Data Science for Biologists - 5023Y

Philip Leftwich

2022-01-28

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

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 assigments 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 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 "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 quesitons 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 their 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 its 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 impossilbe 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)
```

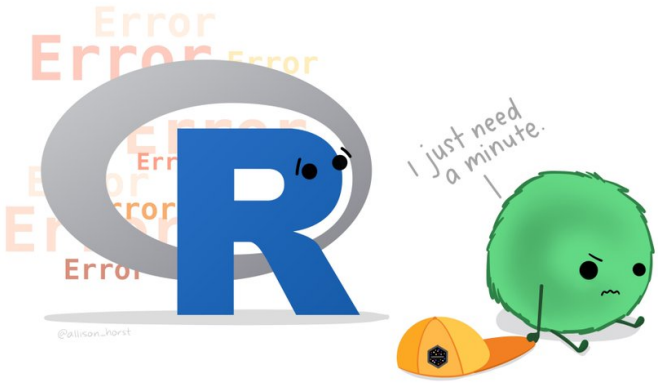


Figure 2.1: courtesy of Allison Horst

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 commmand 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 funcitons 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. (2021b)) 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.

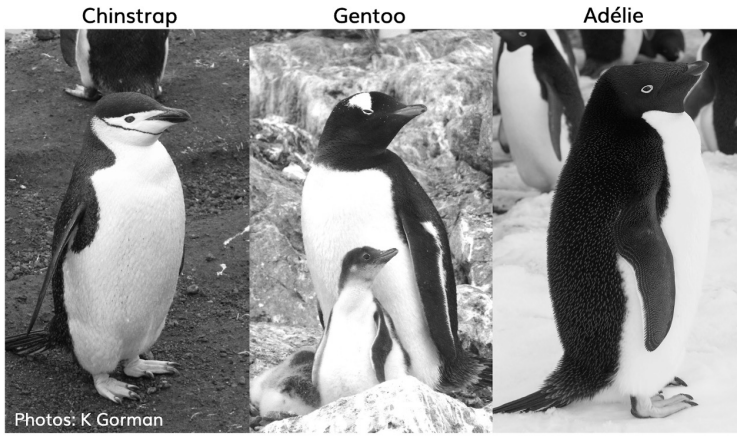
We are going to be working as though we are in the latter stages of a research project, where data has been collected, possibly over several years, to test against our hypotheses.

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

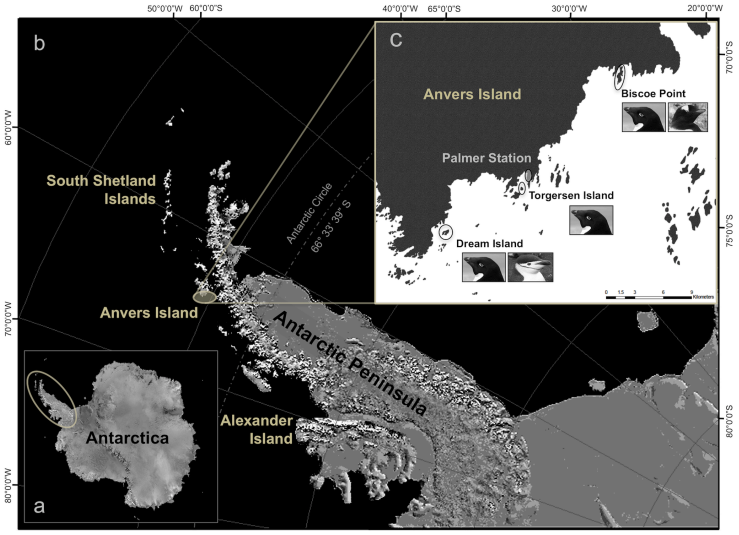
This data, taken from the `palmerpenguins` (Horst et al. (2020)) package was originally published by Gorman et al. (2014).

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 inshort and are deep-divers. We might expect that we can find some features of Gentoo's that align with their different lifestyle/feeding habits.

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 expecations of the magnitude of the difference. This would come from doing more research on the subject area.

- The bill lengths/depths ratio to body size of Gentoo penguins will be different to Adelie and Chinstrap penguins.
- Gentoo penguins will have a different N and carbon isotope ratio than Adelie and Chinstrap penguins.

3.3 Preparing the data

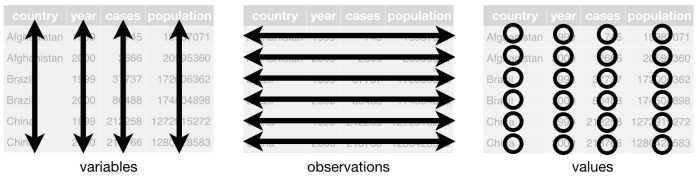
Imagine we have completed our practical study and have our data. The data is probably distributed in lots of places, originally notes collected in the field were probably on paper & notebooks. Then someone will have taken time to transcribe those into a spreadsheet. This will almost certainly have been 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* format[1]. 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*. Obe 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 tpye 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 enough [2]. 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

	studyNam	Sample Nu	Species	Region	Island	Stage	Individual	Clutch Cor	Date Egg
2	PAL0708	1	Adelie Per	Anvers	Torgersen	Adult, 1 Eg	N1A1	Yes	11/11/:
3	PAL0708	2	Adelie Per	Anvers	Torgersen	Adult, 1 Eg	N1A2	Yes	11/11/:
4	PAL0708	3	Adelie Per	Anvers	Torgersen	Adult, 1 Eg	N2A1	Yes	16/11/:
5	PAL0708	4	Adelie Per	Anvers	Torgersen	Adult, 1 Eg	N2A2	Yes	16/11/:
6	PAL0708	5	Adelie Per	Anvers	Torgersen	Adult, 1 Eg	N3A1	Yes	16/11/:
7	PAL0708	6	Adelie Per	Anvers	Torgersen	Adult, 1 Eg	N3A2	Yes	16/11/:
8	PAL0708	7	Adelie Per	Anvers	Torgersen	Adult, 1 Eg	N4A1	No	15/11/:
9	PAL0708	8	Adelie Per	Anvers	Torgersen	Adult, 1 Eg	N4A2	No	15/11/:
10	PAL0708	9	Adelie Per	Anvers	Torgersen	Adult, 1 Eg	N5A1	Yes	09/11/:
11	PAL0708	10	Adelie Per	Anvers	Torgersen	Adult, 1 Eg	N5A2	Yes	09/11/:
12	PAL0708	11	Adelie Per	Anvers	Torgersen	Adult, 1 Eg	N6A1	Yes	09/11/:

```
studyName,Sample Number,Species,Region,Island,Stage,Individual ID,Clutch Comp
PAL0708,1,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
PAL0708,2,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
PAL0708,3,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
PAL0708,4,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
PAL0708,5,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
PAL0708,6,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
PAL0708,7,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
PAL0708,8,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
PAL0708,9,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
PAL0708,10,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
PAL0708,11,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
PAL0708,12,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
PAL0708,13,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
PAL0708,14,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
PAL0708,15,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
PAL0708,16,Adelie Penguin (Pygoscelis adeliae),Anvers,Torgersen,"Adult, 1 Egg
```

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).

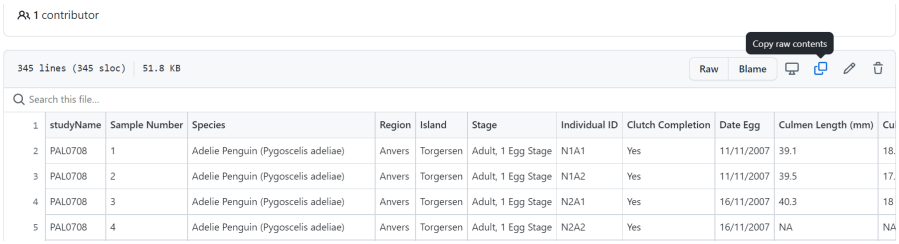


Figure 3.2: Click on the copy raw contents button

We then need to

1. Select the copy raw contents button
2. Open a blank notepad
3. Paste the contents
4. Save the file as 'penguin_data.csv'
5. 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 easier[3].

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 is case-sensitive so '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 p

# Data first published in  Gorman, KB, TD Williams, and WR Fraser. 2014. "Ecological S
```

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

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

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

```
library(tidyverse) # tidy data packages
library(janitor) # cleaning variable names
library(lubridate) # make sure dates are processed properly
```

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

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

  **Note - `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 x 2

Flipper Length (mm) <dbl>	Body Mass (g) <chr>
181	3750
186	3800
195	3250
NA	na
193	3450
190	3650
181	3625
195	4675
193	3475
190	4250

1-10 of 10 rows

**Note - 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 thew 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 ws 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 v
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 o

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

  **Note - 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 --------------------------------------- tidyverse 1.3.0 --
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     v forcats 0.5.0
-- Conflicts ------------------------------------------ tidyverse_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))
```

What is the deliberate mistake in my code comment above? By now you may have picked up there are the odd mistakes (possibly some non-deliberate ones) - to make sure you aren't just copy-pasting on autpilot.

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 in a va
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. (2021d)
- `dplyr` Wickham et al. (2021c)

As well as:

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

3.8.2 Quitting

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

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

Chapter 4

Workflow Part Two: Week Three

Last week we worked through the journey of importing and tidying data to produce a clean dataset.

It's important to remember what questions you have about the data collected, and to make an outline about what you want to do.

Now is a good time to think about what figures might we want to produce from our data?

We are mostly interested in observable 'differences' between our three penguin species. What sort of figures might illustrate that?

Sometimes it's good to get a pen/pencil and paper - and sketch the figure you might want to make.

4.1 Initial insights

Let's start with some basic insights, perhaps by focusing on further questions about specific variables.

- How many penguins were observed?
- How many Adelie, Gentoo and Chinstrap penguins
- What is the distribution of morphologies such as bill length, body size, flipper length

Some of these are very simple in that they are summaries of single variables. Some are more complex, like evaluating the numbers of males and females which

are the two groups in the sex variable.

These are 'safety-checking' insights. You might already know the answers to some of these questions because you may have been responsible for collecting the data. Checking your answers against what you expect is a good way to check your data has been cleaned properly.

4.2 Numbers and sex of the penguins

```
# how many observations of penguins were made

penguins %>%
  summarise(n())

# but there are multiple observations of penguins across different years
# n_distinct deals with this

penguins %>%
  summarise(n_distinct(individual_id))
```

The answer we get is that there are 190 different penguins observed across our multi-year study.

This answer is provided in a tibble, but the variable name is an ugly composition of the functions applied, but we can modify the code

```
penguins %>%
  summarise(num_penguin_id=n_distinct(individual_id))
```

How about the number of penguins observed from each species?

```
penguins %>%
  group_by(species) %>% # ask for distinct counts in each species
  summarise(num_penguin_id=n_distinct(individual_id))
```

By adding the `group_by` function we tell R to apply any subsequent functions separately according to the group specified. Let's do this again for male and female penguins

```
penguins %>%
  group_by(sex) %>% # ask for distinct counts in each species
  summarise(num_penguin_id=n_distinct(individual_id))
```

Now what about the number of female Gentoo penguins?

```
penguins %>%
  group_by(sex, species) %>% # ask for distinct counts in each species
  summarise(num_penguin_id=n_distinct(individual_id))
```

Now we have a table that shows each combination of penguin species by sex, we can see for example that 65 unique Male Adelie penguins were observed in our study. We can also see that for 6 Adelie and 5 Gentoo penguins, sex was not recorded.

> **Note - You have just had a crash course in using the `pipe` and `dplyr` to produce quick data summaries. Have a go at making some other summaries of your data, perhaps the numbers of penguins by island or region. Try some combinations.

4.2.1 Making a simple figure

Let's translate some of our simple summaries into graphs.

```
# make summary data and assign to an object with a sensible name we can use
penguin_species_sex <- penguins %>%
  group_by(sex, species) %>% # ask for distinct counts in each species
  summarise(num_penguin_id=n_distinct(individual_id)) %>%
  drop_na() # remove the missing data
```

```
# basic ggplot function to make a stacked barplot
penguin_species_sex %>%
  ggplot(aes(x=species,
             y=num_penguin_id,
             fill=sex))+
  geom_col()
```

We will cover how ggplot works in more detail next week. But in brief, to get the graph we first use the `ggplot` function, we give `ggplot` the 'aesthetic mappings' of the plot, the values we wish to assign to the x axis, y axis and how we want to "fill in" the objects we will draw.

This first line of code will just draw a blank plot, with the *plus* + sign we signify that we want to add a new layer, this builds on top of the first layer and by specifying the function `geom_col` we request columns are layered onto the plot. This layer inherits all of the specifications of x,y and fill from `ggplot()`. And it produces a handy legend.

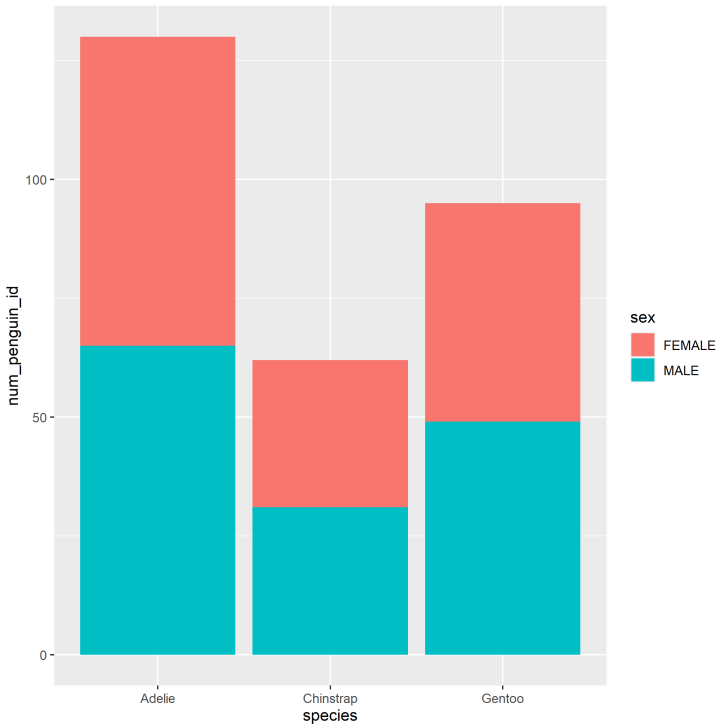


Figure 4.1: A first insight the number of male and female penguins of each species in our study

4.2.2 Challenge

Can you write and run a script, with appropriate comments, that produces a summary tibble and graph for the number of penguins of each species, on the three study islands?

4.3 Distributions

We now know how many penguins were surveyed. Let's move on to look at some distributions. One of our interests was the size of bill lengths, so let's look at the distribution of values in this variable.

Looking at frequency distributions is very *useful* because it shows the shape of the *sample distribution*, that shape is very important for the types of formal statistics we can do later.

Here is the script to plot frequency distribution, as before we pipe the data into ggplot. This time we only specify an x variable because we intend to plot a histogram and the y variable is always the count of observations. We then ask for the data to be presented in 50 equally sized bins of data. So in this case we have chopped the range of the x axis into 50 equal parts and counted the number of observations that fall within each one.

```
penguins %>%
  ggplot(aes(x=culmen_length_mm))+
  geom_histogram(bins=50)
```

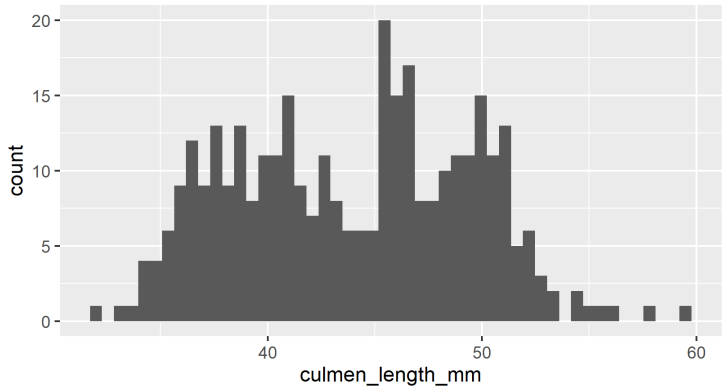


Figure 4.2: Frequency distribution of culmen length in penguins

** Note - Bins. Have a go at changing the value specified to the bins argument, and observe how the figure changes.

4.3.1 Insights

- Are you surprised at all by the distribution? We have drawn a continuous variable from a natural population, what did you expect the distribution to look like?
- Can you change the code for the histogram plot to produce distributions for each sex?
- How has this changed your interpretation of the distributions?

4.4 More distributions

From the figures you have made, you should be able to make some guesses about the means and medians of the data. But we can use `dplyr` to get more accurate answers.

```
# generate the mean, median and standard deviation of culmen length in three species o

penguins %>%
  group_by(species) %>%
  summarise(mean_culmen_length=mean(culmen_length_mm),
            median_culmen_length=median(culmen_length_mm),
            sd_culmen_length=sd(culmen_length_mm))
```

Huh??? What's going on??? We get NAs because there are NAs in our dataset any calculation involving nothing produces nothing. Try this

```
# messing about with NA calculations
4+NA

3*NA

NA^2

(1+2+NA)/2
```

We need to remember to specify the argument to remove `NA`

```
# generate the mean, median and standard deviation of culmen length in three species o
# specify the removal of NA values
penguins %>%
  group_by(species)
```

```
  summarise(mean_culmen_length=mean(culmen_length_mm, na.rm=TRUE),
            median_culmen_length=median(culmen_length_mm, na.rm=TRUE),
            sd_culmen_length=sd(culmen_length_mm, na.rm=TRUE))
```

The mean and median values for each species are *very* similar, which indicates we do not have much *skew* in our data. This detail is important because statistical analyses make lots of assumptions about the underlying distributions of the data.

4.4.1 Initial conclusions

In these data we are already able to make some useful insights

- Fewer Chinstrap penguins were surveyed than Adelie or Gentoo's
- The average bill length for Adelie penguins is 38.8 mm, on average 8.7mm shorter than Gentoo's and 10mm shorter than Chinstrap's
- There does not appear to be much difference between Chinstrap and Gentoo bill lengths (on average 1.3mm)

These might appear to be modest insights - but have learned several data manipulation and summary techniques. We can also start to take a look at some of our initial hypotheses!!!

– How does this data stack up against the hypothesis about bill morphology we put forward last week?

To make more and conclusive insights we have a bit further to go, but I think you deserve a pat on the back

```
# R generate some praise
praise::praise()
```

4.5 Data transformation

In the previous section we made some basic insights into observation numbers, distributed by species, sex and location. We also started to gain core insights into some of our central hypotheses, but you have probably noticed we don't actually have a variable on the `relative bill lengths/depths`. Why is this important? Well we clearly saw there was a difference in bill lengths between our three species. But we haven't taken into account that some of these species might be very different in body size. Our measure of bill lengths as an indicator of feeding strategy, might be confounded by body size (a bigger penguin is likely to have a bigger bill).

We don't have a variable explicitly called body size. Instead we have to use "proxies" suitable proxies might be 'flipper length' or 'body mass'. Neither is perfect

- Flipper length
  - Pros: linked to skeletal structure, constant
  - Cons: relative flipper length could also vary by species
- Body mass
  - Pros: more of an indication of central size?
  - Cons: condition dependent, likely to change over the year

Let's take a look at the distribution of body mass among our three species.

```
# frequency distribution of body mass by species
penguins %>%
  ggplot(aes(x=body_mass_g, fill=species))+
  geom_histogram(bins=50)
```

How does the distribution you have found compare with your insights on bill length? We can do this using the `cor_test` function from the package `rstatix` (Kassambara (2020)). This package contains a number of 'pipe-friendly' simple statistics functions

Add the `library()` for `rstatix` at the **top** of your RScript with an appropriate comment #

```
# a simple correlation test from the rstatix package
penguins %>%
  group_by(species) %>% # group by species
  cor_test(body_mass_g, culmen_length_mm) # correlation between body mass and bill leng
```

We can see that these two variables 'co-vary' a lot, but this appears to be quite species specific. We can already make the insight that Chinstrap penguins appear to have the shortest bill length relative to body mass.

We can make a new variable that is the 'relative size of culmen length compared to flipper length'

```
# use mutate to produce a new variable that is a ratio of culmen length to flipper leng
penguins <- penguins %>%
  mutate(relative_bill_length = culmen_length_mm/body_mass_g)
```

We are probably also interested in bill depth?

```
# use mutate to produce a new variable that is a ratio of culmen depth to flipper length
penguins <- penguins %>%
  mutate(relative_bill_depth = culmen_depth_mm/body_mass_g)

# check that these new variables have been included in the dataset
penguins %>%
  names()
```

4.6 Developing insights

First let's focus on the distributions of four variables of interest. Then we will progress on to look at their relationships and differences

- relative_bill_length
- relative_bill_depth
- delta_15n
- delta_13c

4.6.1 Distributions of the relative bill length

We will examine the shape of the *sample distribution* of the data again by using histograms.

