I'm sure everyone will probably remember the moment they read this, the act of multiplying a number x by itself n times is called "raising x to the n-th power". Mathematically, this is written as x^n. Some values of n have special names: in particular x^2 is called x-squared, and x^3 is called x-cubed. So, the 4th power of 5 is calculated like this:

$$5^4 = 5 \times 5 \times 5 \times 5$$

2.1.4 Perform some combos

Perform some mathematical combos, noting that the order in which R performs calculations is the standard one.

That is, first calculate things inside **B**rackets `()`, then calculate **O**rders of (exponents) `^`, then **D**ivision `/` and **M**ultiplication `*`, then **A**ddition `+` and **S**ubtraction `-`.

Notice the different outputs of these two commands.

```
3^2-5/2
```

```
(3^2-5)/2
```

Similarly if we want to raise a number to a fraction, we need to surround the fraction with parentheses ()

```
16^1/2
```

```
16^(1/2)
```

The first one calculates 16 raised to the power of 1, then divided this answer by two. The second one raises 16 to the power of a half. A big difference in the output.

> **Note - While the cursor is in the console, you can press the up arrow to see all your previous commands. You can run them again, or edit them. Later on we will look at scripts, as an essential way to re-use, store and edit commands.

2.2 Storing outputs

```
10    + 20
```

or this

```
10+20
```

and I would get exactly the same answer **try it for yourself**. However, that doesn't mean that you can insert spaces in any old place. For example you can type `citation()` to get some information about how to cite R. **Try it**

```
citation()
```

… it tells you to cite R when you have used it for data analysis. What happens when the spacing is changed? If you insert spaces in between the word and the parentheses, or inside the parentheses themselves, then all is well. That is, either of these two commands

```
citation ()
```

```
citation(  )
```

will produce exactly the same response. However, what I can't do is insert spaces in the middle of the word. If I try to do this, R gets upset - try it:

```
citat ion()
```

2.2.1 R can sometimes tell that you're not finished yet (but not often)

One more thing I should point out. If you hit enter in a situation where it's "obvious" to R that you haven't actually finished typing the command, R is just smart enough to keep waiting. For example, if you type `10 +` and then press enter, even R is smart enough to realise that you probably wanted to type in another number. So what happens? :

```
> 10+
+
```

and there's a blinking cursor next to the plus sign. What this means is that R is still waiting for you to finish. It "thinks" you're still typing your command, so it hasn't tried to execute it yet. In other words, this plus sign is actually another command prompt. It's different from the usual one (i.e., the `>` symbol) to remind you that R is going to "add" whatever you type now to what you typed last time. For example, type `3` and hit enter, then it finishes the command:

```
> 10 +
+ 20
[1] 30
```

And as far as R is concerned, this is *exactly* the same as if you had typed `10 + 20`. Using the `+` sign is a big part of stringing together commands when we start to build complex plots, so you are likely to stumble over using `+` quite a lot. Similarly, consider the `citation()` command that we talked about in the previous section. Suppose you hit enter after typing `citation(`. Once again, R is smart enough to realise that there must be more coming – since you need to add the `)` character – so it waits. I can even hit enter several times and it will keep waiting. If you want to get out of this situation, just hit the 'escape' key:

```
> citation(
+
+
+ )
```

What about if I typed `citation` and hit enter? In this case we get something very odd, something that we definitely *don't* want, at least at this stage. Here's what happens:

```
citation
```

```
citation
## function (package = "base", lib.loc = NULL, auto = NULL)
## {
##     dir <- system.file(package = package, lib.loc = lib.loc)
##     if (dir == "")
##         stop(gettextf("package '%s' not found", package), domain = NA)
BLAH BLAH BLAH
```

where the `BLAH BLAH BLAH` goes on for rather a long time, and you don't know enough R yet to understand what all this gibberish actually means (of course, it doesn't actually say BLAH BLAH BLAH - it says some other things we don't understand or need to know that I've edited for length) This incomprehensible output can be quite intimidating to novice users, and unfortunately it's very easy to forget to type the parentheses; so almost certainly you'll do this by accident. Do not panic when this happens. *Simply ignore the gibberish* and hit Escape if needed

2.3 Doing simple calculations with R

Okay, now that we've discussed some of the tedious details associated with typing R commands, let's get back to learning how to use the most powerful piece of statistical software in the world as a $2 calculator. So far, all we know how to do is addition. Clearly, a calculator that only did addition would be a bit stupid, so I should tell you about how to perform other simple calculations using R. But first, some more terminology. Addition is an example of an "operation" that you can perform (specifically, an arithmetic operation), and the ***operator*** that performs it is `+`. To people with a programming or mathematics background, this terminology probably feels pretty natural, but to other people it might feel like I'm trying to make something very simple (addition) sound more complicated than it is (by calling it an arithmetic operation). To some extent, that's true: if addition was the only operation that we were interested in, it'd be a bit silly to introduce all this extra terminology. However, as we go along, we'll start using more and more different kinds of operations, so it's probably a good idea to get the language straight now, while we're still talking about very familiar concepts like addition!

2.3.1 Adding, subtracting, multiplying and dividing

So, now that we have the terminology, let's learn how to perform some arithmetic operations in R.These `operators` correspond to the basic arithmetic we learned in primary school: addition, subtraction, multiplication and division.

As you can see, R uses fairly standard symbols to denote each of the different operations you might want to perform: addition is done using the `+` operator, subtraction is performed by the `-` operator, and so on. So if you wanted to find out what 57 times 61 is (and who wouldn't?), you can use R instead of a calculator, like so:

```
57 * 61
```

So that's handy. *By the way what was the answer?*

2.3.2 Taking powers

The first four operations listed in the Table above are things we all learned in primary school, but they aren't the only arithmetic operations built into R.

