

Data Science for Biologists - 5023Y

Philip Leftwich

2021-09-29

Contents

1	Introduction	5
1.1	Approach and style	5
1.2	Teaching	5
1.3	Introduction to R	6
1.4	Getting around on RStudio	7
1.5	Reading	7
1.6	Get Help!	8
2	Getting to know R - Week One	9
2.1	Your first R command	9
2.2	“true or false” data	12
2.3	Storing outputs	14
2.4	Writing scripts	16
2.5	Error	17
2.6	Functions	17
2.7	Packages	19
2.8	My first data visualisation	20
2.9	Quitting	21
3	Workflow Part One - Week Two	23
3.1	Meet the Penguins	23
3.2	The Question?	26
3.3	Preparing the data	26
3.4	Prepare the RStudio workspace	29
3.5	Get the data into R	32
3.6	Dataframes and tibbles	35
3.7	Clean and tidy	36
3.8	Summing up	40

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

We have:

- one lecture per week
- one workshop per week

These are both timetabled in-person sessions, and you should check Timetabler for up to-date information on scheduling. However, all lessons can be accessed

remotely through Collaborate, and **everything** you need to complete workshops will be available on this site.

If you feel unwell, or cannot attend a session in-person because you need to self-isolate then don't worry you can access everything, and follow along in real time, or work at your own pace.

Questions/issues/errors can all be posted on our Yammer page.

1.2.1 Workshops

The workshops are the best way to learn, they teach you the practical skills you need to become an R wizard



Figure 1.1: courtesy of Allison Horst

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

All of our sessions will run on cloud-based software. All you have to do is make a free account, and join our Workspace BIO-5023Y the sharing link is here.

Once you are signed up - you will see that there are two **Spaces**

- Your workspace
- BIO-5023Y

Make sure you are working in the class workspace - there is a limit to the hours/month on your workspace, so all assignments and project work should take place in the BIO-5023Y space.

Watch these short explainer videos to get used to navigating the environment.

1.4.1 An intro to RStudio

RStudio

Note - people often mix up R and RStudio. R is the programming language (the engine), RStudio is a handy interface/wrapper that makes things a bit easier to use.

1.4.2 Using R Studio Cloud

RStudio Cloud works in exactly the same way as RStudio, but means you don't have to download any software. You can access the hosted cloud server and your projects through any browser connection (Chrome works best), from any computer.

1.5 Reading

There are lots of useful books and online resources to help develop and improve your R knowledge. Throughout this webpage I will be adding useful resources for you.

The core textbook you might want to bookmark is R for Data Science (Hadley Wickham, 2020) but we will add others throughout the course, and their is a bibliography at the end which collects everything together!

1.6 Get Help!

There are a **lot** of sources of information about using R out there. Here are a few helpful places to get help when you have an issue, or just to learn more

- The R help system itself - we cover this in Week one Error
- Vignettes - type `browseVignettes()` into the console and hit Enter, a list of available vignettes for all the packages we have will be displayed
- Cheat Sheets - available at RStudio.com. Most common packages have an associate cheat sheet covering the basics of how to use them. Download/bookmark ones we will use commonly such as `ggplot2`, `Data transformation with dplyr`, `Data tidying with tidyr` & `Data import`.
- Google - I use Google constantly, because I continually forget how to do even basic tasks. If I want to remind myself how to round a number, I might type something like `R round number` - if I am using a particular package I should include that in the search term as well
- Ask for help - If you are stuck, getting an error message, can't think what to do next, then ask someone. It could be me, it could be a classmate. When you do this it is very important that you **show the code**, include the **error message**. "This doesn't work" is not helpful. "Here is my code, this is the data I am using, I want it to do X, and here's the problem I get".

Note - It may be daunting to send your code to someone for help. It is natural and common to feel apprehensive, or to think that your code is really bad. I still feel the same! But we learn when we share our mistakes, and eventually you will find it funny when you look back on your early mistakes, or laugh about the mistakes you still occasionally make!

Chapter 2

Getting to know R - 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 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 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 “true or false” data

Time to make a sidebar onto another 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.

2.2.1 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.

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 `==`.

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.

2.3 Storing outputs

With simple questions like the ones above we are happy to just see the answer, but our questions are often more complex than this. If we need to take multiple steps, we benefit from being able to store our answers and recall them for use in later steps. This is very simple to do we can *assign* outputs to a name:

```
a <- 1+2
```

This literally means please *assign* the value of `1+2` to the name `a`. We use the **assignment operator** `<-` to make this assignment.

Note the shortcut key for `<-` is Alt + - (Windows) or Option + - (Mac)

If you perform this action you should be able to do two things

- You should be able to see that in the top right-hand pane in the **Environment** tab there is now an **object** called `a` with the value of 3.
- You should be able to look at what `a` is by typing it into your Console and pressing Enter

```
a
```

```
> a  
[1] 3
```

You can now call this object at any time during your R session and perform calculations with it.

```
2 * a
```

```
[1] 6
```

What happens if we assign a value to a named object that already exists in our R environment??? for example

```
a <- 10  
a
```

```
[1] 10
```

You should see that the previous assignment is lost, *gone forever* and has been replaced by the new value.

We can assign lots of things to objects, and use them in calculations to build more objects.

```
b <- 5  
c <- a + b
```

Note that if you now change the value of b, the value of c does *not* change. Objects are totally independent from each other once they are made

```
b <- 7  
b  
c
```

Look at the environment tab again - you should see it's starting to fill up now!