```
# frequency distribution of relative bill length by species
# we already know we are interested in looking at the distributions 'within' each species
penguins %>%
  ggplot(aes(x=relative_bill_length, fill=species))+
  geom_histogram(bins=50)
```

Important questions, what shape is the distribution?

- First there must be a lower limit of zero (penguins cannot have negative length bills), does this lead to any truncating of the expected normal distribution bell curve? It doesn't look it.
- Is it symmetrical? Mostly.

If it's a little difficult to see - we can separate out these figures using the handy function `facet_wrap`

```
# frequency distribution of relative bill length by species
# we already know we are interested in looking at the distributions 'within' each species
penguins %>%
```

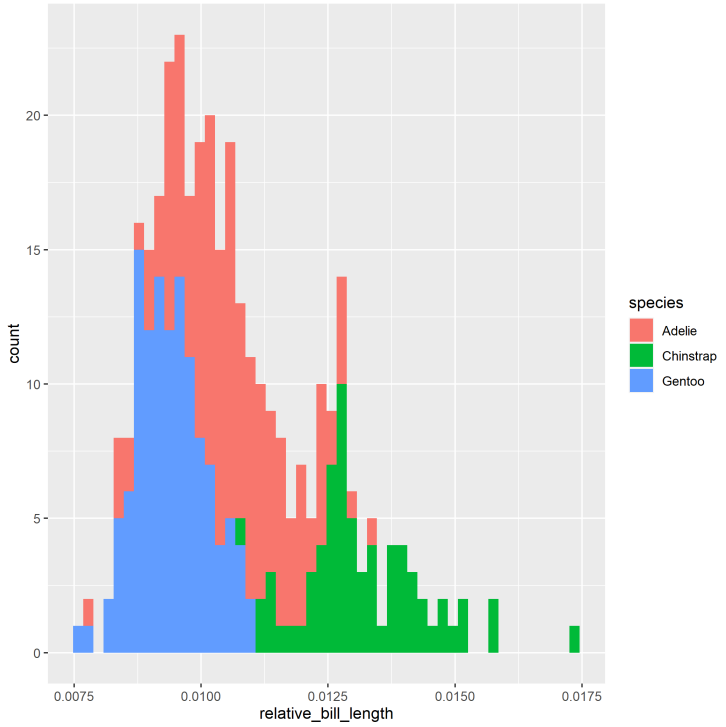
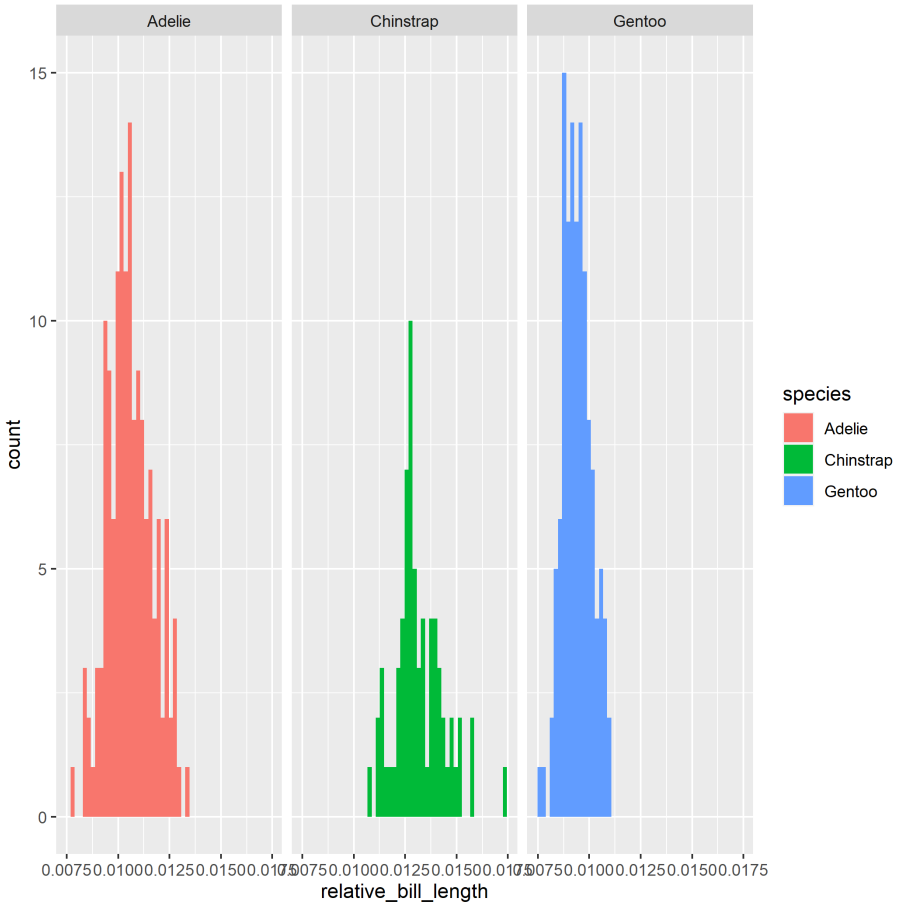


Figure 4.3: Frequency distribution of relative bill length in three species of penguins, Adelie, Chinstrap and Gentoo

```
ggplot(aes(x=relative_bill_length, fill=species))+
geom_histogram(bins=50)+
facet_wrap(~species) # facet wrap to look at the separate species more easily
```



\begin{figure}
\caption{Frequency distribution of relative bill length in three species of penguins, Adelie, Chinstrap and Gentoo - histograms split into three panes by facet_wrap} \end{figure}

Looking at these distributions, how do you think the mean & median values will compare in these three species? Think about it first - then run the code below.

```
# Mean and Median summaries
penguins %>%
```

```
  group_by(species) %>%
  summarise(mean_relative_bill_length=mean(relative_bill_length, na.rm=TRUE),
            median_relative_bill_length=median(relative_bill_length, na.rm=TRUE))
```

```
# A tibble: 3 x 3
  species   mean_relative_bill_length median_relative_bill_length
  <chr>                         <dbl>                       <dbl>
1 Adelie                      0.0106                      0.0105
2 Chinstrap                   0.0132                      0.0129
3 Gentoo                      0.00941                     0.00939
```

We can see that the mean and median values are almost identical for each species. This indicates we *aren't* dealing with a lot of skew, this is important for when using statistics, which are based on a lot of assumptions like normal distribution.

Can you **repeat** these steps for the variable `relative_bill_depth`, `delta_15n` and `delta_13c` - add all appropriate comments and commands to your R Script.

4.7 Relationship/differences

Getting proper data insights involves looking for relationships or differences. Remember, if we have a manipulated variable in a well-designed experiment, we may be able to identify a causal effect. With a study without this manipulation, like this penguin study - we cannot be sure any relationships or differences are causal. We have to include some caution in our interpretations.

4.7.1 Differences

We have already looked at frequency distributions of the data, where it is possible to see differences. However we can use ggplot and `geom_point` to produce difference plots.

```
# specifying position with a jitter argument positions points randomly across the x ax
penguins %>%
    ggplot(aes(x=species,
               y=culmen_length_mm,
               colour=species))+ # some geoms use color rather than fill to specify co
    geom_point(position=position_jitter(width=0.2))
```

**Note - try altering the width argument and see how it affects the output of the plot. We will do a deeper dive into ggplot later.

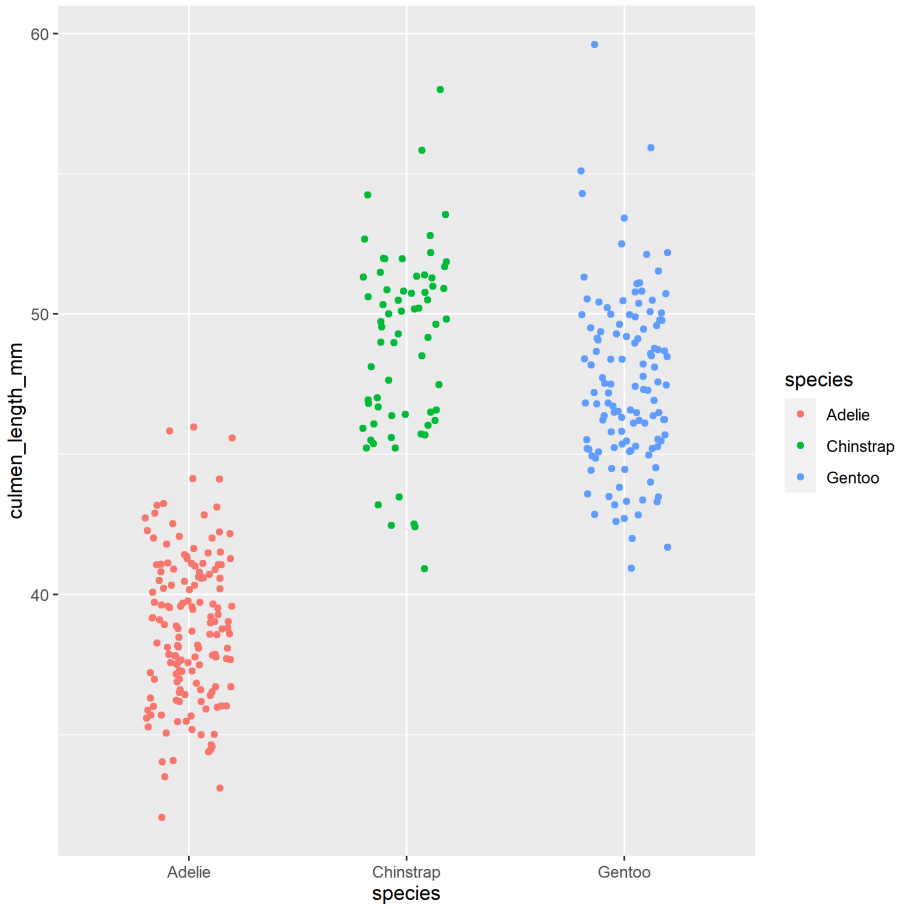


Figure 4.4: Differences in relative bill length of three different species of Antarctic Penguin.

Try producing these figures for the `culmen_depth_mm`, `delta_15n` and `delta_13c` variables as well.

What are you observations of the relative differences?

4.7.2 Associations

It might also be of interest to look at whether any of our variables of interest are strongly associated. For example what is the relationship between heavy carbon/heavy nitrogen isotopes?

```
penguins %>%
    ggplot(aes(x=delta_15n, y=delta_13c, colour=species))+
    geom_point()+
    geom_smooth(method="lm", # produces a simple regression line
                se=FALSE)    # no standard error intervals
```

We can see here that the association between these isotopes varies a lot by different species, there is probably quite a complex relationship here. It also indicates we should probably investigate these two isotopes separately.

- Have a look at the relative bill length and depth relationships as well

4.8 Sex interactions

Now let's concern ourselves with interactions. We have *already* seen how our interpretation of certain variables can be heavily altered if we don't take into account important contexts - like morphology in relation to species.

Thinking about sensible interactions takes patience and good biological understanding, the payoff is it can produce unique insights.

Focusing on relative bill length, we have seen there are differences between species, however we have not considered the potential association of sex. In many species males and females are 'dimorphic' and this has the potential to influence our observations if:

- sex has a substantial/bigger effect on morphology than species
- uneven numbers of males/females were scored in our studies

Using `summarise` `group_by` and `n_distinct` you should quickly be able to check the numbers of males and females surveyed within each species.

```
# A tibble: 6 x 3
# Groups:   species [3]
  species   sex    num_penguin_id
```

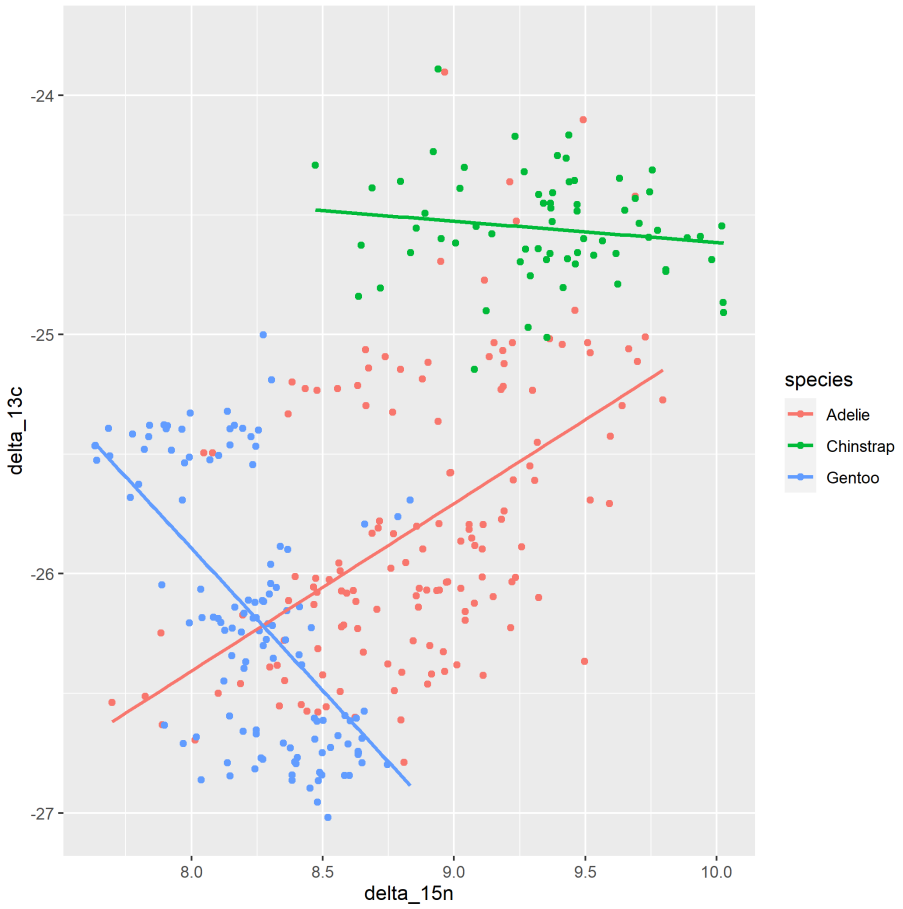


Figure 4.5: Asssocation between heavy Nitrogen and heavy Carbon isotopes in blood samples from three Antarctic Penguin species

```
  <chr>     <chr>           <int>
1 Adelie    FEMALE             65
2 Adelie    MALE               65
3 Chinstrap FEMALE             31
4 Chinstrap MALE               31
5 Gentoo    FEMALE             46
6 Gentoo    MALE               49
```

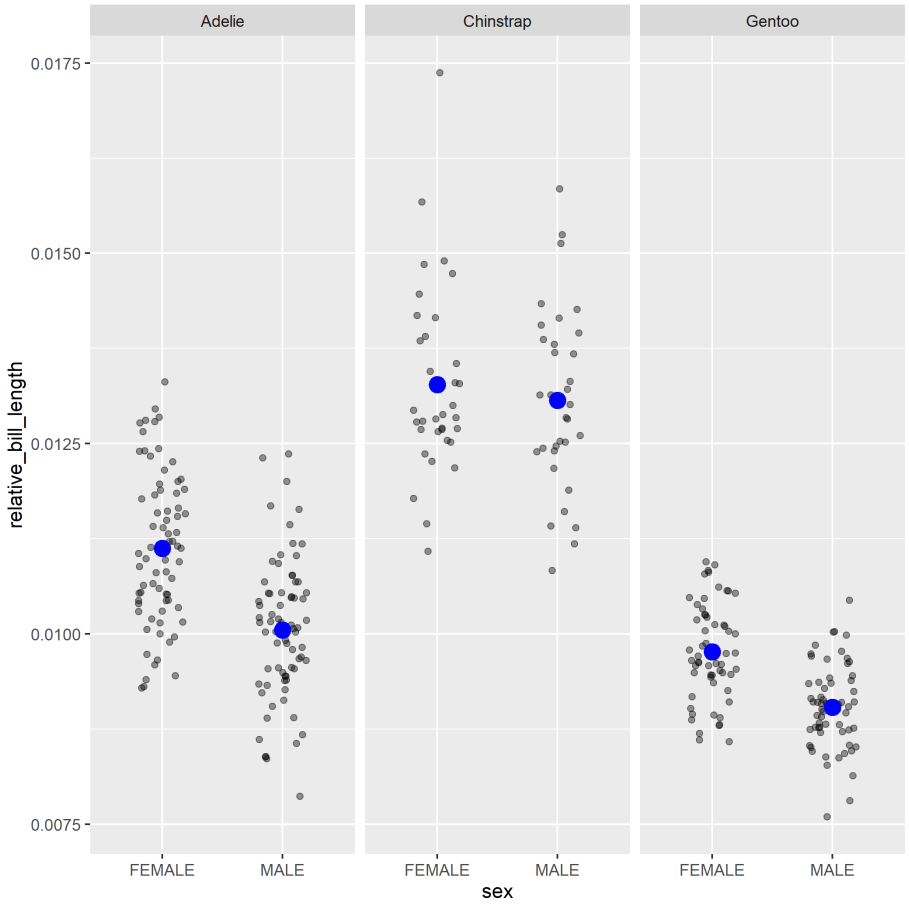
Looks like numbers are even, which is good. Again focusing on relative bill length let's generate some figures breaking down this variable by sex and species.

First let's generate some more simple summary stats - this time we are assigning it to an object to use later

```
penguin_stats <- penguins %>%
  group_by(sex, species) %>%
  summarise(mean_relative_bill_length=mean(relative_bill_length, na.rm=TRUE))
```

Now we are going to make our first figure which includes both raw and summary data. Copy and run the code below, and see if you can add comments next to the arguments you are unfamiliar with about what they might be doing.

```
penguins %>%
  drop_na(sex) %>%
  ggplot(aes(x=sex,
             y=relative_bill_length))+
  geom_jitter(position=position_jitter(width=0.2),
              alpha=0.4)+
  geom_point(data=penguin_stats, aes(x=sex,
                                     y=mean_relative_bill_length),
             size=4,
             color="blue")+
  facet_wrap(~species)
```



> Note - we could be producing something much simpler, like a box and whisker plot. It is often better to plot the data points rather than summaries of them

Imagine a line connects the two blue dots in each facet, these blue dots are the mean values. The slope of the line (if we drew them) would be downwards from females to males, indicating that on average females are larger. But the slop would not be very large. If we drew slope between *each* of the three female averages these would be much steeper. So we can say that size 'on average' varies more *between* species than *between* sexes.

See if you can make figures for all four of our variables of interest, then compare them to our initial hypotheses.

At this stage we do not have enough evidence to formally "reject" any of our null hypotheses, however we can describe the trends which appear to be present.

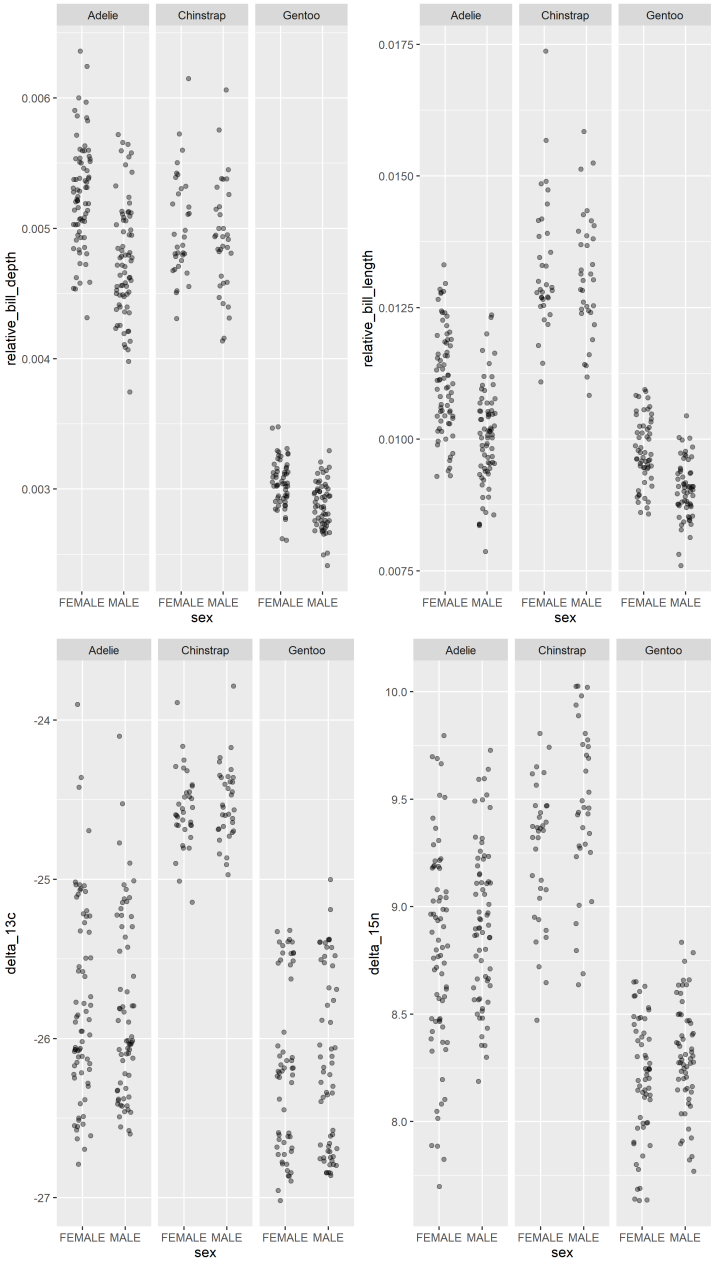


Figure 4.6: Difference in relative bill length and depth, and heavy carbon and nitrogen ratios, in male and female penguins from three different species - Adelie, Chinstrap and Gentoo

4.9 Making our graphs more attractive

We will dive into making attractive ggplots in more detail later. But let's spend a little time now fixing some of the more serious issues with our figures.

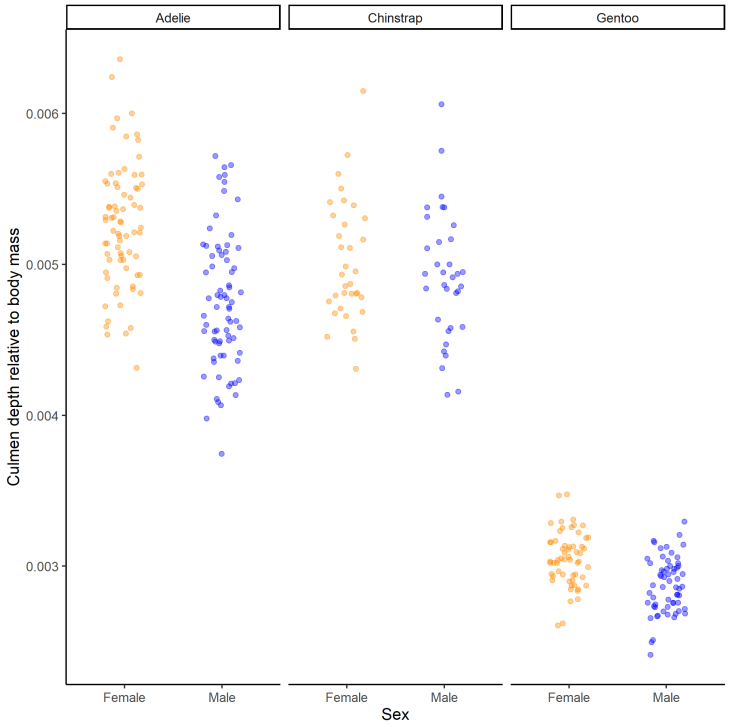
The aim with making a good figure, is that it is:

- Accurate - the figure must present the data properly, and not distort or mislead
- Beautiful - attractive figures will invite people to spend more time studying them
- Clear - a good figure should be able to stand on its own - see the answer or insight without prompting from the text

There are lots of things we could change here, but we will stick with the basics for now. We will:

- Change the axis labels
- Add some colours
- Remove the grey background

```
penguins %>%
    drop_na(sex) %>%
    ggplot(aes(x=sex,
               y=relative_bill_depth,
               colour=sex))+
    geom_jitter(position=position_jitter(width=0.2),
                alpha=0.4)+
  scale_color_manual(values=c("darkorange", "blue"))+
        facet_wrap(~species)+
  ylab("Culmen depth relative to body mass")+
  xlab("Sex")+
  scale_x_discrete(labels=c("Female", "Male"))+
  theme_classic()+
  theme(legend.position="none")
```



4.9.1 Saving your output

Now we have made a plot, that perhaps we think is worth saving. We can use the `ggsave` function. Just like when we imported data, when we output we specify a relative file path, this time we are saying where we want to 'send' the output. At the start of our project set-up we made a folder for figures to end up.