As I'm sure everyone will probably remember the moment they read this, the act of multiplying a number x by itself n times is called "raising x to the n-th power". Mathematically, this is written as x^n. Some values of n have special names: in particular x^2 is called x-squared, and x^3 is called x-cubed. So, the 4th power of 5 is calculated like this:

$$5^4 = 5 \times 5 \times 5 \times 5$$

One way that we could calculate 5^4 in R would be to type in the complete multiplication as it is shown in the equation above. That is, we could do this

```
5 * 5 * 5 * 5
```

but it does seem a bit tedious. It would be very annoying indeed if you wanted to calculate 5^{15}, since the command would end up being quite long. Therefore, to make our lives easier, we use the power operator instead. When we do that, our command to calculate 5^4 goes like this:

```
5 ^ 4
```

Much easier.

2.3.3 Doing calculations in the right order

Okay. At this point, you know how to take one of the most powerful pieces of statistical software in the world, and use it as a Casio calculator. And as a bonus, you've learned a few very basic programming concepts. In order to use R more effectively, we need to introduce more programming concepts.

In most situations where you would want to use a calculator, you might want to do multiple calculations. R lets you do this, just by typing in longer commands.

```
1 + 2 * 4
```

Clearly, this isn't a problem for R either. However, it's worth stopping for a second, and thinking about what R just did. Clearly, since it gave us an answer of `9` it must have multiplied `2 * 4` (to get an interim answer of 8) and then added 1 to that. But, suppose it had decided to just go from left to right: if R had decided instead to add `1+2` (to get an interim answer of 3) and then multiplied by 4, it would have come up with an answer of `12`.

To answer this, you need to know the ***order of operations*** that R uses. It's actually the same order that you got taught when you were at school: the "***BODMAS***" order. That is, first calculate things inside **B**rackets `()`, then calculate **O**rders of (exponents) `^`, then **D**ivision `/` and **M**ultiplication `*`, then **A**ddition `+` and **S**ubtraction `-`. So, to continue the example above, if we want to force R to calculate the `1+2` part before the multiplication, all we would have to do is enclose it in brackets:

```
(1 + 2) * 4
```

This is a fairly useful thing to be able to do. The only other thing I should point out about order of operations is what to expect when you have two operations that have the same priority: that is, how does R resolve ties? For instance, multiplication and division are actually the same priority, but what should we expect when we give R a problem like `4 / 2 * 3` to solve? If it evaluates the multiplication first and then the division, it would calculate a value of two-thirds. But if it evaluates the division first it calculates a value of 6. The answer, in this case, is that R goes from *left to right*, so in this case the division step would come first:

```
4 / 2 * 3
```

All of the above being said, it's helpful to remember that *brackets always come first*. So, if you're ever unsure about what order R will do things in, an easy solution is to enclose the thing *you* want it to do first in brackets. There's nothing stopping you from typing `(4 / 2) * 3`. By enclosing the division in brackets we make it clear which thing is supposed to happen first.

2.4 Storing a number as a variable

One of the most important things to be able to do in R (or any programming language, for that matter) is to store information in ***variables***. At a conceptual level you can think of a variable as *label* for a certain piece of information, or even several different pieces of information. When doing statistical analysis in R all of your data (the variables you measured in your study) will be stored as variables in R, let's look at the very basics for how we create variables and work with them.

2.4.1 Variable assignment using `<-`

Since we've been working with numbers so far, let's start by creating variables to store our numbers. And since most people like concrete examples, let's invent one. Suppose I'm trying to calculate how many apples I have (hard hitting issues only here). What I want to do is assign a ***value*** to my variable `apples`, and that value should be `5`. We do this by using the ***assignment operator***, which is `<-`. Here's how we do it copy this code:

```
apples <- 5
```

then try this

```
apples
```

So I have 5 apples, that's nice to know. Anytime you can't remember what R has got stored in a particular variable, you can just type the name of the variable and hit enter.

2.4.2 Doing calculations using variables

Okay, let's get back to my original story. In my quest to up my Vitamin C intake, I also want to know how many oranges I have

```
apples <- 5
oranges <- 7
```

The nice thing about variables (in fact, the whole point of having variables) is that we can do anything with a variable that we ought to be able to do with the information that it stores. That is, since R allows me to add `5` and `7`

```
5 + 7
```

it also allows me to add apples and oranges

```
apples + oranges
```

As far as R is concerned, the command is the same. Not surprisingly, I can assign the output of this calculation to a new variable, which I'll call `fruit`. And when we do this, the new variable gets the value `12`, try running this command and then typing `fruit`

```
fruit <- apples+oranges
```

That's fairly straightforward. A slightly more subtle thing we can do is reassign the value of my variable, based on its current value. For instance, suppose that each one of my students decides to bribe me by giving me more fruit!

```
fruit <- fruit + 41
fruit
```

In this calculation, R has taken the old value of `fruit` and added 41 to that value, producing a value of 53. This new value is assigned to the `fruit` variable, **overwriting its previous value**.

2.4.3 Rules and conventions for naming variables

- Variable names cannot include spaces: therefore `my fruit` is not a valid name, but `my_fruit` is.
- Variable names are case sensitive: that is, `Fruit` and `fruit` are *different* variable names.
- Variable names cannot be one of the reserved keywords. These are special names that R needs to keep "safe" from us mere users, so you can't use them as the names of variables. The keywords are: `if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `next`, `break`, `TRUE`, `FALSE`, `NULL`, `Inf`, `NaN`, `NA`, `NA_integer_`, `NA_real_`, `NA_complex_`, and finally, `NA_character_`. Don't feel like you have to remember these: if you make a mistake and try to use one of the keywords as a variable name, R will complain about it like the whiny little automaton it is.

In addition to those rules that R enforces, there are some informal conventions that people tend to follow when naming variables. One of them you've already seen: i.e., don't use variables that start with a period. But there are several others. You aren't obliged to follow these conventions, and there are many situations in which it's advisable to ignore them, but it's generally a good idea to follow them when you can:

- Use informative variable names. As a general rule, using meaningful names like `orange` and `apple` is preferred over arbitrary ones like `variable1` and variable2. Otherwise it's very hard to remember what the contents of different variables actually are.