Note - RStudio will by default save the objects in its memory when you close a session. These will then be there the next time you logon. It might seem nice to be able to close things down and pick up where you left off, but it's actually quite dangerous. It's messy, and can cause lots of problems when we work with scripts later, so don't do this!!! To stop RStudio from saving objects by default go to the Preferences option and change "Save workspace to .RData on exit" to "Never". Instead we are going to learn how to use scripts to quickly re-run analyses we have been working on.

2.3.1 Choosing names

- 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. R only lets you use certain things as **legal** names. Legal names must start with a letter **not** a number, which can then be followed by a sequence of letters, numbers, ., or __. R does not like using spaces. Upper and lower case names are allowed, but R is case sensitive so `Apple` and `apple` are different.
- My favourite naming convention is `snake_case` short, lower case only, spaces between words are separated with a __. It's easy to read and easy to remember.

2.4 Writing scripts

Until now we have been typing words directly into the Console. This is fine for short/simple calculations - but as soon as we have a more complex, multi-step process this becomes time consuming, error-prone and *boring*. **Scripts** are a document containing all of your commands (in the order you want them to run), they are *repeatable, shareable, annotated records of what you have done*. In short they are incredibly useful - and a big step towards **open** and **reproducible** research.

To create a script go to File > New File > R Script.

This will open a pane in the top-left of RStudio with a tab name of **Untitled1**. In your new script, type some of the basic arithmetic and assignment commands you used previously. When you write a script, make sure it has all of the commands you need to complete your analysis, *in the order you want them to run*.

2.4.1 Commenting on scripts

Annotating your instructions provides yourself and others insights into why you are doing what you are doing. This is a vital aspect of a robust and reproducible workflow. And when you come back to a script, one week, one month or one year from now you will often wonder what a command was for. It is very, very useful to make notes for yourself, and its useful in case anyone else will ever read your script. Make these comments helpful they are for humans to read.

In R we signal a comment with the `#` key. Everything in the line after a `#` is ignored by R and won't be treated as a command. You should see that it is marked in a different colour in your script.

Put the following comment in your script. Try adding a few comments to your previous lines of code

```
# I really love R
```

2.4.2 Running your script

To run the commands from your script, we need to get it into the Console. You could select and copy/paste this into the Console. But there are a couple of faster shortcuts.

- Hit the Run button in the top right of the script pane. Pressing this will run the line of code the cursor is sitting on.
- Pressing Ctrl+Enter will do the same thing as hitting the Run button
- If you want to run the whole script in one go then press Ctrl+A then either click Run or press Ctrl+Enter

2.4.3 Saving your script

Our script now contains code and comments from our first workshop. We need to save it.

Alongside our data, our script is the most precious part of our analysis. We don't need to save anything else, any outputs etc. because our script can always be used to generate everything again. Note the colour of the script - the name changes colour when we have unsaved changes. Press the Save button or go to File > Save as. Give the File a sensible name like "Simple commands in R" and in the bottom right pane under **Files** you should now be able to see your saved script.

You could now safely quit R, and when you log on next time to this project, your script will be waiting for you.

2.5 Error

Things will go wrong eventually, they always do...

R is *very* pedantic, even the smallest typo can result in failure and typos are impossible to avoid. So we will make mistakes. One type of mistake we will make is an **error**. The code fails to run. The most common causes for an error are:

- typos
- missing commas
- missing brackets

There's nothing wrong with making *lots* of errors. The trick is not to panic or get frustrated, but to read the error message and our script carefully and start to *debug*...

... and sometimes we need to walk away and come back later!

2.6 Functions

Functions are the tools of R. Each one helps us to do a different task.

Take for example the function that we use to round a number to a certain number of digits - this function is called **round**

Here's an example

```
round(x = 2.4326782647, digits = 2)
```



Figure 2.1: courtesy of Allison Horst

We start the command with the function name `round`. The name is followed by parentheses `()`. Within these we place the *arguments* for the function, each of which is separated by a comma.

The arguments

- `x = 2.4326782647`
- `digits = 2`

Arguments are the information we give to a function. These arguments are in the form `name = value` the name specifies the argument, and the value is what we are providing. That is the first argument `x` is the number we would like to round, it has a value of `2.4326782647`. The second argument `digits` is how we would like the number to be rounded and we specify `2`.

Ok put the above command in your script and add a comment with `#` as to what you are doing.

2.6.1 Storing the output of functions

What if we need the answer from a function in a later calculation. The answer is to use the assignment operator again.

Can you work out what is going on here? If so copy this into your R script and a `#comment` next to each line.

```
number_of_digits <- 2
my_number <- 2.4326782647
rounded_number <- round(x = my_number,
                         digits = number_of_digits)
```

2.6.2 More fun with functions

Check this out

```
round(2.4326782647, 2)
```

We don't *have* to give the names of arguments for a function to still work. This works because the function `round` expects us to give the number value first, and the argument for rounding digits second. *But* this assumes we know the expected ordering within a function, this might be the case for functions we use a lot. If you give arguments their proper names *then* you can actually introduce them in any order you want.

Try this:

```
round(digits = 2, x = 2.4326782647)
```

But this gives a different answer

```
round(2, 2.4326782647)
```

Are you happy with what is happening here? naming arguments overrides the position defaults

Ok what about this?

```
round(2.4326782647)
```

We didn't specify how many digits to round to, but we still got an answer. That's because in many functions arguments have `defaults` - the default argument here is `digits = 0`. So we don't have to specify the argument if we are happy for `round` to produce whole numbers.

How do we know argument orders and defaults? Well we get to know how a lot of functions work through practice, but we can also use the inbuilt R help. This is a function - but now we specify the name of another function to provide a help menu.

```
help(round)
```

2.7 Packages

An R package is a container for various things including functions and data. These make it easy to do very complicate protocols by using custom-built functions. Later we will see how we can write our own simple functions.