```
# save the last figure to a .png file in the figures folder

ggsave("figures/Jitter plot of relative culmen depth.png",
       dpi=300, # resolution
       width=7, # width in inches
       height=7)
```

Be careful here, if you ran this command again it would overwrite your previous file with a new output.

Check your Files tab - your image file should be saved in the appropriate folder

4.10 Quitting

Make sure you have saved your script!

Complete this week's Blackboard Quiz!

Chapter 5

ggplot2 A grammar of graphics: Week Four

5.1 Intro to grammar

The ggplot2 package is widely used and valued for its simple, consistent approach to making plots.

The 'grammar' of graphics relates to the different components of a plot that function like different parts of linguistic grammar. For example, all plots require axes, so the x and y axes form one part of the 'language' of a plot. Similarly, all plots have data represented between the axes, often as points, lines or bars. The visual way that the data is represented forms another component of the grammar of graphics. Furthermore, the colour, shape or size of points and lines can be used to encode additional information in the plot. This information is usually clarified in a key, or legend, which can also be considered part of this 'grammar'.

The most common components of a ggplot are:

- aesthetics
- geometric representations
- facets
- coordinate space
- coordinate labels
- plot theme

We will cover each below.

The philosophy of ggplot is much better explained by the package author, Hadley Wickham (Wickham et al. (2021b)). For now, we just need to be aware that ggplots are constructed by specifying the different components that we want to display, based on underlying information in a data frame.

5.2 Building a plot

We are going to use the *simple* penguin data set contained in the `palmerpenguins` package (Horst et al. (2020)).

Make sure you have a folder structure for your project - scripts - data (won't be needed this week) - figures

Make sure your new script is in the scripts folder!

Throughout this practical you will be introduced to new R packages. Please remember to include all the library() calls at the TOP of your script.

And if you need to install a package, the command is 'install.packages("")' - but DON'T include this in your script

```
library(palmerpenguins)
```

Let's check the first 6 rows of information contained in the data frame, using the head() function:

```
head(penguins)
```

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

Here, we aim to produce a scatter plot

5.3 Plot background

To start building the plot, we first specify the data frame that contains the relevant data. We can do this in two ways

1) Here we are 'sending the penguins data set into the ggplot function':

```
# render the plot background
penguins %>%
  ggplot()
```

2) Here we are specifying the dataframe *within* the `ggplot()` function

```
ggplot(data=penguins)
```

The output is identical

> **Note - Running this command will produce an empty grey panel. This is because we need to specify how different columns of the data frame should be represented in the plot.

5.4 Aesthetics - aes()

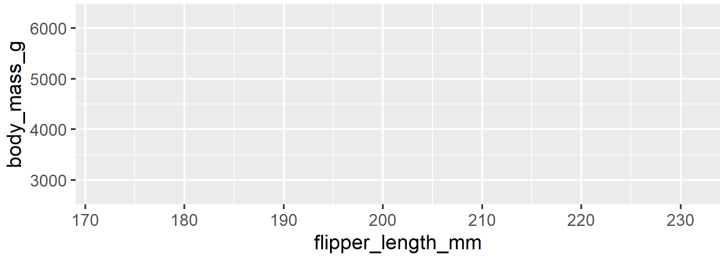
We can call in different columns of data from any dataset based on their column names. Column names are given as 'aesthetic' elements to the ggplot function, and are wrapped in the aes() function.

Because we want a scatter plot, each point will have an x and a y coordinate. We want the x axis to represent flipper length (x = flipper_length_mm), and the y axis to represent the body mass (y = body_mass_g).

We give these specifications separated by a comma. Quotes are not required when giving variables within aes().

> **Note - Those interested in why quotes aren't required can read about [non-standard evaluation] (https://edwinth.github.io/blog/nse/).

```
penguins %>%
  ggplot(aes(x=flipper_length_mm,
             y = body_mass_g))
```



So far we have the grid lines for our x and y axis. ggplot knows the variables required for the plot, and thus the scale, but has no information about how to display the data points.

5.5 Geometric representations - geom()

Given we want a scatter plot, we need to specify that the geometric representation of the data will be in point form, using geom_point().

Here we are adding a layer (hence the + sign) of points to the plot. We can think of this as similar to e.g. Adobe Photoshop which uses layers of images that can be reordered and modified individually. Because we add to plots layer by layer **the order** of your geoms may be important for your final aesthetic design.

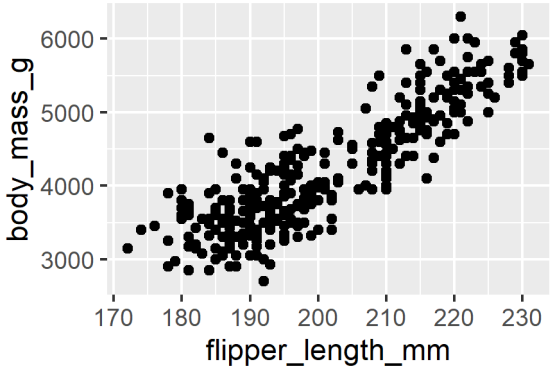
For ggplot, each layer will be added over the plot according to its position in the code. Below I first show the full breakdown of the components in a layer. Each layer requires information on

- data
- aesthetics
- geometric type
- any summary of the data
- position

```
penguins %>%
  ggplot(aes(x=flipper_length_mm,
             y = body_mass_g))+
  layer(                # layer inherits data and aesthetic arguments from previous
    geom="point",       # draw point objects
    stat="identity",    # each individual data point gets a geom (no summaries)
    position=position_identity()) # data points are not moved in any way e.g. we could specify ji
```

This is quite a complicate way to write new layers - and it is more usual to see a simpler more compact approach

```
penguins %>%
  ggplot(aes(x=flipper_length_mm,
             y = body_mass_g))+
  geom_point() # geom_point function will always draw points, and unless specified otherwise the
```



Now we have the scatter plot! Each row (except for two rows of missing data) in the penguins data set now has an x coordinate, a y coordinate, and a designated geometric representation (point).

From this we can see that smaller penguins tend to have smaller flipper lengths.

5.6 %>% and +

ggplot2, an early component of the tidyverse package, was written before the pipe was introduced. The + sign in ggplot2 functions in a similar way to the

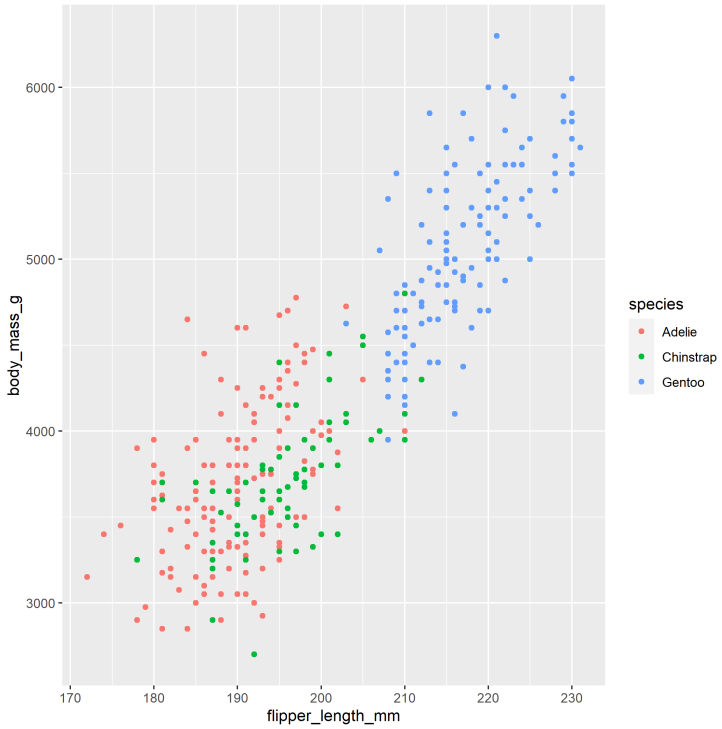
pipe in other functions in the tidyverse: by allowing code to be written from left to right.

5.7 Colour

The current plot could be more informative, to include information about the species of each penguin.

In order to achieve this we need to use aes() again, and specify which column we want to be represented as the colour of the points. Here, the aes() function containing the relevant column name, is given within the geom_point() function.

```
penguins %>%
  ggplot(aes(x=flipper_length_mm,
             y = body_mass_g))+
  geom_point(aes(colour=species))
```



**Note - you may (or may not) have noticed that the grammar of ggplot (and tidyverse in general) accepts British/Americanization for spelling!!!

So now we can see that the Gentoo penguins tend to be both larger and have

longer flippers

Remember to keep adding carriage returns (new lines), which must be inserted after the %>% or + symbols. In most cases, R is blind to white space and new lines, so this is simply to make our code more readable, and allow us to add readable comments.

5.8 More layers

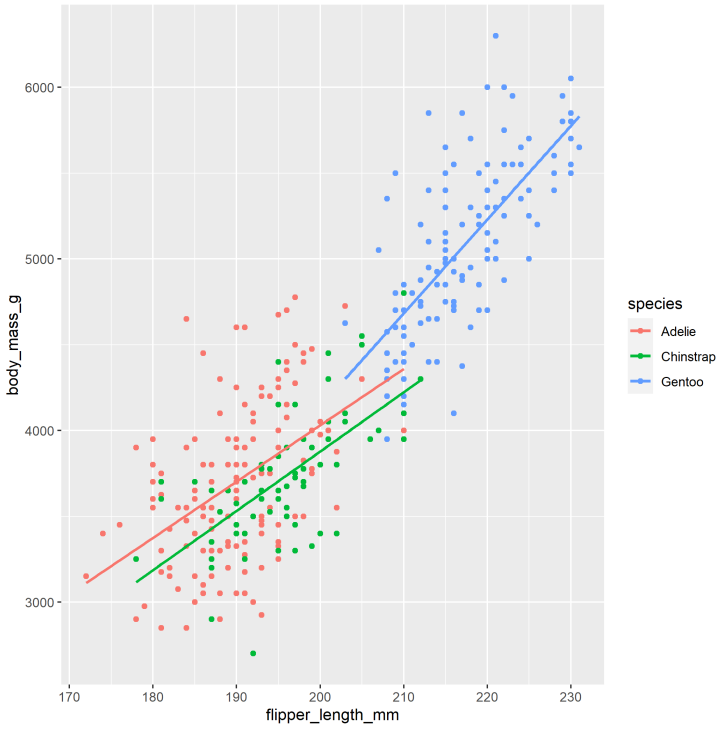
We can see the relationship between body size and flipper length. But what if we want to model this relationship with a trend line? We can add another 'layer' to this plot, using a different geometric representation of the data. In this case a trend line, which is in fact a summary of the data rather than a representation of each point.

The geom_smooth() function draws a trend line through the data. The default behaviour is to draw a local regression line (curve) through the points, however these can be hard to interpret. We want to add a straight line based on a linear model ('lm') of the relationship between x and y.

This is our **first** encounter with linear models in this course, but we will learn a lot more about them later on.

```
penguins %>%
  ggplot(aes(x=flipper_length_mm,
             y = body_mass_g))+
  geom_point(aes(colour=species))+
  geom_smooth(method="lm",    #add another layer of data representation.
              se=FALSE,
              aes(colour=species)) # note layers inherit information from the top ggplot() functi
```

```
penguins %>%
  ggplot(aes(x=flipper_length_mm,
             y = body_mass_g,
             colour=species))+ ### now colour is set here it will be inherited by ALL layers
  geom_point()+
  geom_smooth(method="lm",    #add another layer of data representation.
              se=FALSE)
```



> **Note - that the trend line is blocking out certain points, because it is the 'top layer' of the plot. The geom layers that appear early in the command are drawn first, and can be obscured by the geom layers that come after them.

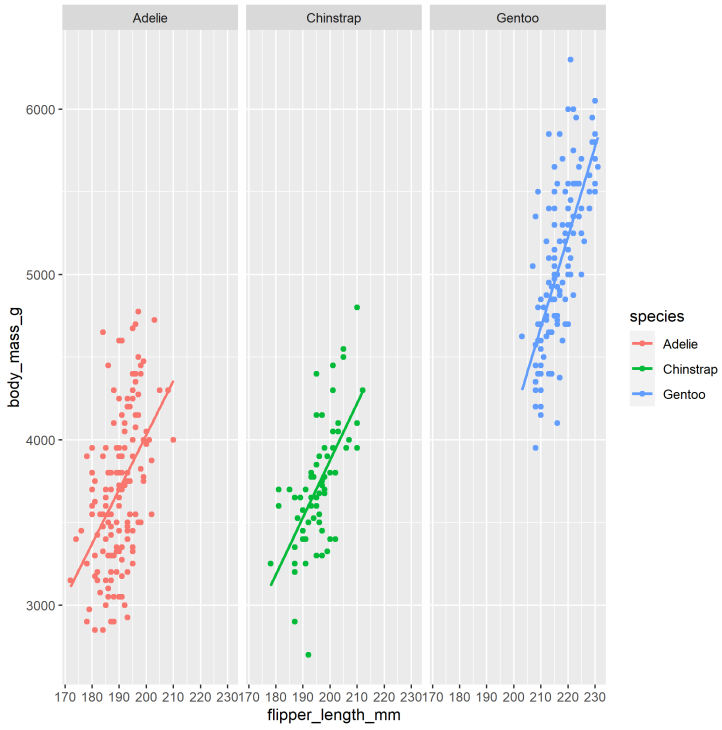
What happens if you switch the order of the geom_point() and geom_smooth() functions above? What do you notice about the trend line?

5.9 Facets

In some cases we want to break up a single plot into sub-plots, called 'faceting'. Facets are commonly used when there is too much data to display clearly in a single plot. We will revisit faceting below, however for now, let's try to facet the plot according to species. To do this we use the tilde symbol '~' to indicate the column name that will form each facet.

```
penguins %>%
  ggplot(aes(x=flipper_length_mm,
             y = body_mass_g,
             colour=species))+
```

```
  geom_point()+
  geom_smooth(method="lm",
              se=FALSE)+
  facet_wrap(~species)
```



**Note - the aesthetics and geoms including the regression line that were specified for the original plot, are applied to each of the facets.

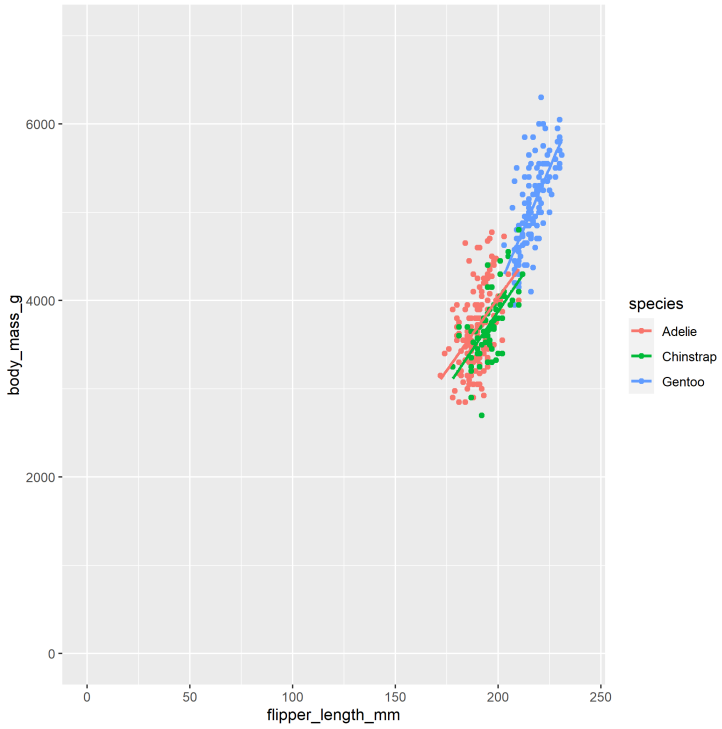
5.10 Co-ordinate space

ggplot will automatically pick the scale for each axis, and the type of coordinate space. Most plots are in Cartesian (linear X vs linear Y) coordinate space.

For this plot, let's say we want the x and y origin to be set at 0. To do this we can add in xlim() and ylim() functions, which define the limits of the axes:

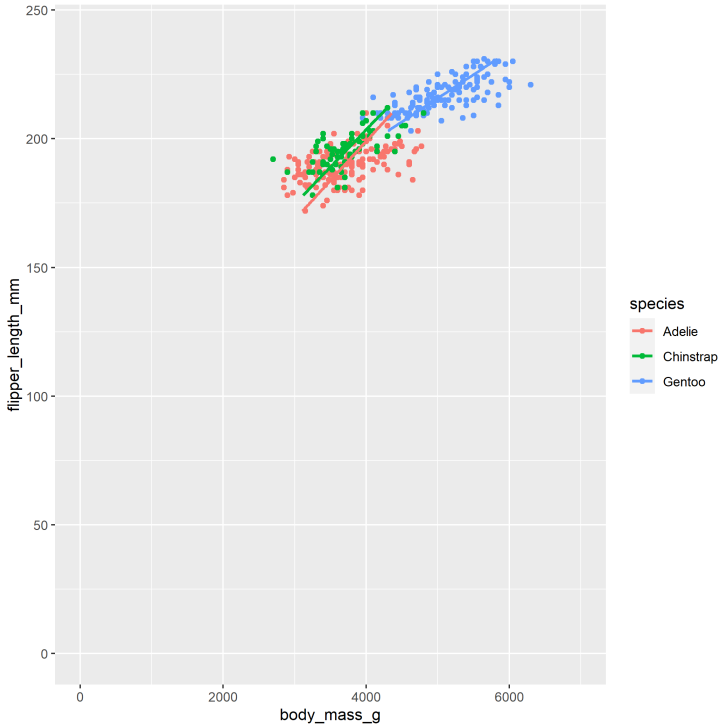
```
penguins %>%
  ggplot(aes(x=flipper_length_mm,
             y = body_mass_g,
             colour=species))+
```

```
  geom_point()+
  geom_smooth(method="lm",
              se=FALSE)+
  xlim(0,240) + ylim(0,7000)
```



Further, we can control the coordinate space using coord() functions. Say we want to flip the x and y axes, we add coord_flip():

```
penguins %>%
  ggplot(aes(x=flipper_length_mm,
             y = body_mass_g,
             colour=species))+
  geom_point()+
  geom_smooth(method="lm",
              se=FALSE)+
  xlim(0,240) + ylim(0,7000)+
  coord_flip()
```



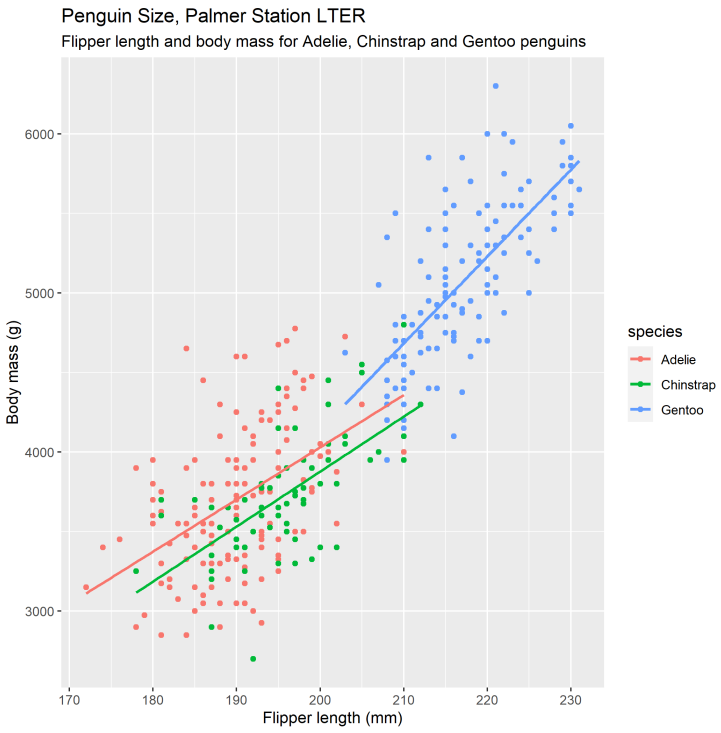
5.11 Labels

By default, the axis labels will be the column names we gave as aesthetics aes(). We can change the axis labels using the xlab() and ylab() functions. Given that column names are often short and can be cryptic, this functionality is particularly important for effectively communicating results.

```
penguins %>%
  ggplot(aes(x=flipper_length_mm,
             y = body_mass_g,
             colour=species))+
  geom_point()+
  geom_smooth(method="lm",
              se=FALSE)+
  labs(x = "Flipper length (mm)",
       y = "Body mass (g)")
```

We can also add titles and subtitles

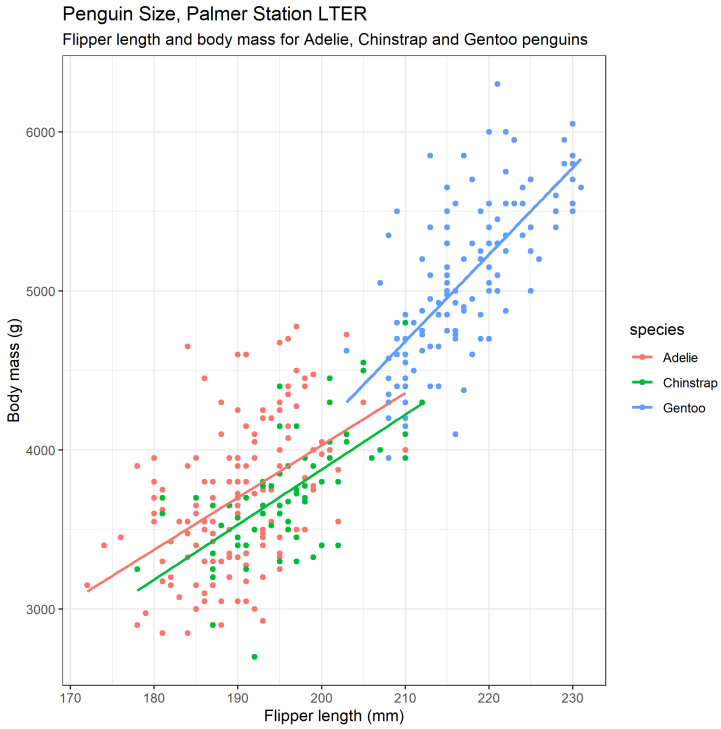
```
penguins %>%
  ggplot(aes(x=flipper_length_mm,
             y = body_mass_g,
             colour=species))+
  geom_point()+
  geom_smooth(method="lm",
              se=FALSE)+
  labs(x = "Flipper length (mm)",
       y = "Body mass (g)",
       title= "Penguin Size, Palmer Station LTER",
       subtitle= "Flipper length and body mass for Adelie, Chinstrap and Gentoo penguin
```



5.12 Themes

Finally, the overall appearance of the plot can be modified using theme() functions. The default theme has a grey background which maximizes contrast with other contrasts. You may prefer `theme_classic()`, a `theme_minimal()` or even `theme_void()`. Try them out.