- Use short variable names. Typing is a pain and no-one likes doing it. So we much prefer to use a name like `apple` over a name like `pink_lady_apple`.
- Use one of the conventional naming styles for multi-word variable names. Suppose I want to name a variable that stores "my new salary". Obviously I can't include spaces in the variable name, so how should I do this? There are three different conventions that you sometimes see R users employing. Firstly, you can separate the words using periods or dashes, which would give you `my-new-salary` as the variable name. Alternatively, you could separate words using underscores, as in `my_new_salary` or if you are feeling shouty `MY_NEW_SALARY`. Finally, you could use capital letters at the beginning of each word (except the first one), which gives you `myNewSalary` as the variable name. There's no very strong reason to prefer one over the other, but we are going to being learning `tidyverse` style R, which for consistency uses **'snake_case'**

2.5 Using functions

The symbols `+`, `-`, `*` and so on are examples of `operators`. As we've seen, you can do quite a lot of calculations just by using these operators. However, in order to do more advanced calculations (and later on, to do actual statistics), you're going to need to start using `functions`. To get started, suppose I wanted to take the square root of 225. The square root, in case your high school maths is a bit rusty, is just the opposite of squaring a number. So, for instance, since "5 squared is 25" I can say that "5 is the square root of 25". The usual notation for this is

$$\sqrt{25} = 5$$

To calculate the square root of 25, I can do it in my head pretty easily, since I memorised my multiplication tables when I was a kid. It gets harder when the numbers get bigger, and pretty much impossible if they're not whole numbers. This is where something like R comes in very handy.

Let's say I wanted to calculate $\sqrt{225}$, the square root of 225. There's two ways I could do this using R.

Firstly, $\sqrt{x}$ is always the same as $x^{0.5}$ so I could use the power operator `^`, just like we did earlier:

```
225 ^ 0.5
```

However, there's a second way that we can do this, since R also provides a ***square root function***, `sqrt()`. To calculate the square root of 255 using this function, what I do is insert the number `225` in the parentheses. That is, the command I type is this:

```
sqrt(225)
```

and as you might expect from our previous discussion, the spaces in between the parentheses are purely cosmetic. I could have typed `sqrt(225)` or `sqrt( 225 )` and gotten the same result. When we use a function to do something, we generally refer to this as ***calling*** the function, and the values that we type into the function (there can be more than one) are referred to as the ***arguments*** of that function.

Obviously, the `sqrt()` function doesn't really give us any new functionality, since we could have performed the square root calculations by using the power operator `^`. However, there are lots of other functions in R: in fact, almost everything of interest is an R function of some kind.

2.5.1 Function arguments, their names and their defaults

There's two more fairly important things that you need to understand about how functions work in R, and that's the use of "named" arguments, and default values" for arguments. To understand what these two concepts are all about, I'll introduce another function. The `round()` function can be used to round some value to the nearest whole number. For example, I could type this:

```
round(3.1415)
```

Pretty straightforward, really. What's happening is that this function contains an `argument` that the number needs to be rounded (i.e., `3.1415`)

However, suppose I only wanted to round it to two decimal places: that is, I want to get `3.14` as the output. The `round()` function supports this, by allowing you to input a *second* argument to the function that specifies the number of decimal places that you want to round the number to. In other words, I could do this:

```
round(3.14165, 2)
```

What's happening here is that I've specified *two* arguments: the first argument is the number that needs to be rounded (i.e., `3.1415`), the second argument is the number of decimal places that it should be rounded to (i.e., `2`), and the two arguments are separated by a comma. Functions often contain multiple arguments, allowing you to carry out complex tasks.

In this simple example, it's quite easy to remember which argument comes first and which one comes second, but for more complicated functions this is not easy. Fortunately, most R functions make use of ***argument names***. For the `round()`

function, for example the number that needs to be rounded is specified using the `x` argument, and the number of decimal points that you want it rounded to is specified using the `digits` argument. Because we have these names available to us, we can specify the arguments to the function by name. We do so like this:

```
round(x = 3.1415, digits = 2)
```

Notice that this is kind of similar in spirit to variable assignment, except that I used `=` here, rather than `<-`. In both cases we're specifying specific values to be associated with a label. However, there are some differences between what I was doing earlier on when creating variables, and what I'm doing here when specifying arguments, and so as a consequence it's important that you use `=` in this context.

As you can see, specifying the arguments by name involves a lot more typing, but it's also a lot easier to read, and it will be a lot easier for *you* or anyone reading your code to know what a particular function might be doing. This is much better for data reproducibility The writers of R functions will try to use conventional names for this reason making everyone's life easier. Or at least that's the theory…

One important thing to note is that when specifying the arguments using their names, it doesn't matter what order you type them in. But if you don't use the argument names, then you have to input the arguments in the correct order. In other words, these three commands all produce the same output…

```
round(3.14165, 2)
round(x = 3.1415, digits = 2)
round(digits = 2, x = 3.1415)
```

but this one does not…

```
round(2, 3.14165)
```

How do you find out what the correct order is? There's a few different ways, but the easiest one is to look at the help documentation for the function[1].

Okay, so that's the first thing I said you'd need to know: argument names. The second thing you need to know about is default values. Notice that the first time I called the `round()` function I didn't actually specify the `digits` argument at all, and yet R somehow knew that this meant it should round to the nearest whole number. How did that happen? The answer is that the `digits` argument has a ***default value*** of `0`, meaning that if you decide not to specify a value

[1]simply type `help(round)` or `help("round)`and read more (here)[https://www.r-project.org/help.html]

for `digits` then R will act as if you had typed `digits = 0`. This is quite handy: the vast majority of the time when you want to round a number you want to round it to the nearest whole number, and it would be pretty annoying to have to specify the `digits` argument every single time. On the other hand, sometimes you actually do want to round to something other than the nearest whole number, and it would be even more annoying if R didn't allow this! Thus, by having `digits = 0` as the default value, we get the best of both worlds.