On RStudio Cloud I have already installed several add-on packages, all we need to do is use a simple function to load these packages into our workspace. Once this is complete we will have access to all the custom functions they contain.

Let's try that now:

```
library(ggplot2)
library(palmerpenguins)
```

Errors part 2 Another common source of errors is to call a function that is part of a package but forgetting to load the package. If R says something like “Error in function-name” then most likely the function was misspelled or the package containing the function hasn’t been loaded.

Packages are a lot like new apps extending the functionality of what your phone can do. To use the functionalities of a package they must be loaded *before* we call on the functions or data they contain. So the most sensible place to put library calls for packages is at the very **top** of our script. So let’s do that now.

2.8 My first data visualisation

Let’s run our first data visualisation using the functions and data we have now loaded - this produces a plot using functions from the `ggplot2` package (Wickham et al. (2020a)) and data from the `palmerpenguins` (Horst et al. (2020)) package. Use the `#` comments to add notes on what you are using each package for in your script.

Using these functions we can write a simple line of code to produce a figure. We specify the data source, the variables to be used for the x and y axis and then the type of visual object to produce, colouring them by the species.

Copy this into your console and hit Enter.

```
ggplot(data = penguins,aes(x = bill_length_mm, y = bill_depth_mm)) + geom_point(aes(co
```

Note - you may have noticed R gave you a warning. Not the same as a big scary error, but R wants you to be aware of something. In this case that two of the observations had missing data in them (either bill length or bill depth), so couldn’t be plotted.

The above command can also be written as below, its in a longer style with each new line for each argument in the function. This style can be easier to read, and makes it easier to write comments with `#`. Copy this longer command into your `script` then run it by either highlighting the entire command or placing the cursor in the first line and then hit Run or Ctrl+Enter.

```
ggplot(data = penguins, # calls ggplot function, data is penguins
       aes(x = bill_length_mm, # sets x axis as bill length
           y = bill_depth_mm)) + # sets y axis value as bill depth
       geom_point(aes(colour=species)) # plot points coloured by penguin species
```

2.9 Quitting

- Make sure you have saved any changes to your R script - that's all you need to make sure you've done!
- If you want me to take a look at your script let me know
- If you spotted any mistakes or errors let me know
- Close your RStudio Cloud Browser
- Go to Blackboard to complete a short quiz!

Chapter 3

Workflow Part One - Week Two

Last week we got acquainted with some of the core skills associated with using R and RStudio.

In this workshop we work through the journey of importing and tidying data. Once we have a curated and cleaned dataset we can work on generating insights from the data.

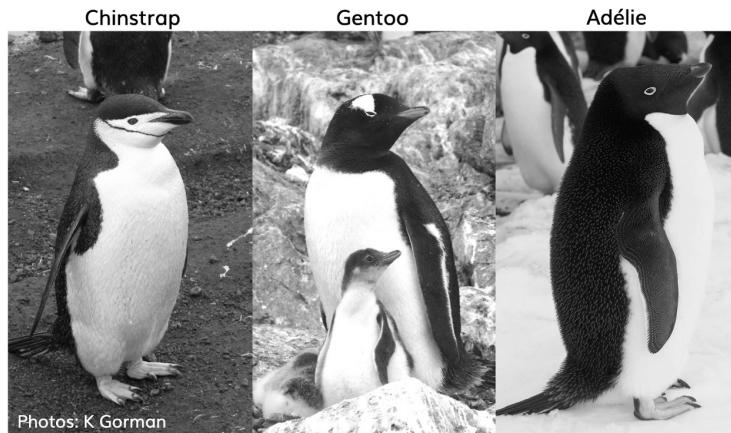
We are going to be working as though we are in the latter stages of a research project, where data has been collected, possibly over several years, to test against our hypotheses.

We have chosen to continue working with a dataset you have been introduced to already - the Palmer Penguins dataset. Previously we loaded a cleaned dataset, very quickly using an R package. Today we will be working in a more realistic scenario - uploading out dataset to our R workspace.

3.1 Meet the Penguins

This data, taken from the `palmerpenguins` (Horst et al. (2020)) package was originally published by Gorman et al. (2014).

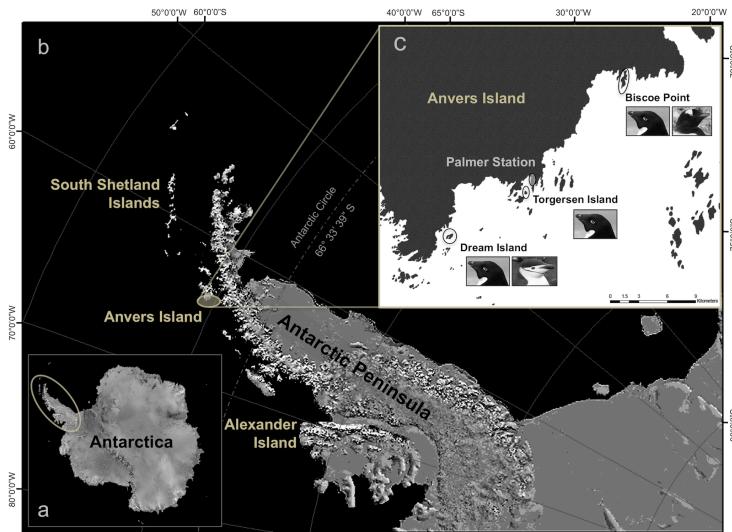
The `palmerpenguins` data contains size measurements, clutch observations, and blood isotope ratios for three penguin species observed on three islands in the Palmer Archipelago, Antarctica over a study period of three years.