```
penguins %>%
  ggplot(aes(x=flipper_length_mm,
             y = body_mass_g,
             colour=species))+
  geom_point()+
  geom_smooth(method="lm",
              se=FALSE)+
  labs(x = "Flipper length (mm)",
       y = "Body mass (g)",
       title= "Penguin Size, Palmer Station LTER",
       subtitle= "Flipper length and body mass for Adelie, Chinstrap and Gentoo penguins")+
  theme_bw()
```



**Note - there is a lot more customisation available through the theme() function. We will look at making our own custom themes in later lessons

You can try installing and running an even wider range of pre-built themes if you install the R package `ggthemes`.

First you will need to run the `install.packages("ggthemes")` command.

Remember this is one of the few times a command should NOT be written in your script but typed directly into the console. That's because it's rude to send someone a script that will install packages on their computer - think of library() as a polite request instead!

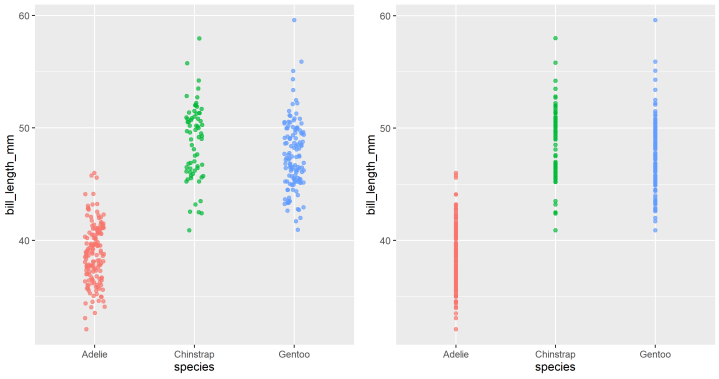
To access the range of themes available type `help(ggthemes)` then follow the documentation to find out what you can do.

5.13 Jitter

The geom_jitter() command adds some random scatter to the points which can reduce over-plotting. Compare these two plots:

```
ggplot(data = penguins, aes(x = species, y = bill_length_mm)) +
  geom_jitter(aes(color = species),
              width = 0.1, # specifies the width, change this to change the range of s
              alpha = 0.7, # specifies the amount of transparency in the points
              show.legend = FALSE) # don't leave a legend in a plot, if it doesn't add
```

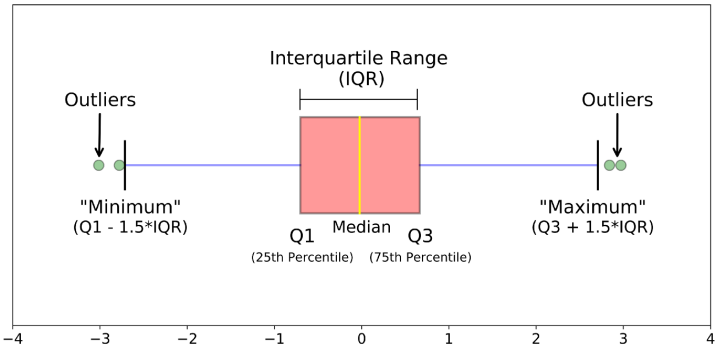
```
ggplot(data = penguins, aes(x = species, y = bill_length_mm)) +
  geom_point(aes(color = species),
              alpha = 0.7,
              show.legend = FALSE)
```



5.14 Boxplots

Box plots, or 'box & whisker plots' are another essential tool for data analysis. Box plots summarize the distribution of a set of values by displaying the

minimum and maximum values, the median (i.e. middle-ranked value), and the range of the middle 50% of values (inter-quartile range). The whisker line extending above and below the IQR box define Q3 + (1.5 x IQR), and Q1 - (1.5 x IQR) respectively. You can watch a short video to learn more about box plots here.



To create a box plot from our data we use (no prizes here) geom_boxplot()!

```
ggplot(data = penguins, aes(x = species, y = bill_length_mm)) +
  geom_boxplot(aes(fill = species), # note fill is "inside" colour and colour is "edges" - try it
               alpha = 0.7,
               width = 0.5, # change width of boxplot
               show.legend = FALSE)
```

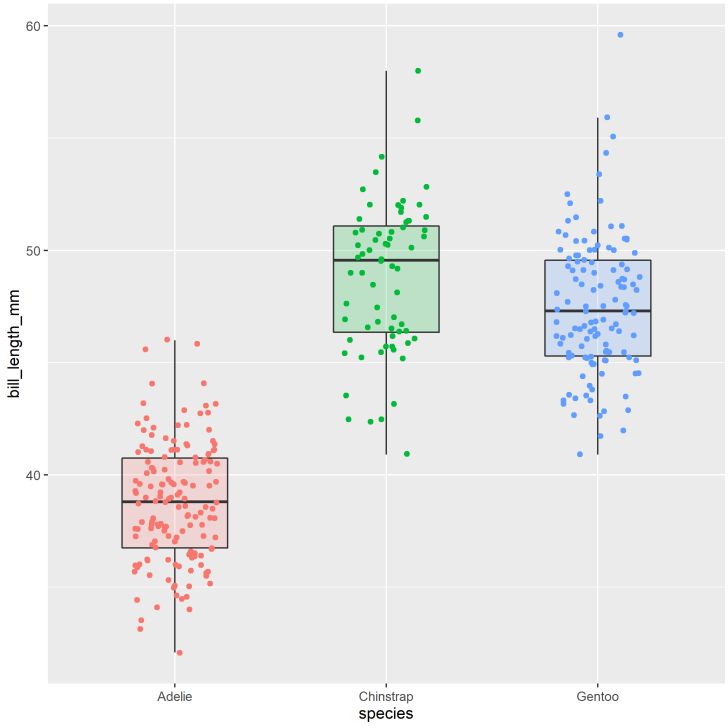
The points indicate outlier values [i.e., those greater than Q3 + (1.5 x IQR)].

We can overlay a boxplot on the scatter plot for the entire dataset, to fully communicate both the raw and summary data. Here we reduce the width of the jitter points slightly.

```
ggplot(data = penguins, aes(x = species, y = bill_length_mm)) +
  geom_boxplot(aes(fill = species), # note fill is "inside" colour and colour is "edges" - try it
               alpha = 0.2, # fainter boxes so the points "pop"
               width = 0.5, # change width of boxplot
               outlier.shape=NA)+
  geom_jitter(aes(colour = species),
                width=0.2)+
  theme(legend.position = "none")
```

In the above example I switched from using show.legend=FALSE inside the geom layer to using theme(legend.position="none"). Why? This is an example of reducing redunant code. I would have to specify

show.legend=FALSE for every geom layer in my plot, but the theme function applies to every layer. Save code, save time, reduce errors!

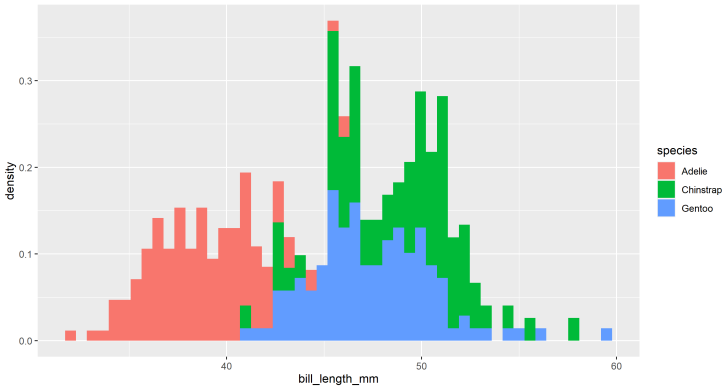


5.15 Density and histogram

Compare the following two sets of code

```
penguins %>%
    ggplot(aes(x=bill_length_mm, fill=species))+
    geom_histogram(bins=50)
```

```
penguins %>%
    ggplot(aes(x=bill_length_mm, fill=species))+
    geom_histogram(bins=50, aes(y=..density..))
```



At first you might struggle to see/understand the difference between these two charts. The shapes should be roughly the same.

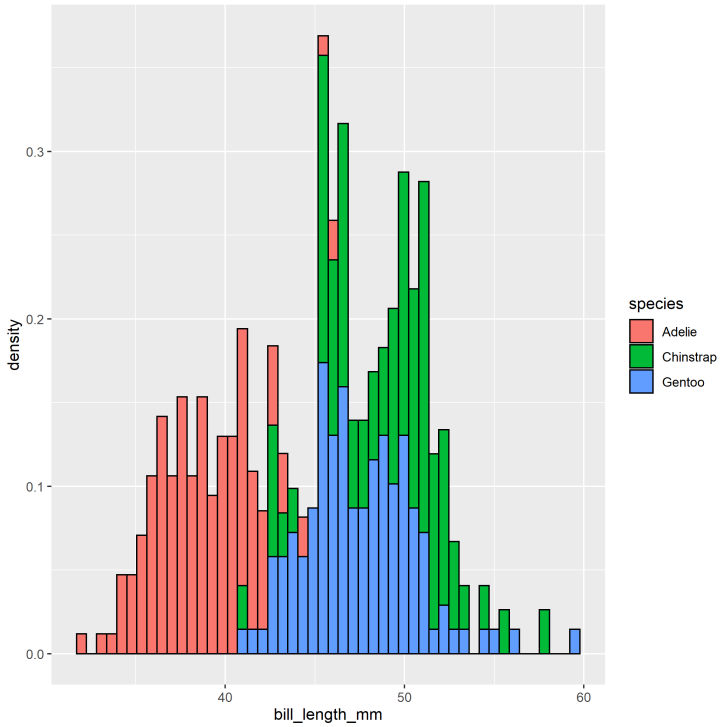
The first block of code produced a frequency histogram, each bar represents the actual number of observations made within each "bin", the second block of code shows the "relative density" within each bin. In a density histogram the area under the curve for each sub-group will sum to 1. This allows us to compare distributions and shapes between sub-groups of different sizes. For example there are far fewer Adelie penguins in our dataset, but in a density histogram they occupy the same area of the graph as the other two species.

5.16 More Colours

There are two main differences when it comes to colors in `ggplot2`. Both arguments, color and fill, can be specified as single color or assigned to variables.

As you have already seen in this tutorial, variables that are inside the aesthetics are encoded by variables and those that are outside are properties that are unrelated to the variables.

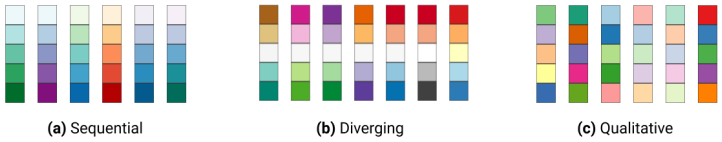
```
penguins %>%
    ggplot(aes(x=bill_length_mm))+
    geom_histogram(bins=50,
                   aes(y=..density..,
                       fill=species),
                   colour="black")
```



5.16.1 Assign colours to variables

You can specify what colours you want to assign to variables in a number of different ways.

In ggplot2, colors that are assigned to variables are modified via the scale_color_* and the scale_fill_* functions. In order to use color with your data, most importantly you need to know if you are dealing with a categorical or continuous variable. The color palette should be chosen depending on type of the variable, with sequential or diverging color palettes being used for continuous variables and qualitative color palettes for categorical variables:



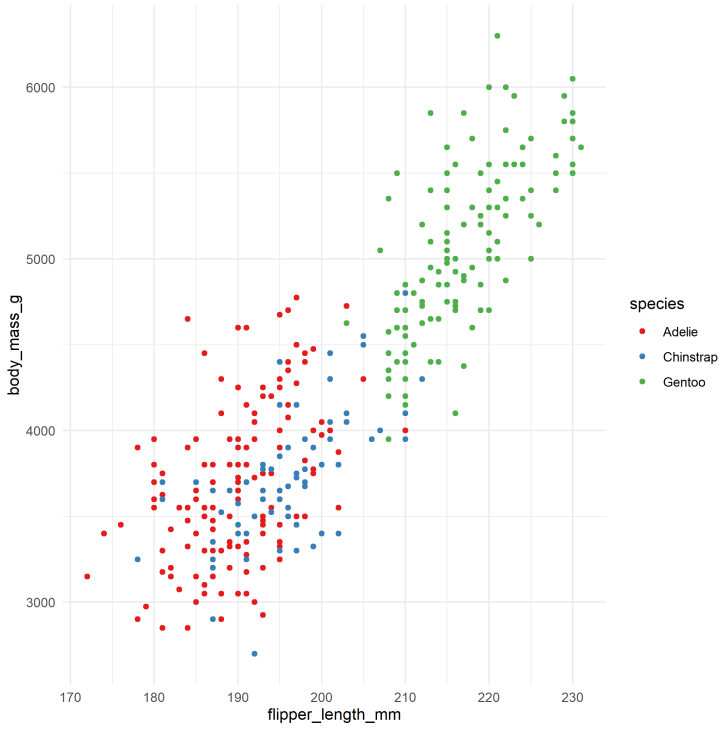
You can pick your own sets of colours and assign them to a categorical variable. The number of specified colours **has** to match the number of categories. You can use a wide number of preset colour names or you can use hexadecimals.

```
penguin_colours <- c("darkolivegreen4", "darkorchid3", "goldenrod1")

penguins %>%
  ggplot(aes(x=flipper_length_mm,
             y = body_mass_g))+
  geom_point(aes(colour=species))+
  scale_color_manual(values=penguin_colours)+
  theme_minimal()
```

You can also use a range of inbuilt colour palettes:

```
penguins %>%
  ggplot(aes(x=flipper_length_mm,
             y = body_mass_g))+
  geom_point(aes(colour=species))+
  scale_color_brewer(palette="Set1")+
  theme_minimal()
```



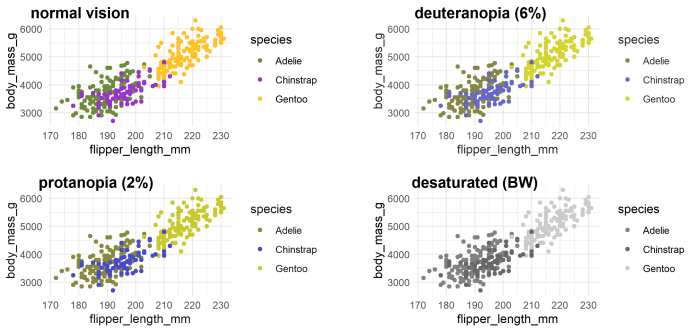
You can explore all schemes available with the command `RColorBrewer::display.brewer.all()`

There are also many, many extensions that provide additional colour palettes. Some of my favourite packages include `ggsci`(Xiao (2018)) and `wesanderson`(Ram and Wickham (2018)).

5.16.2 Accessibility

It's very easy to get carried away with colour palettes, but you should remember at all times that your figures must be accessible. One way to check how accessible your figures are is to use a colour blindness checker Ou (2021)

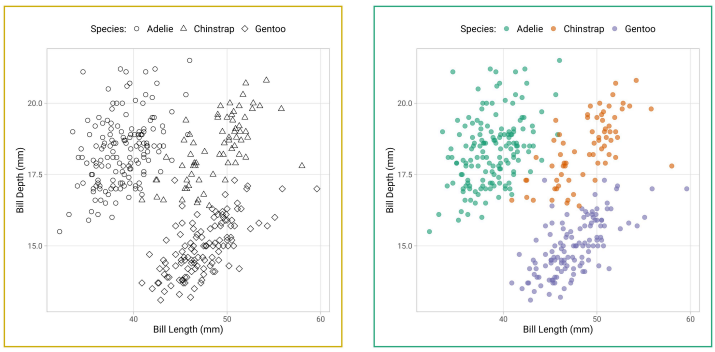
```
library(colorBlindness)
cvdPlot() # will automatically run on the last plot you made
```



5.16.3 Guides to visual accessibility

Using colours to tell categories apart can be useful, but as we can see in the example above, you should choose carefully. Other aesthetics which you can access in your geoms include `shape`, and `size` - you can combine these in complimentary ways to enhance the accessibility of your plots. Here is a hierarchy of "interpretability" for different types of data

Qualitative Nominal	**Qualitative** Ordinal	**Quantitative** Interal/Ratio
Position	Position	Position
Colour (Hue)	Pattern (Density)	Size (Length)
Pattern (Texture)	Colour (Lightness)	Angle
Connection	Colour (Hue)	Size (Area)
Pattern (Density)	Pattern (Texture)	Size (Volume)
Colour (Lightness)	Connection	Pattern (Density)
Symbol	Size (Length)	Colour (Lightness)
Size (Length)	Angle	Colour (Hue)
Angle	Size (Area)	Pattern (Texture)
Size (Area)	Size (Volume)	Connection
Size (Volume)	Symbol	Symbol



5.17 Patchwork

There are many times you might want to combine figures into multi-panel plots. Probably the easiest way to do this is with the `patchwork` package (Pedersen (2020)).

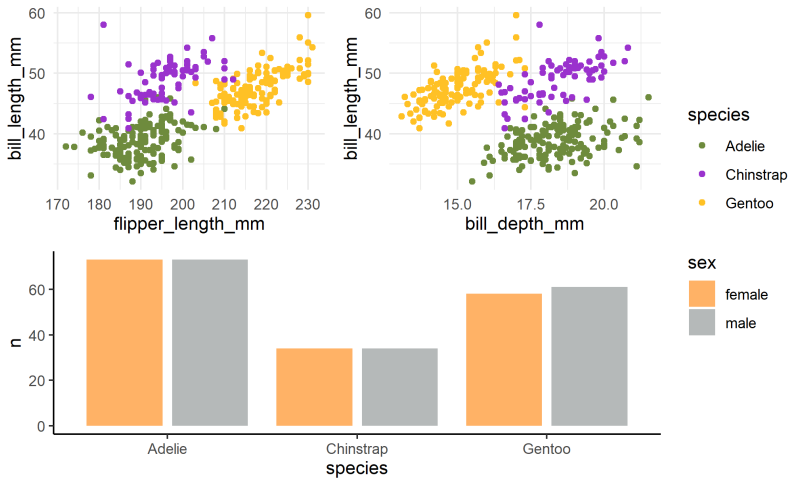
```
p1 <- penguins %>%
  ggplot(aes(x=flipper_length_mm,
             y = bill_length_mm))+
  geom_point(aes(colour=species))+
  scale_color_manual(values=penguin_colours)+
  theme_minimal()

p2 <- penguins %>%
  ggplot(aes(x=bill_depth_mm,
             y = bill_length_mm))+
  geom_point(aes(colour=species))+
  scale_color_manual(values=penguin_colours)+
  theme_minimal()

p3 <- penguins %>%
  group_by(sex,species) %>%
    summarise(n=n()) %>%
     drop_na(sex) %>%
     ggplot(aes(x=species, y=n)) +
  geom_col(aes(fill=sex),
               width=0.8,
               position=position_dodge(width=0.9),
               alpha=0.6)+
     scale_fill_manual(values=c("darkorange1", "azure4"))+
     theme_classic()
```

```
library(patchwork)

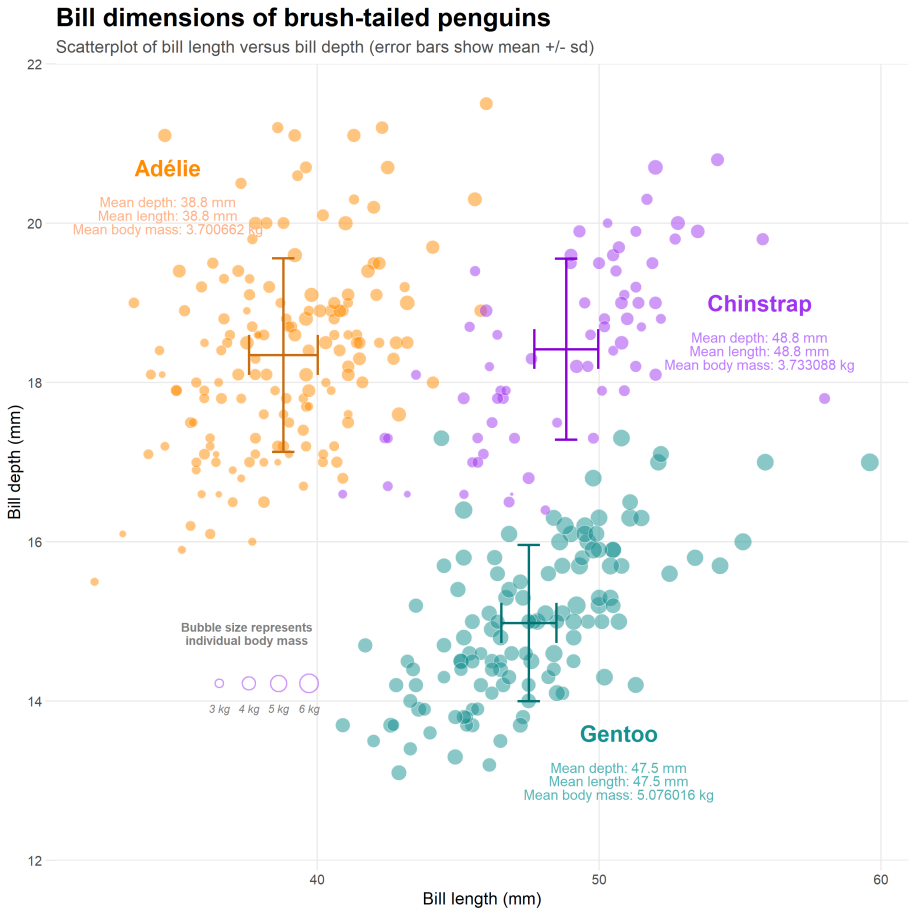
 (p1+p2)/p3+
  plot_layout(guides = "collect")
```



5.18 Test

Challenge - How close can you get to replicating the figure below?

Make sure to use the tips and links at the end of this chapter



5.19 Saving

One of the easiest ways to save a figure you have made is with the `ggsave()` function. By default it will save the last plot you made on the screen.

You should specify the output path to your **figures** folder, then provide a file name. Here I have decided to call my plot *plot* (imaginative!) and I want to save it as a .PNG image file. I can also specify the resolution (dpi 300 is good enough for most computer screens).