2.5.2 Storing many numbers as a vector

At this point we've covered functions in enough detail, so let's return to our discussion of variables. When I introduced variables I showed you how we can use variables to store a single number. In this section, we'll extend this idea and look at how to store multiple numbers within the one variable. In R, the name for a variable that can store multiple values is a ***vector***. So let's create one.

2.5.3 Creating a vector

Let's look at student numbers by module for 2020/21. Suppose I have three modules that I teach on `Data Science`, `Genetics` & `Field Ecology` and I have module enrolment numbers: `41`, `124` & `0`(not running this year - Coronavirus…) respectively.

What I would like to do is have a variable – let's call it `students_by_module` – that stores all this data. The first number stored should be. The simplest way to do this in R is to use the ***combine*** function, `c()`. To do so, all we have to do is type all the numbers you want to store in a comma separated list, like this:[2]

```
students_by_module <- c(41,124,0)
students_by_module
```

To use the correct terminology here, we have a single variable here called `students_by_module`: this variable is a vector that consists of 3 ***elements***.

2.5.4 Getting information out of vectors

Let's consider the problem of how to get information out of a vector. At this point, you might have a sneaking suspicion that the answer has something to do with the `[1]` that R has been printing out. And of course you are correct. Suppose I want to pull out the numbers for the second module in the list.

[2]Notice that I didn't specify any argument names here. The `c()` function is one of those cases where we don't use names. We just type all the numbers, and R just dumps them all in a single variable.

```
students_by_module[2]
```

Yep, that's the enrolment numbers for Genetics. This behaviour makes more sense when you realise that we can use this trick to create new variables. For example, I could create a `february.sales` variable like this:

```
genetics_student_numbers <- students_by_module[2]
genetics_student_numbers
```

Obviously, the new variable `genetics_student_numbers` should only have one element and so when I print it out this new variable, the R output begins with a `[1]` because `124` is the value of the first (and only) element. The fact that this also happens to be the value of the second element of `students_by_module` is irrelevant.

2.5.5 Storing text data

A lot of the time your data will be numeric in nature, but not always. Sometimes your data really needs to be described using text, not using numbers. To address this, we need to consider the situation where our variables store text. To create a variable that stores the word "hello", we can type this:

```
greeting <- "hello"
greeting
```

When interpreting this, it's important to recognise that the quote marks here *aren't* part of the string itself. They're just something that we use to make sure that R *knows* to treat the characters that they enclose as a piece of text data, known as a ***character string***. In other words, R treats `"hello"` as a string containing the word "hello"; but if I had typed `hello` instead, R would go looking for a variable by that name! You can also use `'hello'` to specify a character string.

Okay, so that's how we store the text. Next, it's important to recognise that when we do this, R stores the entire word `"hello"` as a *single* element: our `greeting` variable is *not* a vector of five different letters. Rather, it has only the one element, and that element corresponds to the entire character string `"hello"`. To illustrate this, if I actually ask R to find the first element of `greeting`, it prints the whole string:

```
greeting[1]
```

Of course, there's no reason why I can't create a vector of character strings. For instance, if we were to continue with the example of my attempts to look at the student numbers by module, one variable I might want would include the names of all the `modules`.

```
module_names <- c("Data Science", "Genetics", "Field Ecology")
```

This is a ***character vector*** containing 3 elements, each of which is the name of a month. So if I wanted R to tell me the name of the third module, all I would do is this:

```
modules_names[3]
```

2.5.6 Looking at the type of data in a vector

So now we have looked at vectors that contain two different types of data `numeric` and `character`.

For now let's look at how you check the data class of your vectors

```
class(module_names)
class(students_by_module)
```

You can see that R will tell you what type of data each vector contains. This is really useful, and something we will come back to in the future, because R is stupid and while it tells you what class your data `is` it doesn't always know what class your data `needs` to be, so you have to check and correct it. For now these are both fine, so let's continue:

2.5.7 Naming the elements of a vector

You have seen how the elements of a vector are referred to by their position, and that if you take an element and assign it to a new vector its position will change. One very powerful feature of R is the ability to give names to the elements of a vector

```
names(students_by_module) <- module_names
students_by_module
```

Now when you call the vector students_by_module you can see that each element has been given the corresponding module name. Using the names function we set an argument that the vector students_by_module requires names for each element. We then used <- to set provide a character vector of the same length <- module_names. Now you can call elements of the vector by position *or* name.

2.5.8 Altering the elements of a vector

Sometimes you'll want to change the values stored in a vector. This could be done per element. So I just found out 30 extra students want to join this module because it's so amazing!

```
students_by_module["Data Science"] <- 71
students_by_module
```

Secondly, you often want to alter all of the elements of a vector at once. For instance, suppose I found out that student intake has doubled for next year and we expect each module to go up proportionately

```
students_by_module * 2
```

Of course maybe I want to keep a separate vector for next year's estimated numbers

```
students_by_module <- c(41,124,0)
names(students_by_module) <- module_names
next_year <- students_by_module * 2
```

2.5.9 Storing "true or false" data

Time to move onto a third kind of data. A key concept in that a lot of R relies on is the idea of a ***logical value***. A logical value is an assertion about whether something is true or false. This is implemented in R in a pretty straightforward way. There are two logical values, namely `TRUE` and `FALSE`. Despite the simplicity, a logical values are very useful things. Let's see how they work.

2.5.10 Assessing mathematical truths

In George Orwell's classic book *1984*, one of the slogans used by the totalitarian Party was "two plus two equals five", the idea being that the political domination of human freedom becomes complete when it is possible to subvert even the most basic of truths.