These data were collected from 2007 - 2009 by Dr. Kristen Gorman with the Palmer Station Long Term Ecological Research Program, part of the US Long Term Ecological Research Network. The data were imported directly from the Environmental Data Initiative (EDI) Data Portal, and are available for use by CC0 license (“No Rights Reserved”) in accordance with the Palmer Station Data Policy. We gratefully acknowledge Palmer Station LTER and the US LTER Network. Special thanks to Marty Downs (Director, LTER Network Office) for help regarding the data license & use. Here is our intrepid package co-author, Dr. Gorman, in action collecting some penguin data:



Here is a map of the study site



3.1.1 Insights from data

This dataset is relatively simple, as there aren't too many variables to consider. But there are a reasonably large number of datapoints (individual penguins) making it possible to generate insights.

However, there are some specific and rather common problems in this data. Problems that we need to work through *before* we can start to make any visuals or try to draw any conclusions. There are some problems with the variable names, there are some problems with some of the values. There are problems that one of the response variables isn't actually encoded on the dataset (we have to make it).

Today we are going to

- Formulate clear research questions
- Import our dataset
- Learn how to prepare our RStudio Project workspace
- Learn how to clean, tidy and manipulate our data to allow tables, graphs and analyses to be run

Don't worry if you don't understand exactly what each function does at the moment, or struggle to remember every concept we are introduced to. We will cover these again, in lots more detail as we progress. The main point is to get familiar with our process for handling data and organising our projects.

3.2 The Question?

Imagine that you are a Penguin biologist. Chilly.

Imagine that you want to know more about the feeding habits of the different penguin species in the Antarctic. You also wish to characterise features such as bill morphology, and body size and compare them across species. Adelie and Chinstrap penguins are off-shore, shallow diving foragers, while Gentoo's feed closer inshore and are deep-divers.

With simple measuring techniques and identification skills we can sex the penguins, identify their species and take simple non-invasive measurements of features such as body size, flipper length and bill dimensions. You also recently read a paper about the ratios of different Nitrogen and Carbon isotopes in blood, and how these can be used to infer the types of prey that are forming an organism's diet.

3.2.1 Hypotheses

These hypotheses are fairly basic, we have not included directionality or specific expectations of the magnitude of the difference. This would come from doing more research on the subject area.

- Gentoo penguins will have a different beak length to Adelie and Chinstrap penguins
- Gentoo penguins will have a different beak depth to Adelie and Chinstrap penguins
- Gentoo penguins will have a different isotope ratio than Adelie and Chinstrap penguins

3.3 Preparing the data

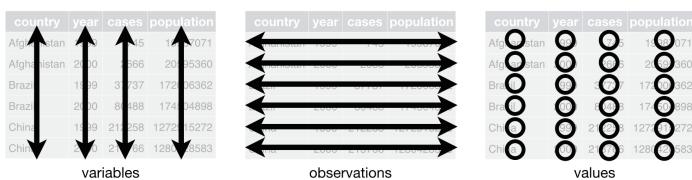
Imagine we have completed our practical study and have our data. The data is probably distributed in lots of places, originally notes collected in the field were probably on paper & notebooks. Then someone will have taken time to transcribe those into a spreadsheet. This will almost certainly have been done by typing all the data in by hand.

It is very important for us to know how we would like our data to be organised at the end. We are going to learn how to organise data using the *tidy* format¹. This is because we are going to use the **tidyverse** packages by Wickham et al. (2019). This is an opinionated, but highly effective method for generating reproducible analyses with a wide-range of data manipulation tools. Tidy data is an easy format for computers to read.

¹(<http://vita.had.co.nz/papers/tidy-data.pdf>)

3.3.1 Tidy data

Here ‘tidy’ refers to a specific structure that lets us manipulate and visualise data with ease. In a tidy dataset each *variable* is in one column and each row contains one *observation*. Each cell of the table/spreadsheet contains the *values*. One observation you might make about tidy data is it is quite long - it generates a lot of rows of data.



Typing data in, using any spreadsheet program (e.g. Excel, Google sheets), if we type in the penguin data, we would make each row contain one observation about one penguin. If we made a second observation about a penguin (say in the next year of the study) it would get a new row in the dataset. You are probably thinking this is a lot of typing and a lot of repetition - and you are right! But this format allows the computer to easily make summaries at any level.

If the data we input to R is “untidy” then we have to spend a little bit of time tidying, we will explore this more later.

Once data has been typed up into a spreadsheet and double/triple-checked against the original paper records, then they are saved as a ‘comma-separated values (CSV)’ file-type. These files are the simplest form of database, no coloured cells, no formulae, no text formatting. Each row is a row of the data, each value of a row (previously separate columns) is separated by a comma.

It is convenient to use something like Excel to type in our data - its much more usefully friendly than trying to type something in csv format. But we don’t save files in the Excel format because they have a nasty habit of formatting or even losing data when the file gets large enough². If you need to add data to a csv file, you can always open it in an Excel-like program and add more information.

It is possible to import data into R in an Excel format, but I recommend sticking with csv formats. Any spreadsheet can be easily converted with the *Save As..* command.