```
ggsave("figures/plot.png", dpi=300)
```

5.20 Quitting

Make sure you have saved your script!

Download your saved figure from RStudio Cloud and submit it to Blackboard "Week Four Test"

5.21 Summing up ggplot

5.21.1 What we learned

You have learned

- The anatomy of ggplots
- How to add geoms on different layers
- How to use colour, colour palettes, facets, labels and themes
- Putting together multiple figures
- How to save and export images

You have primarily worked with `tidyverse` and the package

- `ggplot2` Wickham et al. (2021b)

As well as:

- `colorBlindness` Ou (2021)
- `patchwork` Pedersen (2020)

5.21.2 Further Reading, Guides and tips

R Cheat Sheets

Wilke (2020) https://clauswilke.com/dataviz/

this book tells you everything you need to know about presenting your figures for accessbility and clarity

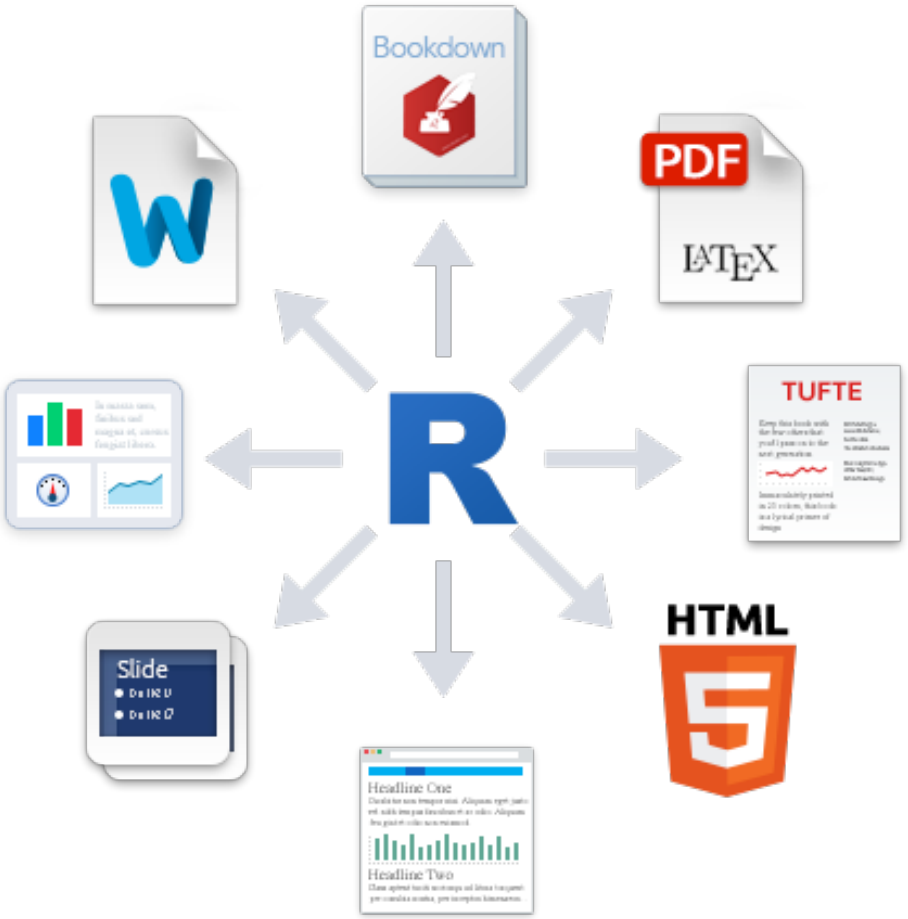
https://www.cedricscherer.com/2019/08/05/a-ggplot2-tutorial-for-beautiful-plotting-in-r/

an incredibly handy ggplot guide for how to build and improve your figures

Hadley Wickham and Pedersen (2020) https://ggplot2-book.org/
the original Hadley Wickham book on ggplot2

Chapter 6

Reports with Rmarkdown: Week Five



R Markdown is a widely-used tool for creating automated, reproducible, and share-worthy outputs, such as reports. It can generate static or interactive outputs, in Word, pdf, html, powerpoint, and other formats.

An R Markdown script combines R code and text such that the script actually becomes your output document. You can create an entire formatted document, including narrative text (can be dynamic to change based on your data), tables, figures, bullets/numbers, bibliographies, etc.

Documents produced with Rmarkdown, allow analyses to be included easily - and make the link between raw data, analysis & and a published report *completely reproducible.*

With Rmarkdown we can make reproducible html, word, pdf, powerpoints or websites and dashboards[1]

To make your Rmd publish - hit the knit button at the top of the doc

6.0.1 Format

- Go to RStudio Cloud and open **Week 5 - Rmarkdown**
- Follow the instructions carefully - and assemble your Rmarkdown file bit by bit - when prompted to 'knit' the document do it. We will then observed the results and might make changes.

6.1 Background to Rmarkdown

- Markdown is a "language" that allows you to write a document using plain text, that can be converted to html and other formats. It is not specific to R. Files written in Markdown have a '.md' extension.
- R Markdown: is a variation on markdown that is specific to R - it allows you to write a document using markdown to produce text and to embed R code and display their outputs. R Markdown files have '.Rmd' extension.
- rmarkdown - the package: This is used by R to render the .Rmd file into the desired output. It's focus is converting the markdown (text) syntax, so we also need…
- knitr: This R package will read the code chunks, execute it, and 'knit' it back into the document. This is how tables and graphs are included alongside the text.
- Pandoc: Finally, pandoc actually convert the output into word/pdf/powerpoint etc. It is a software separate from R but is installed automatically with RStudio.

[1] (https://rmarkdown.rstudio.com/lesson-9.html)

Most of this process happens in the background (you do not need to know all these steps!) and it involves feeding the .Rmd file to knitr, which executes the R code chunks and creates a new .md (markdown) file which includes the R code and its rendered output. The .md file is then processed by pandoc to create the finished product: a Microsoft Word document, HTML file, powerpoint document, pdf, etc. When using RStudio Cloud - all of these features are pre-loaded - if you take your R journey further in the future and install a copy of R and RStudio on your own computer you might have to do a little setting up to get this working.

6.2 Starting a new Rmd file

In RStudio, if you open a new R markdown file, start with 'File', then 'New file' then 'R markdown…'.

R Studio will give you some output options to pick from. In the example below we select "HTML" because we want to create an html document. The title and the author names are not important. If the output document type you want is not one of these, don't worry - you can just pick any one and change it in the script later.

For now open the markdown file which I have made in the Markdown sub-folder.

The working directory for .rmd files is a little different to working with scripts.

With a .Rmd file, the working directory is wherever the Rmd file itself is saved.

For example if you have your .Rmd file in a subfolder ~/markdown-files/markdown.Rmd the code for read_csv("data/data.csv") within the markdown will look for a .csv file in a subfolder called data *inside* the `markdownfiles` folder and not the root project folder where the .RProj file lives.

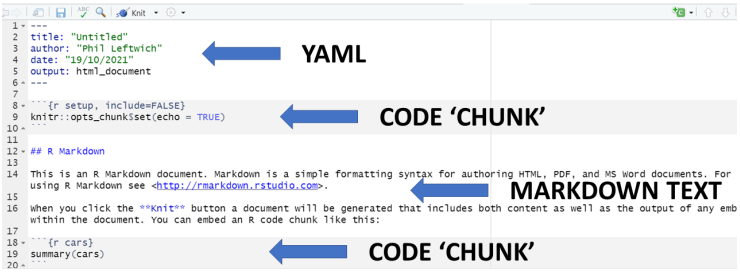
So we have two options when using .Rmd files

1) Don't put the .Rmd file in a subfolder and make sure it lives in the same directory as your .RProj file
2) Use the 'here' package to describe file locations - more later

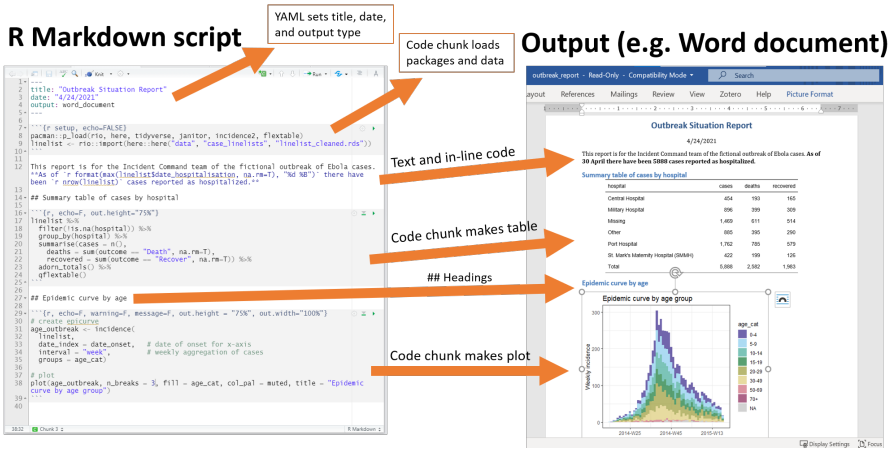
6.3 R Markdown parts

An R Markdown document can be edited in RStudio just like a standard R script. When you start a new R Markdown script, RStudio tries to be helpful by showing a template which explains the different section of an R Markdown script.

The below is what appears when starting a new Rmd script intended to produce an html output.



As you can see, there are three basic components to any Rmd file: YAML, Markdown text, and R code chunks.



6.3.1 YAML

Referred to as the 'YAML metadata' or just 'YAML', this is at the top of the R Markdown document. This section of the script will tell your Rmd file **what type of output to produce**, formatting preferences, and other metadata such as document title, author, and date. In the example above, because

we clicked that our default output would be an html file, we can see that the YAML says output: `html_document`. However we can also change this to say `powerpoint_presentation` or `word_document` or even `pdf_document`.

Can you edit the YAML in the Rmarkdown file in the markdown folder to have your name as author, today's date and the title of the file should be called "Darwin's Maize Plants".

6.3.2 Text

This is the narrative of your document, including the titles and headings. It is written in the "markdown" language, which is used across many different software.

Below are the core ways to write this text. See more extensive documentation available on R Markdown "cheatsheets" at the RStudio website[2].

6.3.2.1 New lines

Uniquely in R Markdown, to initiate a new line, enter *two spaces** at the end of the previous line and then Enter/Return.

6.3.2.2 Text emphasis

Surround your normal text with these characters to change how it appears in the output.

Underscores (_text_) or single asterisk (*text*) to *italicise*

Double asterisks (**text**) for **bold** text

Back-ticks (‘text‘) to display text as `code`

The actual appearance of the font can be set by using specific templates (specified in the YAML metadata).

6.3.2.3 Titles and headings

A hash symbol in a text portion of a R Markdown script creates a heading. This is different than in a chunk of R code in the script, in which a hash symbol is a mechanism to comment/annotate/de-activate, as in a normal R script.

Different heading levels are established with different numbers of hash symbols at the start of a new line. One hash symbol is a title or primary heading. Two hash symbols are a second-level heading. Third- and fourth-level headings can be made with successively more hash symbols.

[2](https://www.rstudio.com/resources/cheatsheets/)

```
# First-level heading / title

## Second level heading

### Third-level heading
```

6.3.2.4 Bullets and numbering

Use asterisks (*) to created a bullets list. Finish the previous sentence, enter two spaces, Enter/Return twice, and then start your bullets. Include a space between the asterisk and your bullet text. After each bullet enter two spaces and then Enter/Return. Sub-bullets work the same way but are indented. Numbers work the same way but instead of an asterisk, write 1), 2), etc. Below is how your R Markdown script text might look.

Here are my bullets (there are two spaces after this colon):

```
* Bullet 1 (followed by two spaces and Enter/Return)
* Bullet 2 (followed by two spaces and Enter/Return)
  * Sub-bullet 1 (followed by two spaces and Enter/Return)
  * Sub-bullet 2 (followed by two spaces and Enter/Return)
```

6.3.3 Code Chunks

Sections of the script that are dedicated to running R code are called "chunks". This is where you may load packages, import data, and perform the actual data management and visualisation. There may be many code chunks, so they can help you organize your R code into parts, perhaps interspersed with text. To note: These 'chunks' will appear to have a slightly different background colour from the narrative part of the document.

Each chunk is opened with a line that starts with three back-ticks, and curly brackets that contain parameters for the chunk ({ }). The chunk ends with three more back-ticks.

```
```{r}
```
```

You can create a new chunk by typing it out yourself, by using the keyboard shortcut "Ctrl + Alt + i" (or Cmd + Shift + r in Mac), or by clicking the green 'insert a new code chunk' icon at the top of your script editor.

Some notes about the contents of the curly brackets { }:

They start with 'r' to indicate that the language name within the chunk is R

After the r you can optionally write a chunk "name" – these are not necessary but can help you organise your work. Note that if you name your chunks, you

should ALWAYS use unique names or else R will *complain* when you try to render.

The curly brackets can include other options too, written as tag=value, such as:

- eval = FALSE to not run the R code
- echo = FALSE to not print the chunk's R source code in the output document
- warning = FALSE to not print warnings produced by the R code
- message = FALSE to not print any messages produced by the R code
- include = either TRUE/FALSE whether to include chunk outputs (e.g. plots) in the document
- out.width = and out.height = - size of ouput e.g. out.width = "75%"
- fig.align = "center" adjust how a figure is aligned across the page
- fig.show='hold' if your chunk prints multiple figures and you want them printed next to each other (pair with out.width = c("33%", "67%").

A chunk header must be written in one line

Try to avoid periods, underscores, and spaces. Use hyphens (-) instead if you need a separator.

Read more extensively about the knitr options here[3].

There are also two arrows at the top right of each chunk, which are useful to run code within a chunk, or all code in prior chunks. Hover over them to see what they do.

6.3.4 Here

The package `here` Müller (2020) and its function `here()` make it easy to tell R where to find and to save your files - in essence, it builds file paths.

This is how `here()` works within an R project:

- When the here package is first loaded within the R project, it places a small file called ".here" in the root folder of your R project as a "benchmark" or "anchor"
- In your scripts, to reference a file in the R project's sub-folders, you use the function here() to build the file path in relation to that anchor
- To build the file path, write the names of folders beyond the root, within quotes, separated by commas, finally ending with the file name and file extension as shown below

[3](https://yihui.org/knitr/options/)

- here() file paths can be used for both importing and exporting

So when you use here wrapped inside other functions for importing/exporting (like read_csv or ggsave) if you include here you can still use the RProject location as the root directory when 'knitting' Rmarkdown files, even if your markdown is tidied away into a separate markdown folder.

Can you take the code below and put it into an R code chunk?

Try your first "knit" to make a document.

```
library(here)
library(tidyverse)

darwin <- read_csv(here("data", "darwin.csv"))
darwin
```

Rmarkdown

```
library(here)
```

```
## here() starts at /cloud/project
```

```
library(tidyverse)
```

```
## ── Attaching packages ─────────────────────────────────────── tidyverse 1.3.0 ──
```

```
## ✓ ggplot2 3.3.5     ✓ purrr   0.3.4
## ✓ tibble  3.0.6     ✓ dplyr   1.0.2
## ✓ tidyr   1.1.2     ✓ stringr 1.4.0
## ✓ readr   1.3.1     ✓ forcats 0.5.0
```

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

```
darwin <- read_csv(here("data", "darwin.csv"))
```

```
## Parsed with column specification:
## cols(
##   pot = col_character(),
##   pair = col_double(),
##   Cross = col_double(),
```

Hmmm looks a little messy.

Can you edit the R chunk so that the code and any warnings/messages are invisible. But the code output is still printed?

Once you have made your edits try hitting 'knit' again.

6.3.4.1 Global options

For global options to be applied to all chunks in the script, you can set this up within your very first R code chunk in the script. This is very handy if you know you will want the majority of your code chunks to behave in the same way. For instance, so that only the outputs are shown for each code chunk and not the code itself, you can include this command in the R code chunk (and set this chunk to include=false):

```
knitr::opts_chunk$set(echo = FALSE)
```

6.3.4.2 In-text code

You can also include minimal R code within back-ticks. Within the back-ticks, begin the code with "r" and a space, so RStudio knows to evaluate the code as R code. See the example below.

‘ r Sys.Date()‘

When typed in-line within a section of what would otherwise be Markdown text, it knows to produce an r output instead 2022-01-28

The example above is simple (showing the current date), but using the same syntax you can display values produced by more complex R code (e.g. to calculate the min, median, max of a column). You can also integrate R objects or values that were created in R code chunks *earlier* in the script.

```
average_height <- darwin %>%
  summarise(mean_self=mean(Self),
            mean_cross=mean(Cross))
```

The average height of the self-crossed plants is ‘ r average_height [,1]‘cm while the outcrossed plants are ‘ r average_height [,2]‘cm, so on average the outcrossed plants are ‘ r average_height [,2]- average_height [,1]‘cm taller.

Can you include the above code block and make sure it runs without the code being visible? Underneath include the text - it should produce the code outputs in the knitted doc. Try it!

6.4 Images

You can include images in your R Markdown in several ways:

knitr::include_graphics("path/to/image.png")

```
knitr::include_graphics("../data/images/darwin.png")
# ../ is necessary here because in order to organise the file path starting in the markdown folde
```

Alternatively we could use the `here()` function - which means it doesn't matter where our markdown file 'lives'.

knitr::include_graphics(here::here("path", "to", "image.png"))

```
knitr::include_graphics(here("data", "images", "darwin.png"))
```

Can you get your document to knit with this new image included?

6.5 Tables

To create and manage able objects, we first pass the data frame through the `kable()` function. The package `kableExtra` Zhu (2020) gives us lots of extra styling options.[4]

Can you get this working? Add the library call for `kableExtra` to the first chunk of your Rmd file, then make a chunk for the below at the bottom of your file and hit knit to test.

```
darwin %>%
  summarise(mean_self=mean(Self),
            mean_cross=mean(Cross)) %>%
  kbl(caption="Table 1. example table caption") %>%
  kable_styling(bootstrap_options = "striped", full_width = F, position = "left")
```

6.6 Self-contained documents

For a relatively simple report, you may elect to organize your R Markdown script such that it is "self-contained" and does not involve any external scripts.

Everything you need to run the R markdown is imported or created within the Rmd file, including all the code chunks and package loading. This "self-contained" approach is appropriate when you do not need to do much data processing (e.g. it brings in a clean or semi-clean data file) and the rendering of the R Markdown will not take too long.

[4](https://cran.r-project.org/web/packages/kableExtra/vignettes/awesome_table_in_html.html)

In this scenario, one logical organization of the R Markdown script might be:

- Set global knitr options
- Load packages
- Import data
- Process data
- Produce outputs (tables, plots, etc.)
- Save outputs, *if applicable* (.csv, .png, etc.)

6.7 Source files

One variation of the "self-contained" approach is to have R Markdown code chunks "source" (run) other R scripts.

This can make your R Markdown script less cluttered, more simple, and easier to organize. It can also help if you want to display final figures at the beginning of the report.

In this approach, the final R Markdown script simply combines pre-processed outputs into a document.

```
source(here("scripts", "your-script.R"))
```

Can you try it for yourself? There is a pre-written script in your R project, just use the source command to read in this script - then you can call objects made externally - in this case a penguin plot - put the code block in and hit knit.