But they didn't have R. R will not be subverted. It has rather firm opinions on the topic of what is and isn't true, at least as regards basic mathematics. If I ask it to calculate `2 + 2`, it always gives the same answer, and it's not bloody 5:

```
2 + 2
```

Of course, so far R is just doing the calculations. I haven't asked it to explicitly assert that $2 + 2 = 4$ is a true statement. If I want R to make an explicit judgement, I can use a command like this:

```
2 + 2 == 4
```

What I've done here is use the ***equality operator***, `==`, to force R to make a "true or false" judgement.[3] Okay, let's see what R thinks of the Party slogan:

```
2+2 == 5
```

Take that Big Brother! Anyway, it's worth having a look at what happens if I try to *force* R to believe that two plus two is five by making an assignment statement like `2 + 2 = 5` or `2 + 2 <- 5`. When I do this, here's what happens:

```
2 + 2 = 5
```

R doesn't like this very much. It recognises that `2 + 2` is *not* a variable (that's what the "non-language object" part is saying), and it won't let you try to "reassign" it. While R is pretty flexible, and actually does let you do some quite remarkable things to redefine parts of R itself, there are just some basic, primitive truths that it refuses to give up. It won't change the laws of addition, and it won't change the definition of the number `2`.

That's probably for the best.

2.5.11 Logical operations

So now we've seen logical operations at work, but so far we've only seen the simplest possible example. You probably won't be surprised to discover that we can combine logical operations with other operations and functions in a more complicated way, like this:

```
3*3 + 4*4 == 5*5
```

or this

```
sqrt(25) == 5
```

Not only that, but as this Table illustrates, there are several other logical operators that you can use, corresponding to some basic mathematical concepts.

[3] Note that this is a very different operator to the assignment operator `=` you saw previously. A common typo that people make when trying to write logical commands in R (or other languages, since the "`=` versus `==`" distinction is important in most programming languages) is to accidentally type `=` when you really mean `==`.

Hopefully these are all pretty self-explanatory: for example, the ***less than*** operator `<` checks to see if the number on the left is less than the number on the right. If it's less, then R returns an answer of `TRUE`:

```
99 < 100
```

but if the two numbers are equal, or if the one on the right is larger, then R returns an answer of `FALSE`, as the following two examples illustrate:

```
100 < 100
100 < 99
```

In contrast, the ***less than or equal to*** operator `<=` will do exactly what it says. It returns a value of `TRUE` if the number of the left hand side is less than or equal to the number on the right hand side. So if we repeat the previous two examples using `<=`, here's what we get:

```
100 <= 100
100 <= 99
```

And at this point I hope it's pretty obvious what the ***greater than*** operator `>` and the ***greater than or equal to*** operator `>=` do! Next on the list of logical operators is the ***not equal to*** operator `!=` which – as with all the others – does what it says it does. It returns a value of `TRUE` when things on either side are **NOT** identical to each other. Therefore, since $2 + 2$ isn't equal to 5, we get:

```
2 + 2 != 5
```

We're not quite done yet. There are three more logical operations that are worth knowing about.

These are the ***not*** operator `!`, the ***and*** operator `&`, and the ***or*** operator `|`. Like the other logical operators, their behaviour is more or less exactly what you'd expect given their names. For instance, if I ask you to assess the claim that "either $2 + 2 = 4$ *or* $2 + 2 = 5$" you'd say that it's true. Since it's an "either-or" statement, all we need is for one of the two parts to be true. That's what the `|` operator does:

```
(2+2 == 4) | (2+2 == 5)
```

On the other hand, if I ask you to assess the claim that "both $2 + 2 = 4$ *and* $2+2 = 5$" you'd say that it's false. Since this is an *and* statement we need both parts to be true. And that's what the `&` operator does:

```
(2+2 == 4) & (2+2 == 5)
```

Finally, there's the *not* operator, which is simple but annoying to describe in English. If I ask you to assess my claim that "it is not true that $2+2=5$" then you would say that my claim is true; because my claim is that "$2+2=5$ is false". And I'm right. If we write this as an R command we get this:

```
! (2+2 == 5)
```

In other words, since `2+2 == 5` is a `FALSE` statement, it must be the case that `!(2+2 == 5)` is a `TRUE` one. Essentially, what we've really done is claim that "not false" is the same thing as "true". Obviously, this isn't really quite right in real life. But R lives in a much more black or white world: for R everything is either true or false. No shades of gray are allowed. We can actually see this much more explicitly, like this:

```
! FALSE
```

Of course, in our $2+2=5$ example, we didn't really need to use "not" `!` and "equals to" `==` as two separate operators. We could have just used the "not equals to" operator `!=` like this:

```
2+2 != 5
```

But there are many situations where you really do need to use the `!` operator. We'll being using this a lot later on

2.5.12 Storing and using logical data

Up to this point, I've introduced *numeric data* (in Sections 2.4 and 2.5.2) and *character data* (in Section 2.5.5). So you might not be surprised to discover that these `TRUE` and `FALSE` values that R has been producing are actually a third kind of data, called *logical data*. That is, when I asked R if `2 + 2 == 5` and it said `[1] FALSE` in reply, it was actually producing information that we can store in variables. For instance, I could create a variable called `is.the.Party.correct`, which would store R's opinion:

```
is.the.Party.correct <- 2 + 2 == 5
is.the.Party.correct
```

Alternatively, you can assign the value directly, by typing `TRUE` or `FALSE` in your command. Like this:

```
is.the.Party.correct <- FALSE
is.the.Party.correct
```

Better yet, because it's kind of tedious to type `TRUE` or `FALSE` over and over again, R provides you with a shortcut: you can use `T` and `F` instead (but it's case sensitive: `t` and `f` won't work).[4] So this works:

```
is.the.Party.correct <- F
is.the.Party.correct
```

but this doesn't:

```
is.the.Party.correct <- f
```

2.5.13 Vectors of logicals

The next thing to mention is that you can store vectors of logical values in exactly the same way that you can store vectors of numbers (Section 2.5.2) and vectors of text data (Section 2.5.5). Again, we can define them directly via the `c()` function, like this:

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

or you can produce a vector of logicals by applying a logical operator to a vector. This might not make a lot of sense to you, so let's unpack it slowly. First, let's suppose we have a vector of numbers (i.e., a "non-logical vector"). For instance, we could use the `students_by_module` vector. Suppose I wanted R to tell me, for each module whether I actually have students to teach. I can do that by typing this:

```
students_by_module > 0
```

and again, I can store this in a vector if I want, as the example below illustrates:

[4] Warning! `TRUE` and `FALSE` are reserved keywords in R, so you can trust that they always mean what they say they do. Unfortunately, the shortcut versions `T` and `F` do not have this property. It's even possible to create variables that set up the reverse meanings, by typing commands like `T <- FALSE` and `F <- TRUE`. This is kind of insane, and something that is generally thought to be a design flaw in R. Anyway, the long and short of it is that it's safer to use `TRUE` and `FALSE`.

```
any_students <- students_by_module > 0
any_students
```

2.5.14 Applying logical operation to text

You can also apply logical operations to text as well. It's just that we need to be a bit more careful in understanding how R interprets the different operations. In this section I'll talk about how the equal to operator `==` applies to text, since this is the most important one. Obviously, the not equal to operator `!=` gives the exact opposite answers to `==`

Okay, let's see how it works. In one sense, it's very simple. For instance, I can ask R if the word `"cat"` is the same as the word `"dog"`, like this:

```
"cat" == "dog"
```

That's pretty obvious, and it's good to know that even R can figure that out. Similarly, R does recognise that a `"cat"` is a `"cat"`:

```
"cat" == "cat"
```

Again, that's exactly what we'd expect. However, what you need to keep in mind is that R is not at all tolerant when it comes to grammar and spacing. If two strings differ in any way whatsoever, R will say that they're not equal to each other, as the following examples indicate:

```
" cat" == "cat"
"cat" == "CAT"
"cat" == "c a t"
```

2.5.15 Quitting R

There's one last thing I should cover in this chapter: how to quit R. When I say this, I'm not trying to imply that R is some kind of pathological addition and that you need to call the R QuitLine or wear patches to control the cravings (although you certainly might argue that there's something seriously pathological about being addicted to R). I just mean how to exit the program. Assuming you're running R in the usual way (i.e., through RStudio or the CLoud), then you can just shut down the application in the normal way by clicking the close windown button. However, R also has a function, called `q()` that you can use to quit, which is pretty handy if you're running R in a terminal window e.g. when using servers - we will cover some of this later in the course.

Regardless of what method you use to quit R, when you do so for the first time R will probably ask you if you want to save the "workspace image". I talked about this in one of my videos, it's annoying and occasionally introduces complications, so if prompted it is usually better to click `NO` (but don't worry if you hit the wrong button).

What does this actually *mean*? What's going on is that R wants to know if you want to save **all** those variables that you've been creating, so that you can use them later. This sounds like a great idea, so it's really tempting to type `y` or click the "Save" button. To be honest though, I very rarely do this, and it kind of annoys me a little bit… what R is *really* asking is if you want it to store these variables in a "default" data file (R.data), which it will automatically reload for you next time you open R. But as we will learn to work with scripts, this is unnecessary.

In fact, every time I install R on a new machine one of the first things I do is change the settings so that it never asks me again. You can do this in RStudio really easily: use the menu system to find the RStudio option; the dialog box that comes up will give you an option to tell R never to whine about this again.

2.6 What is Unix/Linux?

UNIX is a computer operating system. It was first developed in 1969 at Bell Labs. Unix is written in the programming language `C`.

Unix is proprietary software, whereas Linux is *basically* free and open-source Unix.

The Linux Operating System is highly flexible, free, open-source (like R) and uses very little RAM to run (Unlike Windows OS) - as such you find most supercomputers run on Linux. Operationally Linux is almost identical to Unix, and so we often refer to it under the umbrella term of "unix-Like" systems.

2.6.1 Some terms

Here are some terms worth knowing, don't worry about memorising them, it can just be useful to have these to refer to in the future.

Note

You should be very familiar with using a GUI (RStudio), but remember we have spent a lot of time working with files and directories using the the command line (CLI) in R. This is useful practice, because most supercomputers lack a GUI, you must work entirely using the command line.

2.7 Why Learn Unix?

Most sequencing data files are large, and require a lot of RAM to process. As a result most of the work Bioinformaticians do is not hosted on their own computers, instead they "remote-connect" to high performance supercomputers or cluster computers. Almost all of these high performance computers use "Unix-like" operatings systems, the most common of which is Linux.

As stated above Linux is free (so no expensive licenses), open-source so lots of developers, its also well known for being stable, secure, reliable and efficient.

You already have some experience with using a Linux OS - every time you log into RStudio Cloud you are connecting to a supercomputer that runs on Linux. Normally we do not interact directly with the OS, instead we use R and RStudio directly.

But when you click on the RStudio Terminal it provides direct access to a command-line where we can execute commands and functions directly in Linux.

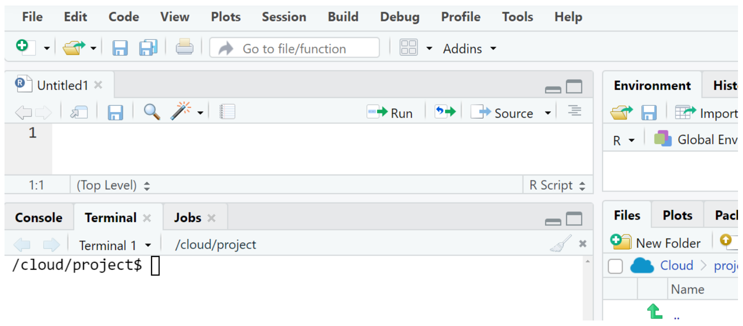
This allows us to start using programs other than R, and potentially use multiple programs & programming languages to work together.

> **Note**
>
> This series of practicals is designed for you to have a first introduction to Bioinformatics, it's about exposure, not memorising or mastering anything. Don't worry about the details!

2.8 Getting started

Before we get started we need a terminal to work in.