3.3.2 The dataset

For today’s workshop we want to acquire the dataset to work on it. We can retrieve the file we need from here (https://github.com/UEABIO/5023Y_

²<https://www.theguardian.com/politics/2020/oct/05/how-excel-may-have-caused-loss-of-16000-covid-tests-in-england>

	studyName	Sample Number	Species	Region	Island	Stage	Individual ID	Clutch Completion	Date Egg	Culmen Length (mm)	Cu
2	PAL0708	1	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg	N1A1	Yes	11/11/	18.	
3	PAL0708	2	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg	N1A2	Yes	11/11/	17.	
4	PAL0708	3	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg	N2A1	Yes	16/11/	18.	
5	PAL0708	4	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg	N2A2	Yes	16/11/	18.	
6	PAL0708	5	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg	N3A1	Yes	16/11/	18.	
7	PAL0708	6	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg	N3A2	Yes	16/11/	18.	
8	PAL0708	7	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg	N4A1	No	15/11/	18.	
9	PAL0708	8	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg	N4A2	No	15/11/	18.	
10	PAL0708	9	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg	N5A1	Yes	09/11/	18.	
11	PAL0708	10	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg	N5A2	Yes	09/11/	18.	
12	PAL0708	11	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg	N6A1	Yes	09/11/	18.	

```

1 studyName,Sample Number,Species,Region,Island,Stage,Individual ID,Clutch Completion,Date Egg,Culmen Length (mm),Cu
2 PAL0708,1,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",N1A1,Yes,11/11/2007,39.1,18.
3 PAL0708,2,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",N1A2,Yes,11/11/2007,39.5,17.
4 PAL0708,3,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",N2A1,Yes,16/11/2007,40.3,18.
5 PAL0708,4,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",N2A2,Yes,16/11/2007,NA,NA
6 PAL0708,5,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",N3A1,Yes,16/11/2007,NA,NA
7 PAL0708,6,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",N3A2,Yes,16/11/2007,NA,NA
8 PAL0708,7,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",N4A1,No,15/11/2007,NA,NA
9 PAL0708,8,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",N4A2,No,15/11/2007,NA,NA
10 PAL0708,9,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",N5A1,Yes,09/11/2007,NA,NA
11 PAL0708,10,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",N5A2,Yes,09/11/2007,NA,NA
12 PAL0708,11,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",N6A1,Yes,09/11/2007,NA,NA
13 PAL0708,12,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",NA,NA,NA
14 PAL0708,13,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",NA,NA,NA
15 PAL0708,14,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",NA,NA,NA
16 PAL0708,15,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",NA,NA,NA
17 PAL0708,16,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg",NA,NA,NA

```

Figure 3.1: Top image: Penguins data viewed in Excel, Bottom image: Penguins data in native csv format

Workshop/blob/main/data/penguins_raw.csv).

	studyName	Sample Number	Species	Region	Island	Stage	Individual ID	Clutch Completion	Date Egg	Culmen Length (mm)	Cu
2	PAL0708	1	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg	N1A1	Yes	11/11/2007	39.1	18.
3	PAL0708	2	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg	N1A2	Yes	11/11/2007	39.5	17.
4	PAL0708	3	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg	N2A1	Yes	16/11/2007	40.3	18.
5	PAL0708	4	Adelie Penguin (Pygoscelis adeliae)	Anvers	Torgersen	Adult, 1 Egg	N2A2	Yes	16/11/2007	NA	NA

Figure 3.2: Click on the copy raw contents button

We then need to

- Select the copy raw contents button
- Open a blank notepad
- Paste the contents
- Save the file as ‘penguin_data.csv’
- Open the newly saved file in Excel and take a look

We can see a dataset with 345 rows (including the headers) and 17 variables

- Study name: an identifier for the year in which sets of observations were made

- Region: the area in which the observation was recorded
- Island: the specific island where the observation was recorded
- Stage: Denotes reproductive stage of the penguin
- Individual ID: the unique ID of the individual
- Clutch completion: if the study nest observed with a full clutch e.g. 2 eggs
- Date egg: the date at which the study nest observed with 1 egg
- Culmen length: length of the dorsal ridge of the bird's bill (mm)
- Culmen depth: depth of the dorsal ridge of the bird's bill (mm)
- Flipper Length: length of bird's flipper (mm)
- Body Mass: Bird's mass in (g)
- Sex: Denotes the sex of the bird
- Delta 15N : the ratio of stable Nitrogen isotopes 15N:14N from blood sample
- Delta 13C: the ratio of stable Carbon isotopes 13C:12C from blood sample

3.4 Prepare the RStudio workspace

Now we should have our question, we understand more about where the data came from, and we have our data in a CSV format.

The next step of our workflow is to have a well organised project space. RStudio Cloud does a lot of the hard work for you, each new data project can be set up with its own Project space.

We will define a project as a series of linked questions that uses one (or sometimes several) datasets. For example a coursework assignment for a particular module would be its own project, or eventually your final year research project.

A Project will contain several files, possibly organised into sub-folders containing data, R scripts and final outputs. You might want to keep any information (wider reading) you have gathered that is relevant to your project.

Open the Week Two - workflow project on RStudio Cloud.

Within this project you will notice there is already one file *.Rproj*. This is an R project file, this is a very useful feature, it interacts with R to tell it you are working in a very specific place on the computer (in this case the cloud server we have dialed into). It means R will automatically treat the location of your project file as the 'working directory' and makes importing and exporting easier³.

³More on projects can be found in the R4DS book (<https://r4ds.had.co.nz/workflow.html>)

3.4.1 Organise

Now we are going to organise our workspace, first its always a good first step to go to Tools > Project options and switch all of the options for saving and loading .Rdata to ‘No’

Then we create the following folders:

- data
- scripts
- figures

Make sure you type these **exactly** as printed here - remember that to R ‘data’ and ‘Data’ are two different things!

Having these separate subfolders within our project helps keep things tidy, means it’s harder to lose things, and lets you easily tell R exactly where to go to retrieve data.