```
source(here("scripts", "penguin script.R"))
scat ### object generated in penguin script
```

6.8 Exercise - Make your own Rmd document

The data found in "darwin.csv" was originally collected by Charles Darwin and published in *The effects of cross and self-fertilisation in the vegetable kingdom*, in 1876. In this publication he described how he produced seeds of *Zea mays* (maize) that were fertilised with pollen from the same individual (inbreeding) or by crossing to a different plant (outbreeding). Pairs of seeds were then taken from self-fertilised and crossed plants and germinated in pots, and the height of the young seedlings measured as a proxy for their fitness. Darwin wanted to know whether inbreeding reduced their fitness.

Can you make your first reproducible document?

- Write a short background on this subject 250-300 words on the data and the experimental hypothesis (use some of the information above)
- Write a results section on this data including:
  - A figure of the individual data points
  - A summary results table with the mean and s.d. for the plants
  - Write a summary of the results where the figures and table are referenced but you *must* describe the results too. e.g.

"Results are shown in Figure 1" is **not ok**

But *in your own words* describe the direction and effect size of any differences and then refer to Figures and tables

"The mean height of self-crossed plants is x and the mean height of crossed plants is y, meaning on average self-crossed plants are… which may indicate that … (Figure 1, Table 1)"

- knit your rmarkdown as a **pdf** and submit to Blackboard.

Remember your code blocks must be in the right order!!!

6.9 Summing up Rmarkdown

6.9.1 What we learned

You have learned

- How to use markdown
- How to embed R chunks, to produce code, figures and analyses
- How to organise projects with `here`
- How to knit to pdf and html outputs
- How to make simple tables

You have used

- `knitr` as part of Rmarkdown
- `kableExtra` Zhu (2020)
- `here` Müller (2020)

6.9.2 Further Reading, Guides and tips

R Cheat Sheets

Xie (2015) Dynamic documents with Rmarkdown

The fully comprehensive guide

(https://rmarkdown.rstudio.com/articles_intro.html)

(https://rmarkdown.rstudio.com/authoring_quick_tour.html)

Chapter 7

Version control with GitHub: Week Six

7.1 Let's Git it started

Git is a **version control system**. Originally built to help groups of developers work collaboratively on big software projects. It helps us manage our RStudio projects - with tracked changes.

Git and GitHub are a big part of the data science community. We can use GitHub in a number of ways

1) To source code and repurpose analyses built by others for our own uses

2) Manage our analysis projects so that all parts of it:

- Data
- Scripts
- Figures
- Reports

Are version controlled and open access

3) Version control lets you recover from any mistakes & your analysis is *backed up* externally

4) When you come to publish any reports - your analysis is accessible to others

5) Build up your own library of projects to show what you can do in Data Science

Watch this video *before* or *after* today's session

7.1.1 Will this be fun?

No.

Using GitHub and version control is a bit like cleaning your teeth. It's not exactly fun, but it's good for you and it promotes excellent hygiene. It also only takes about 2 minutes.

When we talk a about projects on GitHub we refer to Repositories / repos.

Repos on GitHub are the same unit as an RStudio Project - it's a place where you can easily store all information/data/etc. related to whatever project you're working on.

The way in which we will set up our RStudio Projects will now have a few extra steps to it

- Make a new GitHub repository (or *fork* an existing one)
- Make a New project on RStudio Cloud - selecting the *from GitHub Repository* option
- Clone your GitHub repo into an RStudio Project
- Make sure RStudio and Github can *talk* to each other
- Go about your normal business
- When it comes to *saving* your files, you will also periodically make a **commit** - this takes a multi-file *snapshot* of your *entire* project
- At the end of a session **push** your changes to GitHub.

These changes to working with RStudio will feel a little different at first, but will quickly become routine - and are a big step forward in your Data Science skills.

7.1.2 The Payoff

- **A portfolio** build up a library of data science projects you can show off
- **be keen** track the development of R packages on GitHUb
- **version control** keep a safe archive of all your edits and changes
- **play with others** easy ways to collaborate on data science projects

For the full rundown on how to use Git and R you can't go wrong with checking out Bryan (2012)

7.2 Set up GitHub

First things first you will need to set yourself up with a GitHub account.

Head to GitHub and sign up for a free account.

Make a careful note of

- The username you choose
- Use the same email you have signed up to RStudio Cloud with
- Note your password carefully!

7.3 Exercise 1. Fork & clone an existing repo on GitHub, make edits, push back

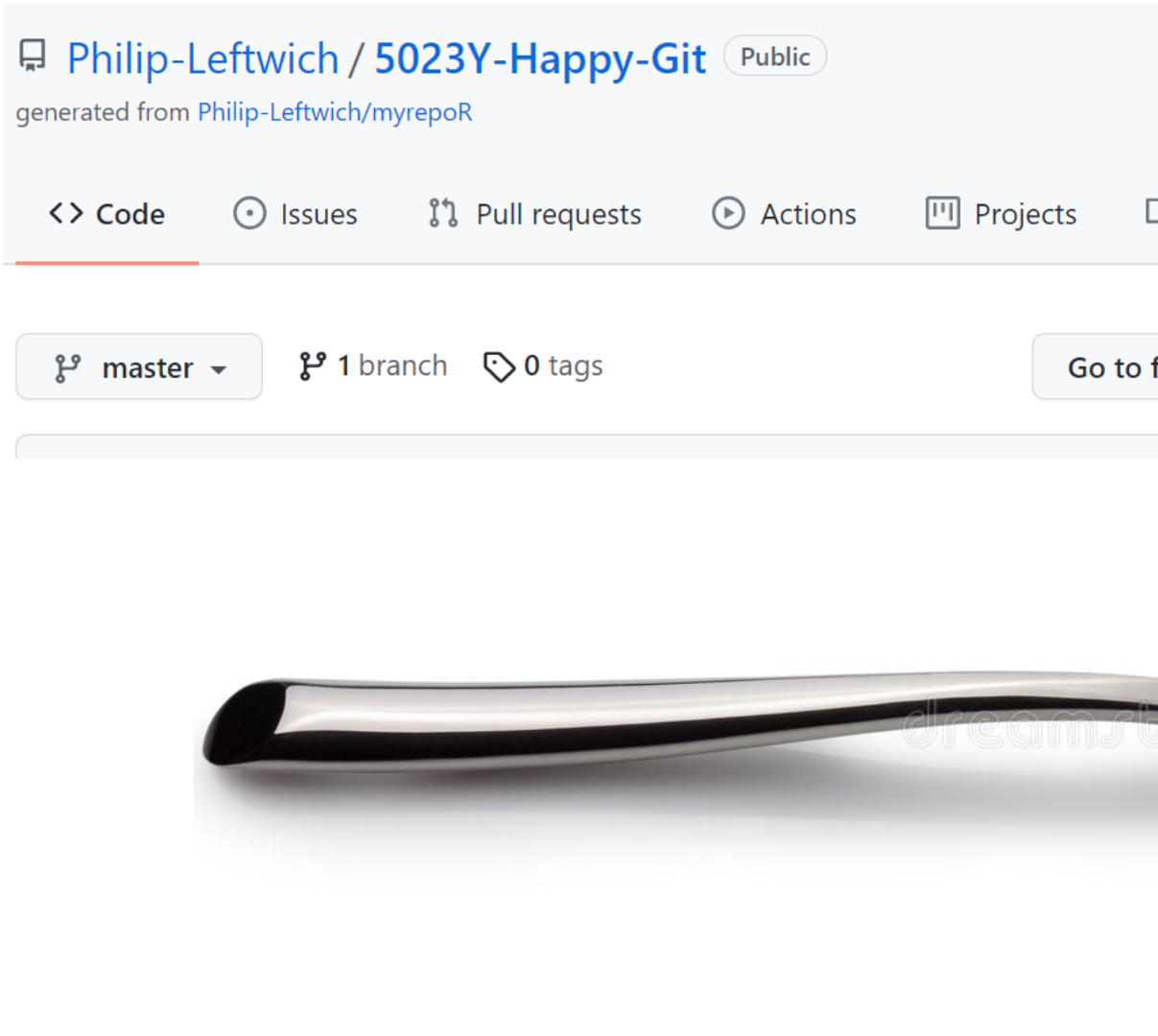
a. Go to github.com and log in (you need your own account - for sign up with your uea.ac.uk e-mail)

b. In the Search bar, look for repo **Philip-Leftwich/5023Y-Happy-Git**

c. Click on the repo name, and look at the existing repo structure

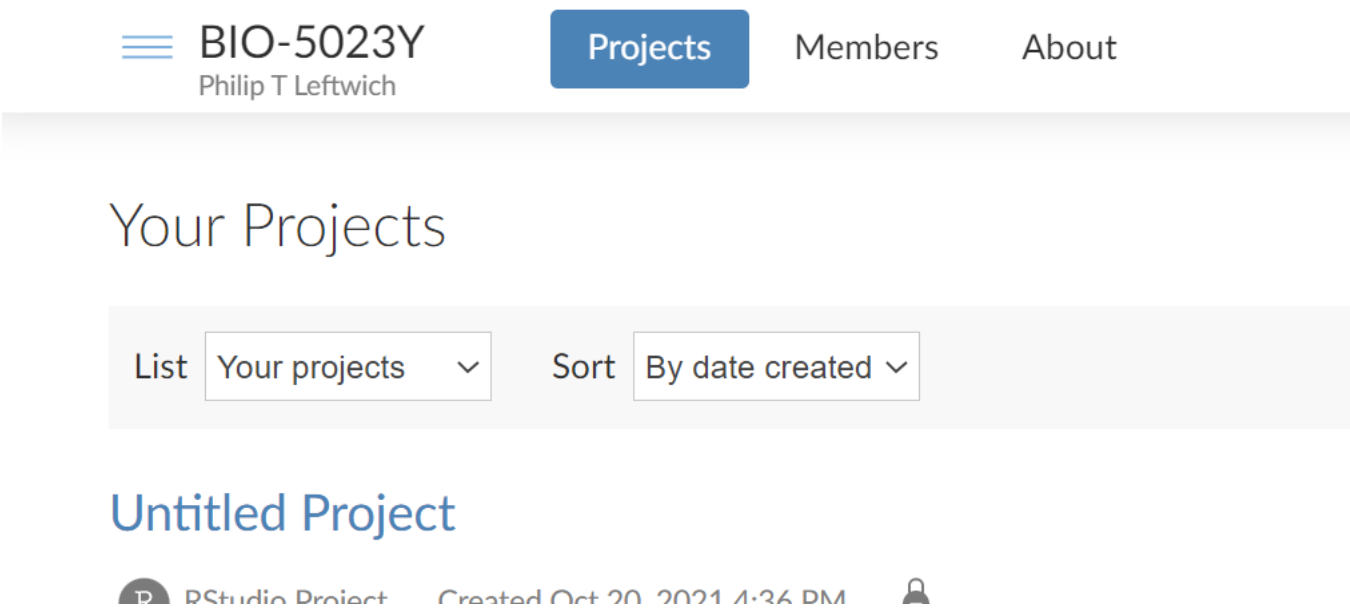
d. FORK the repo

7.3.1 What the hell is a fork?



A fork is when you generate a *personal* copy of another user's repository (7.8).

e. Press Clone/download and copy the URL, then create a **new** project in RStudio Cloud selecting the **New project from Git repository** option - make sure you are in the 5023Y Workspace



f. Open the some_cool_animals.Rmd document, and the accompanying html

g. Add *your name* to the top of the document

h. BUT WAIT. We have forgotten to add a great image and facts about a very important species - Baby Yoda, including an image (the file is in the repo, and the info to add is below).

FACTS

- Also known as “The Child”
- likes unfertilised frog eggs & control knobs
- strong with the force

i. Once you’ve added Grogu, knit the Rmd document to update the html

j. Add your Git credentials go to section (7.4)

k. Stage, Commit & Push all files (7.8)

Staged - pick those files which you intend to bind to a commit

Commit - write a short descriptive message, binds changes to a single commit (7.5.1)

Push - "Pushes" your changes from the local repo to the remote repo on GitHub (7.5.2)

l. On GitHub, refresh and see that files are updated. Cool! Now you've used something someone else has created, customized it, and saved your updated version.

7.4 Talking to GitHub

Getting set up to talk to GitHub can seem like a pain. Eventually when you work on your own computer - with a copy of R & RStudio installed - you will only have to do this once. For now when we use RStudio Cloud - it looks like we have to do this **once per project**. It only takes a few seconds and you should put these commands **directly into your console**.

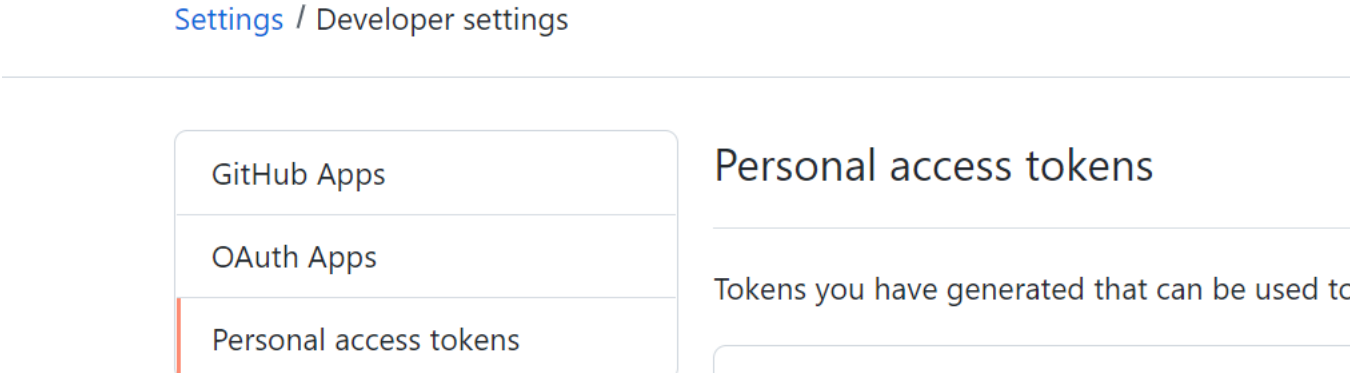
Run this first line **in your console** and put in your GitHub username and the e-mail connected to your GitHub account.

> **Note - you might not have to do this first step if you go to your RStudio Cloud profile > Authentication and select Github Enabled & Private repo access also enabled

```
usethis::use_git_config(user.name = "Jane Doe", user.email = "jane@example.org")
```

Then you need to give RStudio Cloud your GitHub Personal Access Token, which you can retrieve by going to Settings > Develope Settings > Generate Token (7.8)

Select all the "scopes" and name your token.



Make a note of this because you will need it whenever you set up a new project you want to talk to GitHub. GitHub recently removed password authentication in favour of PATs, but RStudio Cloud doesn't seem to have updated this yet - that's ok though - just enter this line of code - then copy+paste your PAT when prompted. - Option set/replace these credentials.