- Open the Bioinformatics RStudio Cloud Project in the 5023Y workspace
- Click on the `Terminal` tab next to `Console` in the bottom-left pane of the RStudio GUI, this opens a command-line *Shell*



```
/cloud/project$
```

Figure 2.1: Here is an example of the Terminal tab, right next to the R console

- This is our "command line" where we will be typing all of our commands. We type our commands in `bash`
- The $ is where you start typing from, left of this it tells you what folder you are currently in (`working directory`)
- If you need to, you can exit the Terminal and start a new session easily with options in RStudio

2.9 A few foundational rules

- Just like in R spaces are special, spaces break things apart, as a rule it is therefore better to have functions and file names with dashes (-) or underscores (_) - e.g. "draft_v3.txt" is preferred to "draft v3.txt".
- The general syntax on the command line is: `command argument`. Again this is very similar to R except we don't use brackets e.g. in R we are used to `command(argument)`
- Arguments can be **optional** e.g. if their is a default argument you may not have to write anything. Some functions *require* that arguments are specified. Again this is just like R.

2.10 Let's get started

We will perform a very simple function and get a flavour of the similarities and differences to working in `R`.

`date` is a command that prints out the date and time. Copy and paste this command into your terminal

```
date
```

This prints out the date/time in UTC

More information on using the date function (here)[https://www.geeksforgeeks.org/date-command-linux-examples/]

We can also ask for the output for a particular timezone using the `TZ` function and `date`

```
TZ=Europe/London date
```

Or we can ask the computer what the date will be next Tuesday…

```
date --date="next tue"
```

2.10.1 Downloading data

We will start by typing in an instruction to download data from an online data repository, unpack the contents and inspect it:

- curl is a command line tool for transferring data to and from the server here we will use this to download data from an online repository.
- tar will *unpack* the data from a compressed file format
- cd change the directory so we *land* in the new folder we have made

```
curl -L -o unix_intro.tar.gz https://ndownloader.figshare.com/files/15573746
tar -xzvf unix_intro.tar.gz && rm unix_intro.tar.gz
cd unix_intro
```

Check each command line has run, in the example above you might find that the first two lines run, to download and unpack data, while the last line to change directory doesn't run until you hit enter

2.10.2 More functions

Unlike date, most commands require arguments and won't work without them. head is a command that prints the first lines of a file, so it requires us to provide the file we want it to act on:

```
head example.txt
```

Here "example.txt" is the required argument, and in this case it is also what's known as a positional argument. Whether things need to be provided as positional arguments or not depends on how the command or program we are using was written.

Sometimes we need to specify the input file by putting something in front of it (e.g. some commands will use the -i flag, but it's often other things as well).

Q. What's in the text file? - Click here for Answer

Pretty boring, each line contains the text "This is line" followed by the line number e.g.

- *This is line 1*

- *This is line 2*

etc.

There are also optional arguments for the head command. The default for head is to print the first 10 lines of a file. We can change that by specifying the -n flag, followed by how many lines we want:

```
head -n 5 example.txt
```

How would we know we needed the -n flag for that? There are a few ways to find out. Many standard Unix commands and other programs will have built-in help menus that we can access by providing –help as the only argument:

```
head --help
```

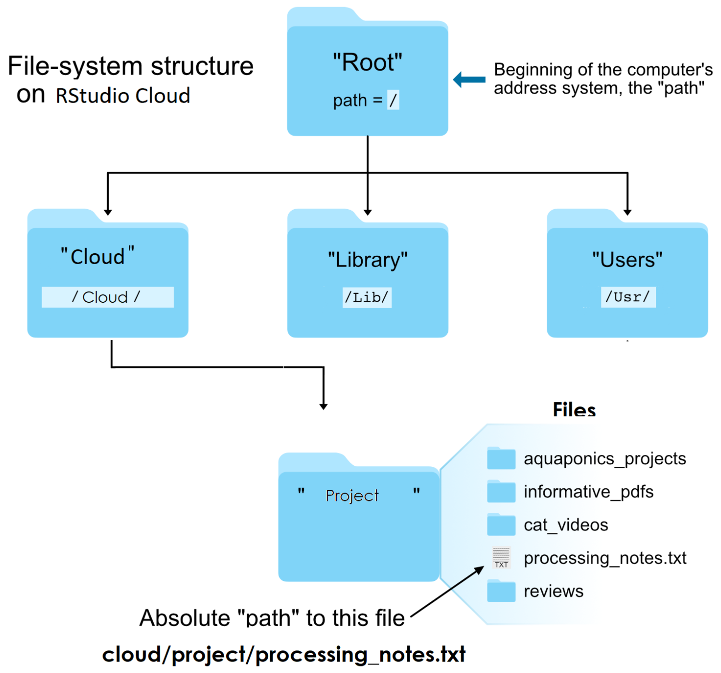
Again this is very similar to the logic in which R commands are strucutred e.g. `??ggplot()` The synatx is similar even if the specific icons or arguments are different.

Remember just like with R, one of your best friends is Google! As you get familiar with any language or OS we might remember a few flags or specific options, but searching for options and details when needed is definitely the norm!

2.11 Unix File Structure

There are two special locations in all Unix-based systems: the "root" location and the current user's "home" location. "Root" is where the address system of the computer starts; "home" is usally where the current user's location starts.

Just to be awkward RStudio Cloud actually has us working in a different location "Cloud", which is underneath Root but separate to home which would be in the "Users" folder.


File-system structure on RStudio Cloud
"Root"
path = /
Beginning of the computer's address system, the "path"
"Cloud"
/ Cloud /
"Library"
/Lib/
"Users"
/Usr/
Files
" Project "
aquaponics_projects
informative_pdfs
cat_videos
processing_notes.txt
reviews
Absolute "path" to this file
cloud/project/processing_notes.txt

We tell the command line where files and directories are located by providing their address, their "path". If we use the pwd command (for print working directory), we can find out what the path is for the directory we are sitting in.

```
pwd
```

And if we use the ls command (for list), we can see what directories and files are in the current directory we are sitting in.

```
ls
```

Note

Why is it important to know this? Usually when you are working on a Unix-like environment there is no GUI (nice-click and point interface), all commands have to be submitted through the terminal. So you would have to get used to navigating directories with typed commands, and it's useful to know what the standard hierarchy is.