Now do you remember where you saved the ‘penguin_data.csv’ file? I hope so!!! Go to the upload button in the Files tab of RStudio Cloud and tell it where the file is located on your local computer and upload it to the ‘data’ folder you have made in your Project.

3.4.2 Create a new R script

Let’s now create a new R script file in which we will write instructions and store comments for manipulating data, developing tables and figures. Use the File > New Script menu item and select an R Script.

Add the following

```
# An analysis of the bill dimensions of male and female Adelie, Gentoo and Chinstrap penguins
# Data first published in Gorman, KB, TD Williams, and WR Fraser. 2014. "Ecological ...
```

Then load the following add-on package to the R script, just underneath these comments. Tidyverse isn’t actually one package, but a bundle of many different packages that play well together - for example it *includes* ggplot2 which we used in the last session, so we don’t have to call that separately

```
library(tidyverse)
library(janitor)
library(lubridate)
```

Now save this file in the scripts folder and call it *penguin_measurements.R*

projects.html)

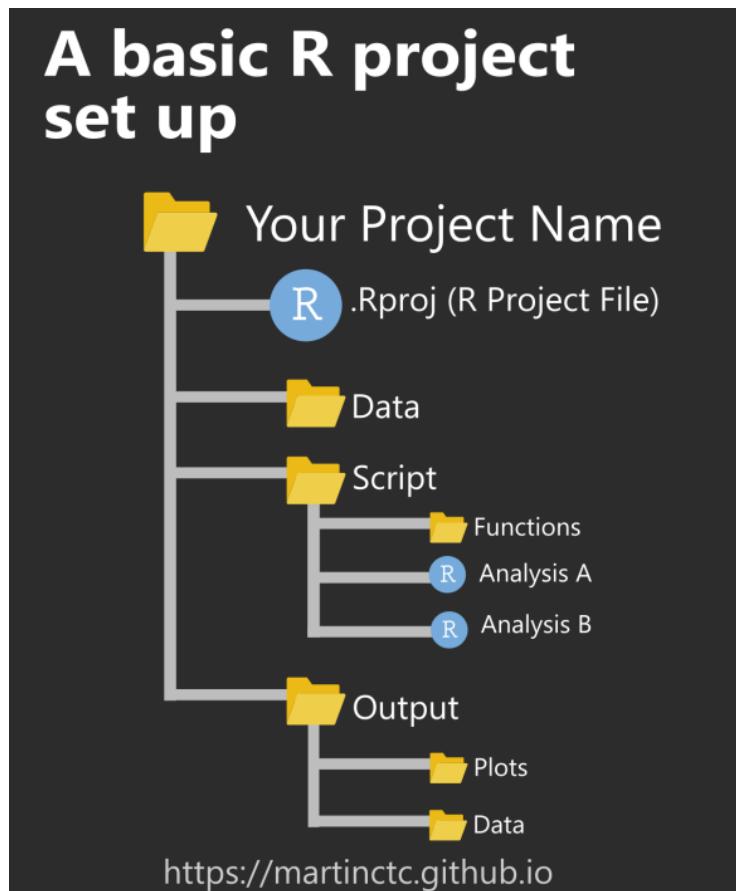


Figure 3.3: An example of a typical R project set-up

3.5 Get the data into R

Now we are *finally* ready to import the data into R.

Add the following code to your script, then starting at line 1 - ask your script to run in order, library function first as the `read_csv()` function is from an add-on package `readr`

```
# read in the data from the data folder in my project
penguins <- read_csv("data/penguins_data.csv")
```

Error!

Houston we have a problem! We got an error!!! blah blah does not exist in current working directory

This usually happens if we told R to look in the wrong place, or didn't give it the correct name of what to look for. Can you spot what the mistake was?

Edit your existing line of script to replace it with the proper file name and run the command again.

```
# read in the data from the data folder in my project
penguins <- read_csv("data/penguins_data.csv")
```

```
Parsed with column specification:
cols(
  studyName = col_character(),
  `Sample Number` = col_double(),
  Species = col_character(),
  Region = col_character(),
  Island = col_character(),
  Stage = col_character(),
  `Individual ID` = col_character(),
  `Clutch Completion` = col_character(),
  `Date Egg` = col_character(),
  `Culmen Length (mm)` = col_double(),
  `Culmen Depth (mm)` = col_double(),
  `Flipper Length (mm)` = col_double(),
  `Body Mass (g)` = col_character(),
  Sex = col_character(),
  `Delta 15 N (o/oo)` = col_double(),
  `Delta 13 C (o/oo)` = col_double(),
  Comments = col_character()
)
```

Great, no error this time, the `read_csv()` function has read the data and we <-

assigned this data to the object `penguins`. If you look in the environment pane you should see the object `penguins`.

The lines printed in the R console tell us the names of the columns that were identified in the file and the type of variable R thinks it is

- `col_character()` means the column contains letters or words
- `col_double()` means the column contains numbers

Have a look - do all of these seem correct to you? If not we should fix these, and we will get onto that in a little bit.

3.5.1 View and refine

So we know our data is in R, and we know the columns and names have been imported. But we still don't know whether all of our values imported, or whether it captured all the rows.

There are lots of different ways to view and check data. One useful method is `glimpse`

```
# check the structure of the data
glimpse(penguins)
```

When we run `glimpse()` we get several lines of output. The number of observations “rows”, the number of variables “columns”. Check this against the csv file you have - they should be the same. In the next lines we see variable names and the type of data.

We should be able to see by now, that all is not well!!! Body Mass (`g`) is being treated as character (string), as is Date Egg, meaning R thinks these contain letters and words instead of numbers and dates.