```
gitcreds::gitcreds_set()
```

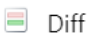
If you forget your PAT - that's ok - you can't retrieve it - but you can just generate a new one.

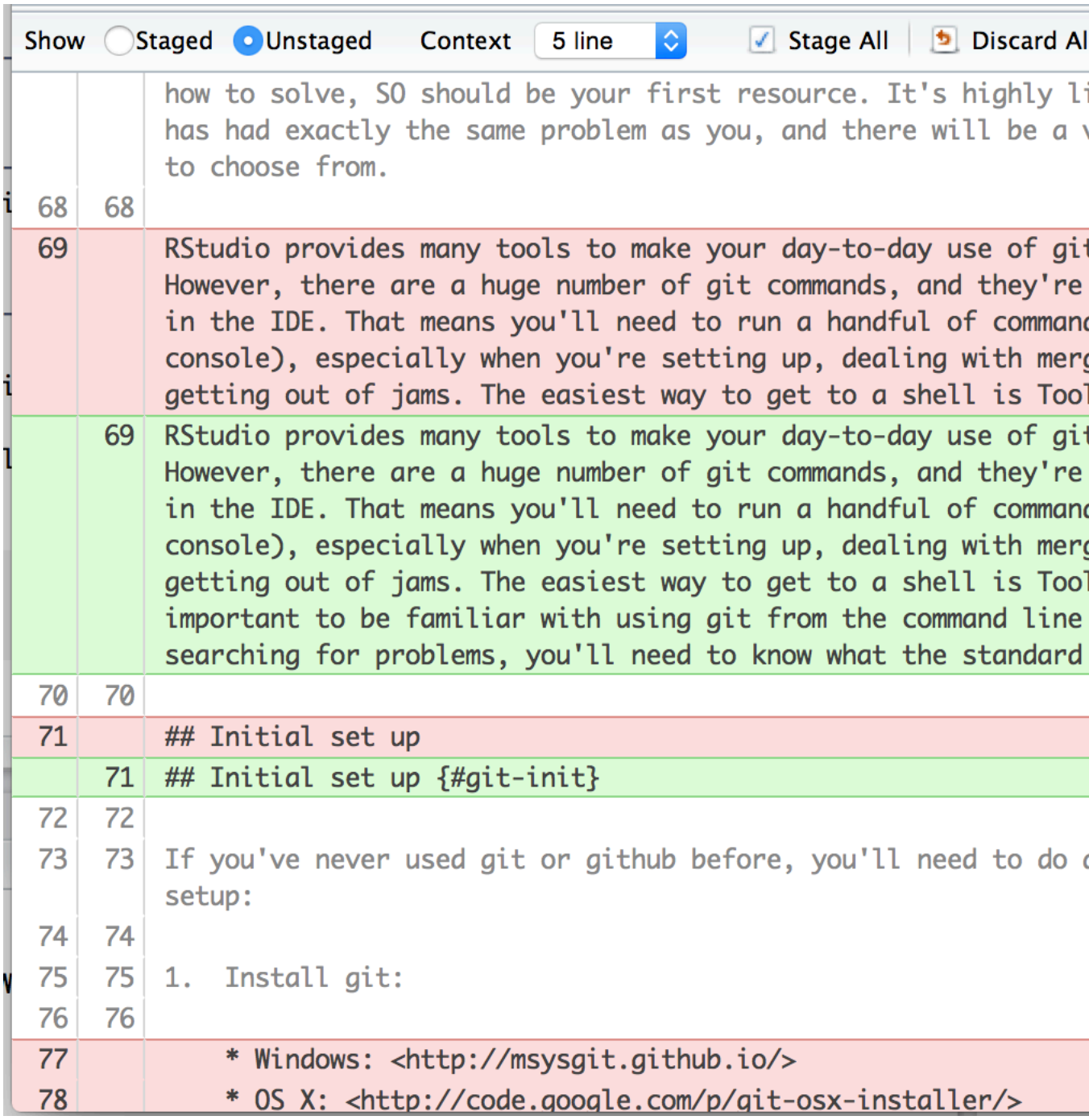
7.4.1 See changes

The first and most immediate benefit of using GitHub with your RStudio Project is seeing the changes you have made since your last commit.

The RStudio Git pane lists every file that's been added, modified or deleted. The icon describes the change:

-  You've changed a file
-  You've added a new file Git hasn't seen before
-  You've deleted a file

You can get more details on the changes that have been made to each file by right-clicking and selecting diff  . This opens a new window highlighting the **diff**erences between your current file and the previous commit.



The background colours tells you whether the text has been added (green) or removed (red). (If you're colourblind you can use the line numbers in the two

columns at the far left as a guide).

7.5 How to use version control - when to commit, push and pull

Repositories (repos) on GitHub are the same unit as an RStudio Project - it's a place where you can easily store all information/data/etc. related to whatever project you're working on.

When we create a Repository in GitHub and have it communicating with a Project in RStudio, then we can get (**pull**) information from GitHub to RStudio, or **push** information from RStudio to GitHub where it is safely stored and/or our collaborators can access it. It also keeps a *complete history* of updated versions that can be accessed/reviewed by you and your collaborators at any time, from anywhere, as long as you have the internet.

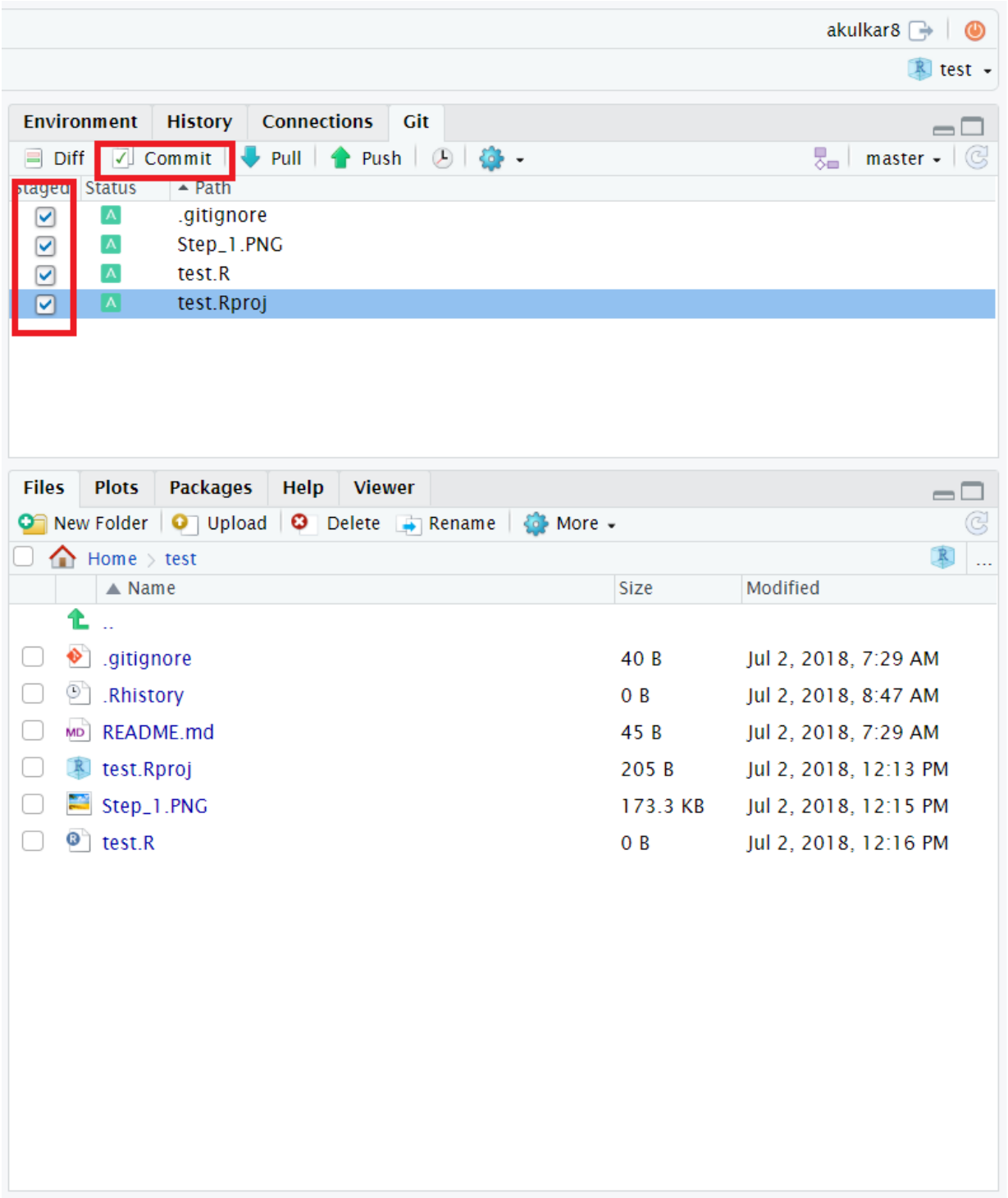
I have mentioned the term **commit** a few times (7.8). The fundamental unit of work in Git is a commit. A commit takes a snapshot of your code at a specified point in time.

You create a commit in two stages:

1. You **stage** files, telling Git which changes should be included in the next commit.

2. You **commit** the staged files, describing the changes with a message.

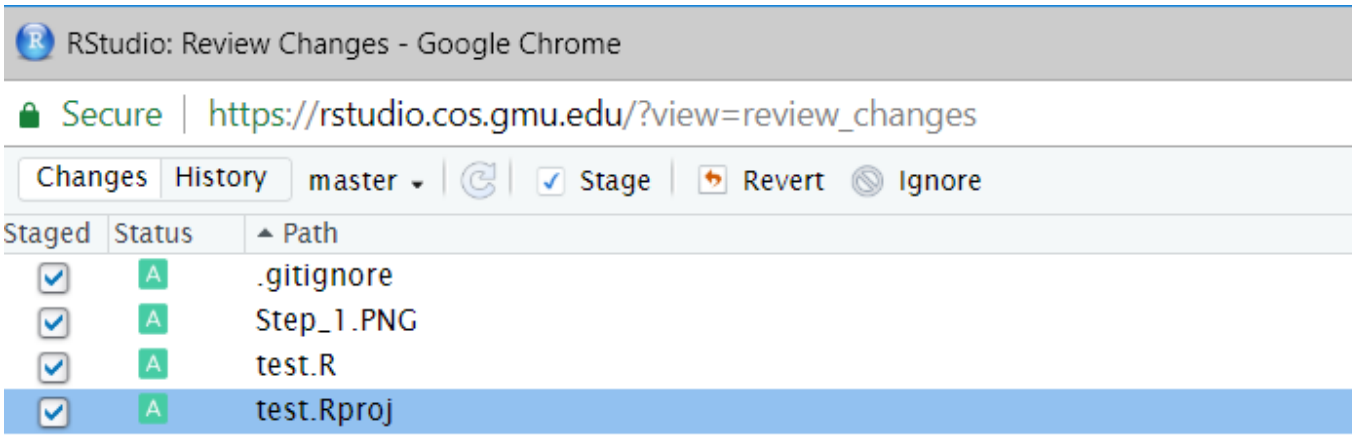
To create a new commit, **save** your files, then select files for inclusion by *staging* them, tick the checkbox and then select the commit box

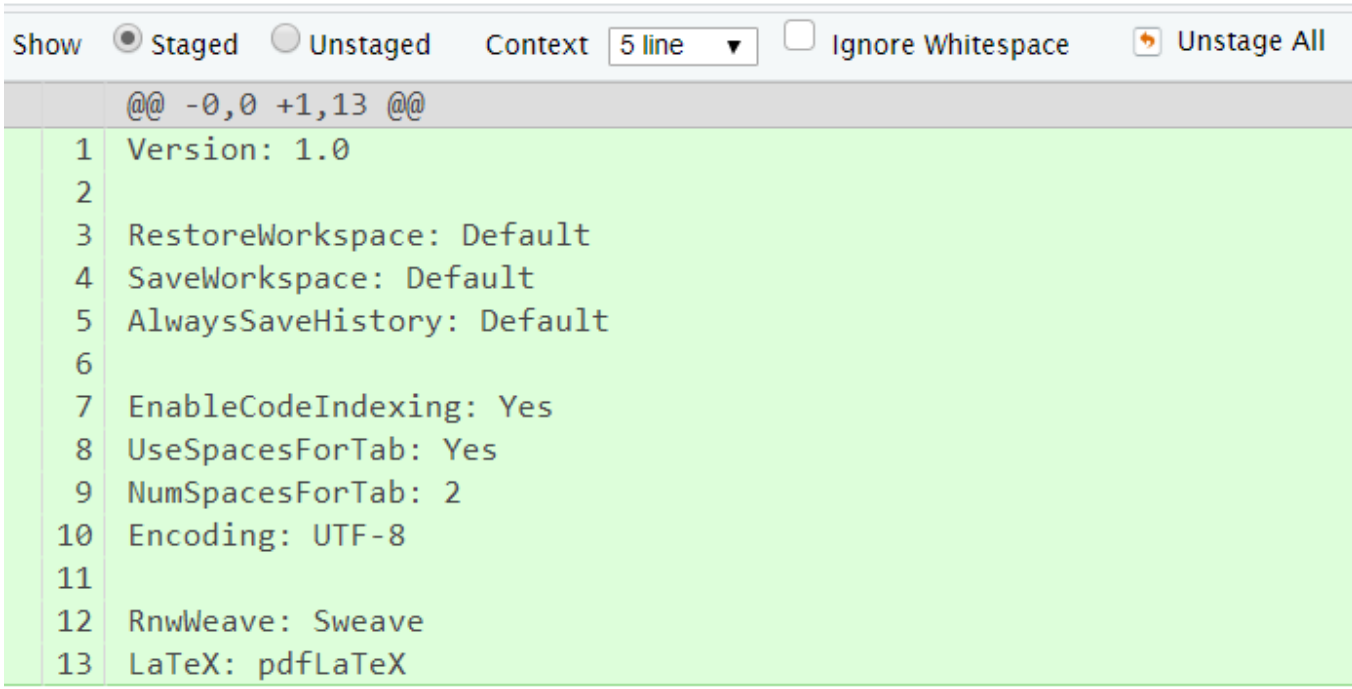

akulkar8
test
Environment
History
Connections
Git
Diff
Commit
Pull
Push
master
Staged
Status
Path
.gitignore
Step_1.PNG
test.R
test.Rproj
Files
Plots
Packages
Help
Viewer
New Folder
Upload
Delete
Rename
More
Home
test
Name
Size
Modified
..
.gitignore
40 B
Jul 2, 2018, 7:29 AM
.Rhistory
0 B
Jul 2, 2018, 8:47 AM
README.md
45 B
Jul 2, 2018, 7:29 AM
test.Rproj
205 B
Jul 2, 2018, 12:13 PM
Step_1.PNG
173.3 KB
Jul 2, 2018, 12:15 PM
test.R
0 B
Jul 2, 2018, 12:16 PM

A new window will open - and you will see the diffs in the bottom pane, and all the files you have selected for the latest commit.

7.5.1 Commit

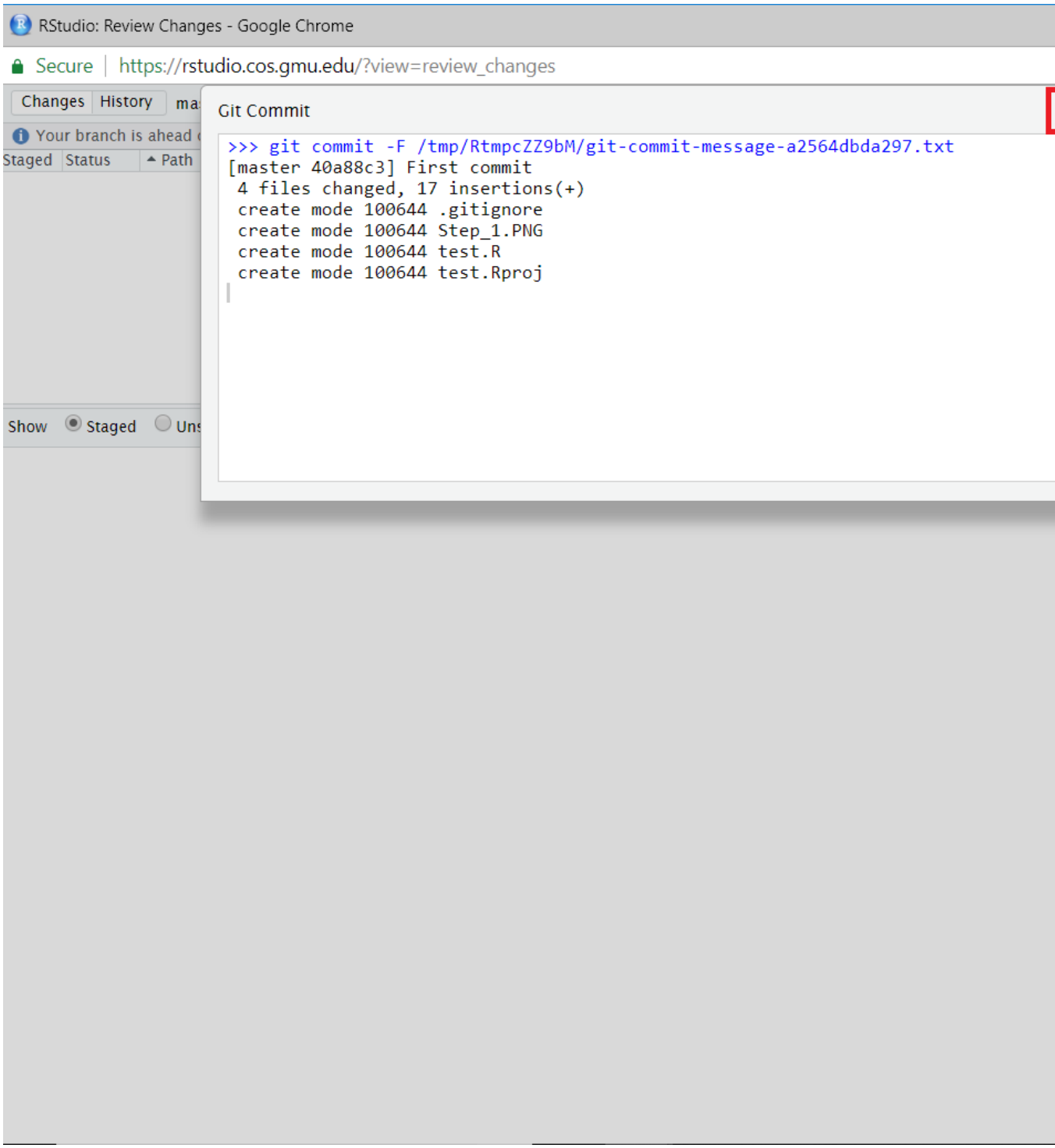
You now need to write a **commit message**. This should be short but meaningful


RStudio: Review Changes - Google Chrome
Secure | https://rstudio.cos.gmu.edu/?view=review_changes
Changes
History
master
Stage
Revert
Ignore
Staged
Status
Path
.gitignore
Step_1.PNG
test.R
test.Rproj


Show
Staged
Unstaged
Context
5 line
Ignore Whitespace
Unstage All
@@ -0,0 +1,13 @@
1 Version: 1.0
2
3 RestoreWorkspace: Default
4 SaveWorkspace: Default
5 AlwaysSaveHistory: Default
6
7 EnableCodeIndexing: Yes
8 UseSpacesForTab: Yes
9 NumSpacesForTab: 2
10 Encoding: UTF-8
11
12 RnwWeave: Sweave
13 LaTeX: pdfLaTeX

Describe the **why**, not the what. Git stores all the associated differences between commits, the message doesn't need to say exactly what changed. Instead it should provide a summary that focuses on the **reasons** for the change. Make this understandable for someone else!

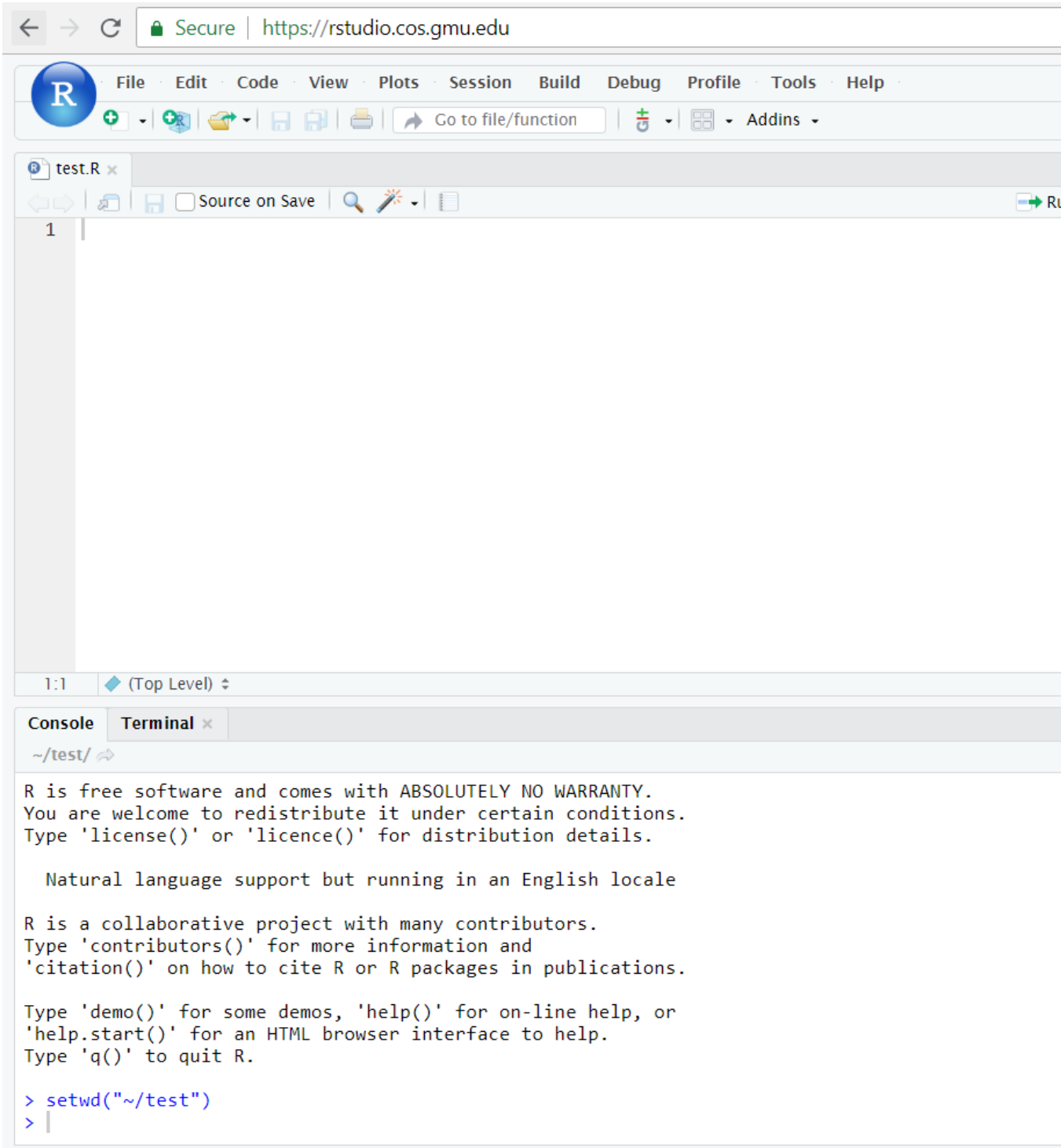
Once you click commit a new window will open that summarises your commit - and you can close this


RStudio: Review Changes - Google Chrome
Secure
https://rstudio.cos.gmu.edu/?view=review_changes
Changes
History
Git Commit
Your branch is ahead
Staged
Status
Path
Show
Staged
>>> git commit -F /tmp/RtmpcZZ9bM/git-commit-message-a2564dbda297.txt
[master 40a88c3] First commit
4 files changed, 17 insertions(+)
create mode 100644 .gitignore
create mode 100644 Step_1.PNG
create mode 100644 test.R
create mode 100644 test.Rproj

7.5.2 Push

At the moment, all of your commits are *local*, in order to send them to GitHub you have to select **Push** at this point your github credentials need to be in place - if you get prompted to provide these, close the windows and follow the steps here (7.4) before trying again.

Your git pane will be empty at this point - but there is a little message at the top detailing how many commits you are *ahead* of the master repo on GitHub.


Secure | https://rstudio.cos.gmu.edu
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
test.R
Source on Save
1
1:1
(Top Level)
Console
Terminal
~/test/
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
> setwd("~/test")
>

7.5.3 A couple of general tips:

- Pull at the start of **every session** this retrieves the master repo from GitHub - which you update at the end of every session. This helps prevent *conflicts*
- **Commit/push** in small, meaningful increments and do this often. You can make **multiple** commits in a session - and **always push at the end of the session**
- In this way your GitHub Repo becomes the **master copy** of your project.

7.5.4 Turn back time with Git

"If I could turn back time,

If I could find a way,

I'd take back those commits that have hurt you,

And you'd stay" - Cher

Once you start using Git with RStudio - you get a whole bunch of different options for undoing changes, correcting mistakes and turning back time to previous versions!

- To undo changes between commits you can select the diff option and remove lines one-by-one
- Right-click on a file in the Git pane and select the revert option will undo all changes between this and the previous commit **beware** cannot be undone.

If you don't catch a mistake right away you can select the history button  and **pull** previous commits from GitHub

7.6 Exercise 2. GitHub Classrooms enabled R Projects with subfolders

GitHub Classrooms is a way for me to set repos as assigments - when you accept an assignment on GitHub Classroom it *automatically* forks a private repo for you.

You should make regular commits and pushes to save your work as you go - and I will be grading your project repositories on GitHub classrooms when you do your assignment work.

When you accept an assignment on GitHub classrooms - the repo won't appear on your main profile, this is because it belongs to our class rather than you. You can always find it by searching through your Organisations - **but** it's probably easiest just to make a bookmark/make a note of your unique URL for each assignment.

a. Follow this invite link

b. You will be invited to sign-in to Github (if not already) & to join the UEABIO organisation

c. Clone your assignment to work locally in RStudio Cloud - 5023Y Workspace

d. In your local project folder, create subfolders 'data' and 'figures', 'scripts', 'markdown' (**Note** use the dir.create commands in the console)

e. Drop the file disease_burden.csv into the 'data' subfolder

f. Open a new .R script

g. Attach the `tidyverse`, `janitor`, and optionally `here` packages

h. Read in the infant_mortality.csv data

This is a file look at the death rate for every country in the world across six decades. See the README for more information

i. Stage, commit & push at this point - notice that the empty folder 'final_graphs' doesn't show up (won't commit an empty folder) - **you will have to set up your git credentials again**

j. Back in the script, write a short script to read and clean the data.

This script is pre-written, it puts the data in tidy format, cleans names, makes sure year is treated as date data and filters four countries of interest.

Assign this to a new object

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

infant_mortality <- read_csv("data/infant_mortality.csv")

subset_infant_mortality <- infant_mortality %>%
  pivot_longer(cols="1960":"2020",
               names_to="year",
               values_to="infant_mortality_rate") %>%
  mutate(year=lubridate::years(year)) %>% # set ymd format
```

```
  mutate(year=lubridate::year(year)) %>% # extract year
  janitor::clean_names() %>% # put names in snake case
  filter(country_name %in%
           c("United States",
             "Japan",
             "Afghanistan",
             "United Kingdom")) # extract four countries

# View(subset_infant_mortality)

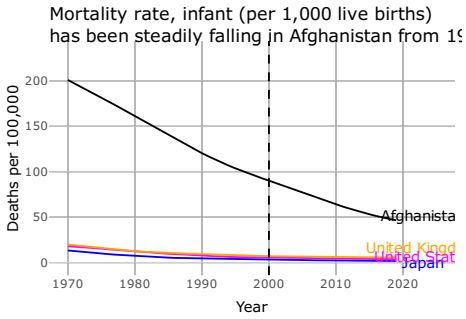
# subset the date according to (US,UK, Japan = lowest infant death rates, Afghanistan =
```

k. Make a ggplot plotting the infant mortality rate by country

HINT - use `geom_line()` and remember to separate countries by colour

```
ggplot(data = subset_infant_mortality) +
  geom_line(aes(x = year,
                y = infant_mortality_rate,
                color = country_name))
```

l. Update your graph with direct labels (using `annotate`) and vertical or horizontal lines with `geom_vline` or `geom_hline`.



m. Use ggsave() to write your graph to a .png in the 'final_graphs' subfolder

```
ggsave("figures/infant mortality graph.png", plot=mortality_figure, dpi=900, width = 7, height =
```

n. Save, stage, commit

o Let's do one more cool and fun thing! And make an interactive version of our plot using plotly Sievert et al. (2021) just for fun!

```
ggplotly(mortality_figure, tooltip = c("infant_mortality_rate"))
## uses plotly package
```

p Now save, stage, commit & **push**

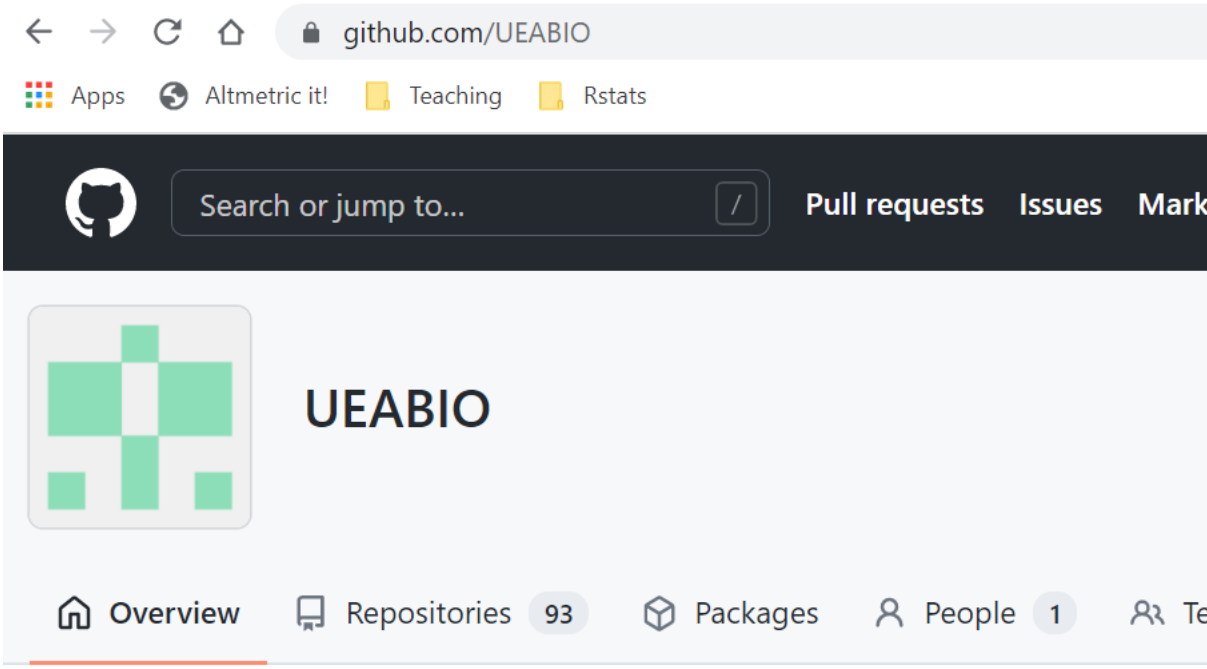
q. Check that changes are stored on GitHub

(**NOTE** this will be in your organisations rather than repos)

Make sure you finish both exercises before next week to become a GitHub pro!!!!!!!!!

7.7 Find your classroom repos

When you work with GitHub classrooms your repos become part of our organisation UEABIO. If you want to find your repos on GitHub then you can use the direct URL (if you noted it), or head to (https://github.com/UEABIO) - you should only be able to see repos that belong to **you**.



7.8 Glossary-GitHub

Terms	Description
clone	A clone is a copy of a repository that lives on your computer instead of on a w
commit	A commit, or revision, is an individual change to a file (or set of files). When
commit message	Short, descriptive text that accompanies a commit and communicates the cha
fork	A fork is a personal copy of another user's repository that lives on your accou
Git	Git is an open source program for tracking changes in text files. It was writte
GitHub Classroom	GitHub Classroom automates repository creation and access control, making i
Markdown	Markdown is an incredibly simple semantic file format, not too dissimilar from
pull	Pull refers to when you are fetching in changes and merging them. For instanc
push	To push means to send your committed changes to a remote repository on Git
README	A text file containing information about the files in a repository that is typica
repository	A repository (repo) is the most basic element of GitHub. They're easiest to in
RMarkdown	Rmarkdown is a package and filetype that are deeply embedded with RStudio
personal access token	A token that is used in place of a password when performing Git operations o

7.9 Summing up GitHub

7.9.1 What we learned

You have learned

- How to fork and clone GitHub Projects
- How to use GitHub classrooms
- How to make RStudio and GitHub talk to each other
- How to use version control, with stage, commit and push

You have used

- `gitcreds` Csárdi (2020)
- `usethis` Wickham et al. (2021a)

7.9.2 Further Reading, Guides and tips

Bryan (2012) https://happygitwithr.com/

The definitive guide

Chapter 8

Dealing with data: dplyr: Week Eight

8.1 Let's get going

You need to accept the latest assignment from Github Classrooms.