2.12 Absolute vs relative file paths

You should be used to these concepts from your work with R projects.

There are two ways to specify the path (address) of the file we want to do something to:

- An **absolute path** is an address that starts from an explicitly specified location: usually the "root" `/` or the "home" `~/` location. (Side note, because we also may see or hear the term, the "full path", is usually the absolute path that starts from the "root" /.)
- A **relative path** is an address that starts from wherever we are currently sitting (the working directory). For example, let's look again at the head command we ran above:

```
    head example.txt
```

What we are actually doing here is using a relative path to specify where the "example.txt" file is located. This is because the command line **automatically looks** in the current working directory if we don't specify anything else about its location.

We can also run the same command on the same file using an **absolute path** - note Rstudio cloud has a slightly unique set-up in that we start from a folder designated cloud:

```
    head /cloud/project/unix_intro/example.txt
```

The previous two commands both point to the same file right now. But the first way, head example.txt, will only work if we are entering it while "sitting" in the directory that holds that file, while the second way will work no matter where we happen to be in the computer.

It is **important to always think about where** we are in the computer when working at the command line. One of the most common errors/easiest mistakes to make is trying to do something to a file that isn't where we think it is. Let's run head on the "example.txt" file again, and then let's try it on another file: "notes.txt":

```
head example.txt
```

```
head notes.txt
```

Here the head command works fine on "example.txt", but we get an error message when we call it on "notes.txt" telling us no such file or directory. If we run the ls command to list the contents of the current working directory, we can see the computer is absolutely right – spoiler alert: it usually is – and there is no file here named "notes.txt".

The ls command by default operates on the current working directory if we don't specify any location, but we can tell it to list the contents of a different directory by providing it as a positional argument:

```
ls
```

```
ls experiment
```

We can see the file we were looking for is located in the subdirectory called "experiment". Here is how we can run head on "notes.txt" by specifying an accurate relative path to that file:

```
head experiment/notes.txt
```

2.13 Moving around

We can also move into the directory containing the file we want to work with by using the `cd` command (**c**hange **d**irectory). This command takes a positional argument that is the path (address) of the directory we want to change into. This can be a relative path or an absolute path. Here we'll use the relative path of the subdirectory, "experiment", to change into it

```
cd experiment/
```

```
pwd
```

```
ls
```

```
head notes.txt
```

Great. But now how do we get **back "up"** to the directory above us? One way would be to provide an absolute path, like `cd /cloud/project/unix_intro`, but there is also a handy shortcut. `..` which are special characters that act as a relative path specifying "up" one level – one directory – from wherever we currently are.

So we can provide that as the positional argument to cd to get back to where we started:

```
    cd ..
```

Moving around the computer like this might feel a bit cumbersome and frustrating at first, but after spending a little time with it, you will get used to it, and it starts to feel more natural.

> **Note**
>
> One way to speed things up is to start using **tab** to perform **tab-completion** often this will auto-complete file names! Press tab twice quickly and it will print all possible combinations.

2.14 Summary

While maybe not all that exciting, these things really are the foundation needed to start utilizing the command line – which then gives us the capability to use lots of tools that only work at a command line, manipulate large files rapidly, access and work with remote computers, and more! These are the fundamental tools that every scientist needs to work with **big data**.

2.15 Summary

You won't get used to operating in bash, or moving around directories using just the command line in a single session. So if you think you are interested in developing your bioinformatic skills, spend some time practising.

Here is a link to a couple of extended tutorials you can bookmark if you want to explore this further:

https://datacarpentry.org/shell-genomics/01-introduction/index.html

2.16 Stretch yourself - optional extras to try a couple of other skills

2.16.1 Creation

I want to create a new directory to store some code files I'm going to write later, so I'll use `mkdir` to create a new directory called Code:

Check you are in the `unix_intro` folder - Click here for Answer

```
pwd
```

Make a new directory called Code - Click here for Answer

```
mkdir Code
```

Check this folder has been created using a list function

```
ls
```

Note that I used a relative file path to create the Code directory - but I could have also specified an absolute filepath to generate that folder in whatever location I want.

There are a few ways to make new files on the command line. The simplest is to generate a blank file with the `touch` command followed by the path (relative or absolute) to the file you want to create

Make a new file called data-science-class.txt - Click here for Answer

```
touch data-science-class.txt
ls -l
```

*Note here I could just use ls to list all files and folders in a directory, but if i set the flag `-l` then it will produce a **l**ong list of files.*

If the entry in the first column is a `d`, then the row in the table corresponds to a directory, otherwise the information in the row corresponds to a file.

The string of characters following the `d` in the case of a directory or following the first `-` in the case of a file represent the permissions for that file or directory - I won't cover that here - but some of the links I provide go into more detail.

We practice what we preach! This site is created with Git and R markdown, using the `bookdown` package. Go ahead and peek behind the scenes.

Long-term, you should understand more about what you are doing. Rote clicking in RStudio may be a short-term survival method but won't work for long.

- trygit is to (command line) Git as swirl is to R. Learn by doing, in small bites.
- The book Pro Git is fantastic and comprehensive.
- Git in Practice by Mike McQuaid is an more approachable book, probably better than Pro Git for most people starting out. Ancillary materials on GitHub.
- Git for Humans is a great set of slides by Alice Bartlett, originally delivered in 2016 at UX Brighton.
- GitHub's own training materials may be helpful. They also point to many other resources

- Find a powerful Git client (chapter **??**) if you'd like to minimize your usage of Git from the command line.
- Hadley Wickham's book R Packages has an excellent chapter on the use of Git, GitHub, and RStudio in R package development. He covers more advanced usage, such as commit best practices, issues, branching, and pull requests.
- Ten Simple Rules for Taking Advantage of Git and GitHub http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004947
- RStudio's guide Version Control with Git and SVN
- The book *Team Geek* has insightful advice for the human and collaborative aspects of version control. It proposes Git strategies suited to different characteristics of teams.