Other ways to view the data

- type `penguins` into the Console
- type `view(penguins)` this will open a spreadsheet in a new tab
- type `head(penguins)` will show the first 10 lines of the data, rather than the whole dataset

`view()` lets you do cool stuff like reorder rows and quickly scroll through the dataset without affecting the underlying data.

3.5.2 Data management

We've imported the data, checked it and found some inconsistencies.

This is where we start to use some of the core functions from `tidyverse`.

If R thinks Body Mass (g) is a character variable, then it probably contains some words as well as numbers. So let's have a look at this and compare it to a variable which has been processed correctly

```
# get the first 10 rows of the Flipper Length and Body Mass variables
penguins %>%
  select(`Flipper Length (mm)`,
         `Body Mass (g)`) %>%
  head(10) # 10 rows
```

A tibble: 10 × 2

	Flipper Length (mm)	Body Mass (g)
	<dbl>	<chr>
1	181	3750
2	186	3800
3	195	3250
4	NA	na
5	193	3450
6	190	3650
7	181	3625
8	195	4675
9	193	3475
10	190	4250

1-10 of 10 rows

What is the %>% ??? It's known as a pipe. It sends the results of one line of code to the next. So the data penguins is sent to the `select` function which picks only those variables we want. The result of this is then sent to the `head` function which with the argument for number of rows set to 10, prints the top 10 rows. The other way to write this series of functions would be to use brackets and the rules of BODMAS. `head(select(penguins, Flipper Length (mm), Body Mass (g)),10)` Hopefully you agree that the pipes make the code a lot more human-reader friendly. More on pipes later

So what's the problem with our data? in the Flipper length variable, missing observations have been correctly marked as NA signifying missing data. R can handle missing data just fine. However, in the Body mass variable, it looks as though someone has actually typed "na" in observations where the data is

missing. Here R has failed to recognise an NA and has read it as a word instead. This is because `read_csv()` looks for gaps or NA but not “na”.

No problem this is an easy fix. Head back to your script and make the following edit the line for data importing

```
# read in the penguins data, specify that NA strings can be "na" or "NA"
penguins <- read_csv("data/penguin_data.csv", na=c("na","NA"))
```

Now re-check your data using the same lines of code from before. All ok now?

3.6 Dataframes and tibbles

A quick sidebar on how R stores data. When we imported the data into R it was put into an object called a **tibble** which is a type of **dataframe**.

Let's have a quick go at making our own **tibble** from scratch.

Make a new script called ‘TibbleTrouble.R’ in the scripts folder as before.

At the top of this new script put

```
# just me messing about making tibbles

# libraries
library(tidyverse)

# make some variables, when we have a one dimensional object like this it is known as an atomic vector
person <- c("Mark", "Phil", "Becky", "Tony")
hobby <- c("kickboxing", "coding", "dog walking", "car boot sales")
awesomeness <- c(1,100,1,1)
```

The function `c` lets you ‘concatenate’ or link each of these arguments together into a single vector.

Now we put these vectors together, where they become the variables in a new tibble

```
# make a tibble
my_data <- tibble(person, hobby, awesomeness)
my_data

# A tibble: 4 x 3
  person    hobby      awesomeness
  <chr>   <chr>        <dbl>
1 Mark    kickboxing     1
```

```

2 Phil    coding          100
3 Becky   dog walking      1
4 Tony    car boot sales    1

```

Have a go at messing about with your script and figure out what each of these does, add comments and save your script.

```

# Some R functions for looking at tibbles and dataframes I will comment next to each one

head(my_data, n=2)
tail(my_data, n=1)
nrow(my_data)
ncol(my_data)
colnames(my_data)
view(my_data)
glimpse(my_data)
str(my_data)

```

3.7 Clean and tidy

Back to your penguins script.

We have checked the data imported correctly, now its time to *clean and tidy* the data.

3.7.1 Tidy

In this example our data is already in `tidy` format i.e. one observation per row. We will cover what to do if data isn't tidy later.

3.7.2 Clean

Here are a few things we might want to do to our data to make it 'clean'.

- Refine variable names
- Format dates and times
- Rename some values
- Check for any duplicate records
- Check for any implausible data or typos
- Check and deal with missing values

Remember because we are doing everything in a script, the original data remains unchanged. This means we have data integrity, and a

clear record of what we did to tidy and clean a dataset in order to produce summaries and data visuals

3.7.3 Refine variable names

Often we might want to change the names of our variables. They might be non-intuitive, or too long. Our data has a couple of issues:

- Some of the names contain spaces
- Some of the names contain brackets

Let's correct these quickly

```
#clean all variable names to snake_case using the clean_names function from the janitor package
# note we are using assign <- to overwrite the old version of penguins with a version that has up
# this changes the data in our R workspace but not the original csv file

penguins <- penguins %>% #
  janitor::clean_names()

colnames(penguins) # quickly check the new variable names
```

Note - in this example you can see that I put the name of the package `janitor`, in front of the function `clean_names`. But if I have already loaded the package with `library(janitor)` then this isn't strictly necessary and `penguins %>% clean_names()` would also have worked just fine. So why do this? Remember when we loaded the tidyverse package - we got a warning about conflicts - this was because two of our packages `dplyr` and `stats` have functions that are different, but share the same name. But we can make sure we call the one we want by specifying which package we are calling a function from. We don't need to do this very often, but it's good to know.

```
> library(tidyverse)
-- Attaching packages -----
v ggplot2 3.3.2     v purrr   0.3.4
v tibble   3.0.4     v dplyr    1.0.2
v tidyr    1.1.2     v stringr  1.4.0
v readr    1.3.1     vforcats  0.5.0
-- Conflicts -----
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

The `clean_names` function quickly converts all variable names into snake case. The N and C blood isotope ratio names are still quite long though, so let's clean those with `dplyr::rename()` where “new_name” = “old_name”.

```
# shorten the variable names for N and C isotope blood samples

penguins <- penguins %>%
  rename("delta_15n"="delta_15_n_o_oo", # use rename from the dplyr package
        "delta_13c"="delat_13_c_o_oo")
```

3.7.4 Dates

We can also see there is a `date_egg` variable. If you check it (using any of the new functions you have learned), you should see that it all looks like its been inputted correctly, but R is treating it as words, rather than assigning it a date value. We can fix that with the `lubridate` package. If we use the function `dmy` then we tell R this is date data in date/month/year format.

```
# use dmy from stringr package to encode date properly
penguins <- penguins %>%
  mutate(date_egg_proper=dmy(date_egg))
```

Here we use the `mutate` function from `dplyr` to create a new variable called `date_egg_proper` based on the output of converting the characters in `date_egg` to date format. The original variable is left intact, if we had specified the “new” variable was also called `date_egg` then it would have overwritten the original variable.

3.7.5 Rename some values

Sometimes we may want to rename the values in our variables in order to make a shorthand that is easier to follow.

```
# use mutate and case_when for an if-else statement that changes the names of the values
penguins <- penguins %>%
  mutate(species = case_when(species == "Adelie Penguin (Pygoscelis adeliae)" ~ "Adelie",
                             species == "Gentoo penguin (Pygoscelis papua)" ~ "Gentoo",
                             species == "Chinstrap penguin (Pygoscelis antarctica)" ~ "Chinstrap"))
```

3.7.6 Check for duplication

It is very easy when inputting data to make mistakes, copy something in twice for example, or if someone did a lot of copy-pasting to assemble a spreadsheet (yikes!). We can check this pretty quickly

```
# check for duplicate rows in the data
penguins %>%
  duplicated() %>% # produces a list of TRUE/FALSE statements for duplicated or not
  sum() # sums all the TRUE statements
```

[1] 0

Great!

3.7.7 Check for typos or implausible values

We can also explore our data for very obvious typos by checking for implausibly small or large values

```
# use summarise to make calculations
penguins %>%
  summarise(min=min(body_mass_g, na.rm=TRUE),
            max=max(body_mass_g, na.rm=TRUE))
```

The minimum weight for our penguins is 2.7kg, and the max is 6.3kg - not outrageous. If the min had come out at 27g we might have been suspicious. We will use `summarise` again to calculate other metrics in the future.

Note - our first data insight, the difference the smallest adult penguin in our dataset is nearly half the size of the largest penguin.

We can also look for typos by asking R to produce all of the distinct values in a variable. This is more useful for categorical data, where we expect there to be only a few distinct categories

```
penguins %>%
  distinct(sex)
```

Here if someone had mistyped e.g. ‘FMALE’ it would be obvious. We could do the same thing (and probably should have before we changed the names) for species.

3.7.8 Missing values - NA

There are multiple ways to check for missing values in our data

```
# Get a sum of how many observations are missing in our dataframe
penguins %>%
  is.na() %>%
  sum()
```

But this doesn't tell us where these are, fortunately the function `summary` does this easily

```
# produce a summary of our data
summary(penguins)
```

This provides a quick breakdown of the max and min for all numeric variables, as well as a list of how many missing observations there are for each one. As we can see there appear to be two missing observations for measurements in body mass, bill lengths, flipper lengths and several more for blood measures. We don't know for sure without inspecting our data further, *but* it is likely that the two birds are missing multiple measurements, and that several more were measured but didn't have their blood drawn.

We will leave the NA's alone for now, but it's useful to know how many we have.

We've now got a clean & tidy dataset!!

3.8 Summing up

3.8.1 What we learned

There was a lot of preparation here, and we haven't really got close to make any major insights. But you have:

- Organised your project space
- Dealt with file formats
- Put data into a specific location and imported into R
- Checked the data import
- Cleaned and tidied the data

You have also been introduced to the `tidyverse` and two of its packages

- `readr` Wickham et al. (2018)
- `dplyr` Wickham et al. (2020b)

As well as:

- `janitor` Firke (2021)
- `lubridate` Spinu et al. (2020)

3.8.2 Quitting

- Remember to save your RScript before you leave. And ideally don't save your .Rdata either!

- Close your RStudio Cloud Browser
- Go to Blackboard to complete this week's quiz!

*R Cheat Sheets

Bibliography

- Firke, S. (2021). *janitor: Simple Tools for Examining and Cleaning Dirty Data*. R package version 2.1.0.
- Gorman, K., Williams, T., and Fraser, W. (2014). Ecological sexual dimorphism and environmental variability within a community of antarctic penguins (genus *pygoscelis*). *PLoS One*, 9(3):e90081.
- Hadley Wickham, G. G. (2020). *R for Data Science*. Chapman and Hall/CRC, Boca Raton, Florida.
- Horst, A., Hill, A., and Gorman, K. (2020). *palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data*. R package version 0.1.0.
- Spinu, V., Golemund, G., and Wickham, H. (2020). *lubridate: Make Dealing with Dates a Little Easier*. R package version 1.7.9.
- Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Golemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., Takahashi, K., Vaughan, D., Wilke, C., Woo, K., and Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686.
- Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., Yutani, H., and Dunnington, D. (2020a). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.3.2.
- Wickham, H., François, R., Henry, L., and Müller, K. (2020b). *dplyr: A Grammar of Data Manipulation*. R package version 1.0.2.
- Wickham, H., Hester, J., and Francois, R. (2018). *readr: Read Rectangular Text Data*. R package version 1.3.1.