We will be using this project until the end of term, so make sure you make regular commits, and push your work at the end of each session.

8.2 Introduction to dplyr

In chapters 3 and 4 we demonstrated a brief data analysis on the `palmerpenguins` dataset. We ran through several pipelines, but without going into huge amounts of detail on each section.

In this chapter we are going to get really well acquainted with **dplyr** Wickham et al. (2021c).

We will look at several core functions:

- `select` (get some columns)
- `filter` (get some rows)
- `arrange` (order the rows)
- `mutate` (make new columns)
- `group_by` (add grouping information)
- `summarise` (calculate summary information)

You should make careful notes about these functions and what they do. Build scripts with carefully added notes that you can use for future work

As you work through this (and other chapters) - make notes, if you take a break, set up a commit, or reach the end of a session - push your work to Github.

Make sure you are using the R Cheat Sheets for dplyr

8.3 select

Start by setting up the packages you will need for this session

```
library(tidyverse)
library(lubridate)

penguins <- read_csv("data/penguins_simple.csv")
```

> ** Note - if you don't have these packages available, then you will need to run `install.packages("tidyverse")` - you should do this in the *console* and **not** your script.

We use select to *select variables* from a dataframe or tibble. for example:

```
data_set %>%
  select(variable_1, variable_2)
```

> ** Note this is *pseudocode* an example, not something you should run.

- The data is piped into the first argument position of the `select` function
- We then include the arguments where each one is the name of a variable in the dataset

For example if we actually run this:

```
penguins %>%
  select(flipper_length_mm)
```

To look at flipper length - we should note a couple of things.

- Variable name is **not** in quotes
- The original data is unchanged e.g. penguins. Unless we assign the results of `select` with the assignment arrow (<-), the results will just be printed in the console
- The order you ask for the variables in `select` determines the order of the columns in any new dataset

- `select` returns a tibble
- If we want to keep most variables, but *remove one*, use the minus operator

```
penguins <- penguins %>%
  select(-flipper_length_mm)
```

In the above option we just *overwrote* the penguin dataset with a new version that *does not* contain flipper length. Be careful - the above code will thrown an error message if you try to run it again (there is no longer a flipper length variable to remove). If you want to *undo* this, the easiest way is to re-run your script to just before this line.

You can also use select to keep sets of consecutive variables together

```
penguins %>%
  select(species:flipper_length_mm)
```

There are also a number of helper functions like `starts_with`, `ends_with`, `contains`, `matches`. So if we want to keep all the variables that start with "b"

```
penguins %>%
  select(starts_with("b"))
```

8.4 mutate

Adding or creating new variables is a common task, we might want to make a log-transformation, subtract one value from another or use `mutate` to add a new date variable

```
penguins <- penguins %>%
  mutate(date_proper=dmy(date))
```

Sometimes we want to change the values being used. For example we might want to change "male" and "female" to some abbreviation like "M" and "F". (Probably not, but it's an example!).

```
  penguins %>%
  mutate(sex=case_when(sex == "male" ~ "M",
                       sex == "female" ~ "F"))
```

Things to know about `mutate`:

- Unless we assign the results of `mutate` back to an object (<-) the changes won't be saved
- We can use newly created variables in further calculations *within* the same `mutate`

8.5 filter

`filter` is used to subset observations in a dataframe. We might want to see the observations where `flipper_length_mm` is larger or smaller than a fixed value. Or we might want to only see the data from female penguins, or a particular species.

```
penguins %>%
  filter(sex == "female",
         species == "Adelie",
         flipper_length_mm < 180)
```

In this example we've created a subset of `penguins` that only includes the five observations where flipper length is less than 180mm in female, Adelie penguins.

We can also set an either/or filter, for example if we only want small & large flipper lengths.

```
penguins %>%
  filter(species == "Adelie",
         flipper_length_mm <180 |
           flipper_length_mm >200)
```

This creates a subset of Adelie penguins, that only includes 14 observations where flipper length is less than 180mm or greater than 200mm.

The vertical bar | is understood by R as OR.

The alternative is to look at values *between* two amounts

```
penguins %>%
  filter(species == "Adelie",
         between(flipper_length_mm, 180,200))
```

8.6 arrange

We use arrange to reorder the rows of our data. Instances where we 'need' to do this are rare. But we may often want to reorder rows so that we can understand the data more easily

```
penguins_ordered <- penguins %>%
  arrange(sex,body_mass_g)

# arrange data first by sex - all females then all males, within each sex order body mass from lo

penguins_ordered %>%
  select(penguin_id, sex, body_mass_g)
# view just a few variables
```

By default arrange sorts from low to high (and alphabetically) - we can reverse this if we wrap the variable names in the function `desc`

```
penguins_reverse_ordered <- penguins %>%
  arrange(desc(sex,body_mass_g))



penguins_reverse_ordered %>%
  select(penguin_id, sex, body_mass_g)


# we can also apply this to specific variables
penguins_reverse_ordered <- penguins %>%
  arrange(sex,desc(body_mass_g))
```

8.7 Group and summarise

Very often we want to make calculations aobut groups of observations, such as the mean or median. We are often interested in comparing responses among groups. For example, we previously found the number of distinct penguins in our entire dataset

```
penguins %>%
  summarise(n_distinct(penguin_id))
```

Now consider when the groups are subsets of observations, as when we find out the number of penguins in each species and sex

```
penguins %>%
  group_by(species, sex) %>%
  summarise(n_distinct(penguin_id))
```

We are using summarise and group_by a lot! They are very powerful functions:

- `group_by` adds *grouping* information into a data object, so that subsequent calculations happen on a *group-specific* basis.
- `summarise` is a data aggregation function thart calculates summaries of one or more variables, and it will do this separately for any groups defined by `group_by`

8.7.1 Using summarise

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

**Note - we provide informative names for ourselves on the left side of the =.

There are a number of different calculations we can use including:

- `min` and `max` to calculate minimum and maximum values of a vector
- `mean` and `median`
- `sd` and `var` calculate standard deviation and variance of a numeric vector

We can use several functions in `summarise`. Which means we can string several calculations together in a single step

```
penguins %>%
  summarise(n=n(),
            num_penguins = n_distinct(penguin_id),
            mean_flipper_length = mean(flipper_length_mm, na.rm=TRUE),
            prop_female = sum(sex == "female", na.rm=TRUE) / n())
```

**Note - we have placed each argument on a separate line. This is just stylistic, it makes the code easier to read.

8.7.2 Summarize all columns

We can use the function `across` to count up the number of NAs in every column (by specifying `everything`).

```
penguins %>%
  summarise(across(.cols = everything(),
                   .fns = ~sum(is.na(.)))) %>%
  glimpse()
```

It has two arguments, `.cols` and `.fns`. The `.cols` argument lets you specify column types, while the `.fns` argument applies the required function to all of the selected columns.

```
# the mean of ALL numeric columns in the data, where(is.numeric) hunts for numeric columns

penguins %>%
  summarise(across(.cols = where(is.numeric),
                   .fns = ~mean(., na.rm=TRUE)))
```

The above example calculates the means of any & all numeric variables in the dataset.

The below example is a slightly complicated way of running the n_distinct for summarise. The `.cols()` looks for any column that contains the word "penguin" and runs the `n_distinct()`command of these

```
# number of distinct penguins, as only one column contains the word penguin

penguins %>%
  summarise(across(.cols = contains("penguin"),
                   .fns = ~n_distinct(.))) %>%
  glimpse()
```

8.7.3 group_by summary

The `group_by` function provides the ability to separate our summary functions according to any subgroups we wish to make. The real magic happens when we pair this with `summarise` and `mutate`.

In this example, by grouping on the individual penguin ids, then summarising by n - we can see how many times each penguin was monitored in the course of this study.

```
penguin_stats <- penguins %>%
  group_by(penguin_id) %>%
  summarise(num=n())
```

**Note - the actions of group_by are powerful, but group_by on its own doesn't change the visible structure of the dataframe.

8.7.4 More than one grouping variable

What if we need to calculate by more than one variable at a time?

```
penguins_grouped <- penguins %>%
  group_by(sex, species)
```

We can then calculate the mean flipper length of penguins in each of the six combinations

```
penguin_summary <- penguins_grouped %>%
summarise(mean_flipper = mean(flipper_length_mm, na.rm=TRUE))
```

Now the first row of our summary table shows us the mean flipper length (in mm) for female Adelie penguins. There are eight rows in total, six unique combinations and two rows where the sex of the penguins was not recorded(`NA`)

8.7.4.1 using group_by with mutate

When `mutate` is used with a grouped object, the calculations occur by 'group'. Here's an example:

```
centered_penguins <- penguins %>%
  group_by(sex, species) %>%
  mutate(flipper_centered = flipper_length_mm-mean(flipper_length_mm, na.rm=TRUE))
```

Here we are calculating a **group centered mean**, this new variable contains the *difference* between each observation and the mean of whichever group that observation is in.

8.7.5 remove group_by

On occasion we may need to remove the grouping information from a dataset. This is often required when we string pipes together, when we need to work using a grouping structure, then revert back to the whole dataset again

Look at our grouped dataframe, and we can see the information on groups is at the top of the data:

```
# A tibble: 344 x 10
# Groups:   sex, species [8]
   species island bill_length_mm bill_depth_mm flipper_length_~ body_mass_g
```

```
   <chr>   <chr>           <dbl>         <dbl>            <dbl>       <dbl>
 1 Adelie  Torge~           39.1          18.7              181        3750
 2 Adelie  Torge~           39.5          17.4              186        3800
 3 Adelie  Torge~           40.3          18                195        3250
```

```
centered_penguins %>%
  ungroup()
```

Look at this output - you can see the information on groups has now been removed from the data.

Chapter 9

Dealing with data part 2: Week Nine

9.1 Expanding the data toolkit

In this chapter we go through some more miscellaneous topics around dealing with data.

9.2 Pipes

As you have seen we often link our dplyr verbs together with `pipes`. Using one function after another e.g. `mutate` then `group_by` and perhaps `summarise` The pipe %>% allows us to make "human-readable" an logical strings of instructions.

The two below examples of pseudocode would be read identically by R - but which one is easier for you to understand?

%>%

```
leave_house(get_dressed(get_out_of_bed(wake_up(me, time =
"8:00"), side = "correct"), pants = TRUE, shirt = TRUE), car
= TRUE, bike = FALSE)
```

```
me %>%
  wake_up(time = "8:00") %>%
  get_out_of_bed(side = "correct") %>%
  get_dressed(pants = TRUE, shirt = TRUE) %>%
  leave_house(car = TRUE, bike = FALSE)
```

One way we can try and breakdown the complexity of multiple nested brackets is to make lots of "intermediate" R objects:

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

penguins_grouped_full <- drop_na(penguins_grouped, sex)

penguins_flipper_length <- summarise(penguins_grouped_full, mean=mean(flipper_length_mm

penguins_flipper_length
```

> **Note - nothing wrong with this, but look at how many R objects are cluttering up your environment tab which you will probably never use again!

Or…

```
penguins %>%
  group_by(., species, sex) %>%
  drop_na(., sex) %>%
  summarise(., mean=mean(flipper_length_mm))
```

> **Note - check for yourself whether the outcome is identical

The pipe is becoming increasingly popular - because it makes code so much more readable. The newest versions of base R (4.1 and above) have even incorporate their own version of a pipe `|>` (https://michaelbarrowman.co.uk/post/the-new-base-pipe/).

But how does the pipe actually work? Well whenever you include a pipe in your code, it signals to take everything on the left-hand side of the pipe and place it as an "argument" in the code which then comes on the right hand side. The . in the code above shows you the placeholder where the argument is placed.

When you use the pipe (as you have been doing previously), you don't actually need to include the . for most functions. If you don't put it in, the pipe will simply place it into the *first argument* position. For this reason you will see that all dplyr functions have the first argument as the data.

```
penguins %>%
  group_by(species, sex) %>%
  drop_na(sex) %>%
  summarise(mean=mean(flipper_length_mm))
```

9.3 Strings

Datasets often contain words, and we call these words "strings". Often these aren't quite how we want them to be, but we can manipulate these as much as we like. Functions in the package `stringr` Wickham (2019), are fantastic. And the number of different types of manipulations are endless

```
str_replace_all(names(penguins), c("e"= "E"))
```

9.3.1 separate

Sometimes a string might contain two pieces of information in one. This does not confirm to our tidy data principles. But we can easily separate the information with `separate` from `tidyr` Wickham (2021)

First we produce some made-up data

```
df <- tibble(label=c("a-1", "a-2", "a-3"))
#make a one column tibble
df
```

```
df %>%
  separate(label, c("treatment", "replicate"), sep="-")
```

We started with one variable called `label` and then split it into two variables, `treatment` and `replicate`, with the split made where `-` occurs. The opposite of this function is `unite()`

9.3.2 More stringr

Check out (https://stringr.tidyverse.org/index.html) Wickham (2019)

```
penguins %>%
  mutate(species=str_to_upper(species))
```

```
penguins %>%
  mutate(species=str_remove_all(species, "e"))
```

We can also trim leading or trailing empty spaces with `str_trim`. These are often problematic and difficult to spot e.g.

```
df2 <- tibble(label=c("penguin", " penguin", "penguin "))
df2
```

We can easily imagine a scenario where data is manually input, and trailing or leading spaces are left in. These are difficult to spot by eye - but problematic because as far as R is concerned these are different values. We can use the function `distinct` to return the names of all the different levels it can find in this dataframe.

```
df2 %>%
  distinct()
```

If we pipe the data throught the `str_trim` function to remove any gaps, then pipe this on to `distinct` again - by removing the whitespace, R now recognised just one level to this data.

```
df2 %>%
  mutate(label=str_trim(label, side="both")) %>%
  distinct()
```

A quick example of how to extract partial strings according to a pattern is to use `grepl`. Combined with `filter` it is possible to subset dataframe by searching for all the strings that match provided information, such as all the penguin IDs that start with "N1"

```
penguins %>%
  filter(grepl("N1", penguin_id)) %>%
  distinct(penguin_id)
```

9.4 Dates and times

Dates and times are difficult.

There's a lot of different ways to write the same date

13-10-2019

10-13-2019

13-10-19

13th Oct 2019

2019-10-13

This variability makes it difficult to tell our software how to read the information, luckily we can use `lubridate` Spinu et al. (2021). We can tell R the order of our date (or time) data.

Back in the last chapter (8.4), we used `mutate` and `lubridate` to convert date data which had been parsed as a string, into date format with the function `dmy()`. Depending on how we interpret the date ordering in a file, we can use `ymd`, `ydm`, `mdy` etc.

If you get a warning that some dates could not be parsed, then you might find the date date has been inconsistently entered into the dataset.

```
penguins %>%
  select(date, date_proper) %>%
  head()
```

Once we have established our date data, we are able to perform calculations. Such as the date range across which our data was collected. Try this with our variable in string form and it will provide the alphabetical order instead.

```
penguins %>%
  summarise(min_date=min(date_proper),
            max_date=max(date_proper))
```

9.4.1 Calculations with dates

How many times was each penguin measured, and across what total time period?

```
penguins %>%
  group_by(penguin_id) %>%
  summarise(min=min(date_proper),
            max=max(date_proper),
            difference = max-min,
            n=n())
```

Cool we can also convert intervals such as days into weeks, months or years with `dweeks(1)`, `dmonths(1)`, `dyears(1)`.

As with all cool functions, you should check out the RStudio cheat sheet for more information. Date type data is common in datasets, and learning to work with it is a uesful skill.

```
penguins %>%
  group_by(penguin_id) %>%
```

```
  summarise(min=min(date_egg_proper),
            max=max(date_egg_proper),
            difference = (max-min)/dyears(1),
            n=n()) %>%
  arrange(desc(difference))
```

9.5 Pivot

In the previous chapters we discussed the principles of tidy data (3.3.1), tidy data is easy to analyse, but not always the format you will find data in.

The most likely format you will find data in is a "wide format", where we have columns of related data. Luckily `tidyr` contains a number of **pivot** functions

wide

id	x	y	z
1	a	c	e
2	b	d	f

Let's start with some dummy data - we will make a dataframe of three penguins and the number of three types of prey they are observed to eat over a week:

```
penguin_id <- c("N1A1","N1A2","N2A1")
krill <- c(70,20,43)
fish <- c(5, 19,4)
squid <- c(11,5,0)


df3 <- tibble(penguin_id, krill, fish, squid)
df3
```

This dataset is perfectly legible, but does not confirm to tidy data principles, each row is not a unique observation, but instead contains three observations of types of prey eaten. Most of the time, we can make data "tidy" but pivoting wide data into a long format. In this example we should rearrange the data to produce a longer dataframe with fewer columns. We should make a new column for prey "type" and a column for "amount"

```
df3_long <- df3 %>%
  pivot_longer(cols=(krill:squid), names_to = "prey_type", values_to = "consumption_pe
df3_long
```

It then becomes extremely easy to plot this data:

```
df3_long %>%
  ggplot(aes(x=penguin_id, y=consumption_per_week, fill=prey_type))+
  geom_bar(stat="identity", position=position_dodge())
```

Once again, there's an RStudio cheatsheet for this, so check it out!

You may very occasionally need to use `pivot_wider()` to generate tidy data, but it's unusual, the above scenario is the most likely messy data you will encounter!

9.6 Summing up

9.6.1 What we learned

You have learned:

The most common data manipulation functions you will need to import and tidy data, and the way in which they operate.

- How to create new columns as products of functions or calculations
- How to select and filter data to access the subsets you are interested in

- How to apply functions to data groups
- How the pipe works, and why we use it
- How to deal with data subtypes like strings and dates
- How to reshape data with pivot

You have used `tidyverse` and the functions made available from:

- `dplyr` Wickham et al. (2021c)
- `tidyr` Wickham (2021)

You have also used two packages, which are also installed as part of the tidyverse, but have to be called with separate library() functions

- `lubridate` Spinu et al. (2021)
- `stringr` Wickham (2019)

9.6.2 Further Reading, Guides and tips

R Cheat Sheets

Chapter 10

Deeper data insights part 1: Week Ten

In these last chapters on generating data insights we are going to cover

- Types of variable
- Exploring numeric variables
- Exploring categorical variables
- Dealing with missing data

Before we dive into this it is important to understand where this data has come from, is it a single study? Is it a lab experiment, fieldwork survey, exploratory work or from a carefully designed study?

You should also play close attention to the data, and remind yourself **frequently** how many variables do you have and what are their names? How many rows/observations do you have?

10.1 Variables

10.1.1 Numerical

You **should** already be familiar with the concepts of numerical and categorical data. **Numeric** variables have values that describe a measure or quantity. This is also known as quantitative data. We can subdivide numerical data further:

- **Continuous numeric variables.** This is where observations can take any value within a range of numbers. Examples might include body mass (g), age, temperature or flipper length (mm). While in theory these values can have any numbers, within a dataset they are likely bounded (set within

a minimum/maximum of observed or measurable values), and the accuracy may only be as precise as the measurement protocol allows.

- **Discrete numeric variables** Observations are numeric but restricted to *whole values* e.g. 1,2,3,4,5 etc. These are also known as **integers**. Discrete variables could include the number of individuals in a population, number of eggs laid etc. Anything where it would make no sense to describe in fractions e.g. a penguin cannot lay 2 and a half eggs. Counting!

10.1.2 Categorical

Values that describe a characteristic of data such as 'what type' or 'which category'. Categorical variables are mutually exclusive - one observation should not be able to fall into two categories at once - and should be exhaustive - there should not be data which does not *fit* a category (not the same as NA - not recorded). Categorical variables are qualitative, and often represented by non-numeric values such as words. It's a bad idea to represent categorical variables as numbers (R won't treat it correctly). Categorical variables can be defined further as:

- **Ordinal variables** Observations can take values that can be logically ordered or ranked. Examples include - activity levels (sedentary, moderately active, very active); size classes (small, medium, large).
- **Nominal variables** Observations that can take values that are not logically ordered. Examples include Species or Sex in the Penguins data.

It is important to order Ordinal variables in the their logical order value when plotting data visuals or tables. Nominal variables are more flexible and could be ordered in whatever pattern works best for your data (perhaps you could order them according to the values of another numeric variable).

Don't use numbers to describe categorical information e.g. (Adelie = 1, Gentoo =2, Chinstrap = 3). This can be done, but it isn't usually very sensible. It's clearer to use the words themselves, and helps when making tables and graphs later.

It's easy to assign levels to categorical data:

Example:

penguins <- penguins %>% mutate(species = factor(species, levels = c("Adelie", "Gentoo", "Chinstrap")))

10.2 Understanding Numerical variables

Let's take a look at some of our variables

```
glimpse(penguins)
```

```
## Rows: 344
## Columns: 9
## $ species           <chr> "Adelie", "Adelie", "Adelie", "Adelie", "Adelie", "A~
## $ island            <chr> "Torgersen", "Torgersen", "Torgersen", "Torgersen", ~
## $ bill_length_mm    <dbl> 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, 34.1, ~
## $ bill_depth_mm     <dbl> 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, 18.1, ~
## $ flipper_length_mm <dbl> 181, 186, 195, NA, 193, 190, 181, 195, 193, 190, 186~
## $ body_mass_g       <dbl> 3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, ~
## $ sex               <chr> "male", "female", "female", NA, "female", "male", "f~
## $ date              <chr> "11/11/2007", "11/11/2007", "16/11/2007", "16/11/200~
## $ penguin_id        <chr> "N1A1", "N1A2", "N2A1", "N2A2", "N3A1", "N3A2", "N4A~
```

We can see that bill length contains numbers, and that many of these are fractions, but only down to 0.1mm. By comparison body mass all appear to be discrete number variables. Does this make body mass an integer? The underlying quantity (bodyweight) is clearly continuous, it is clearly possible for a penguin to weigh 3330.7g but it might *look* like an integer because of the way it was measured. This illustrates the importance of understanding the the type of variable you are working with - just looking at the values isn't enough.

On the other hand, how we choose to measure and record data *can* change the way it is presented in a dataset. If the researchers had decided to simply record small, medium and large classes of bodyweight, then we would be dealing with ordinal categorical variables. These distinctions can become less clear if we start to deal with multiple classes of ordinal categories - for example if the researchers were measuring body mass to the nearest 10g. It might be reasonable to treat these as integers…

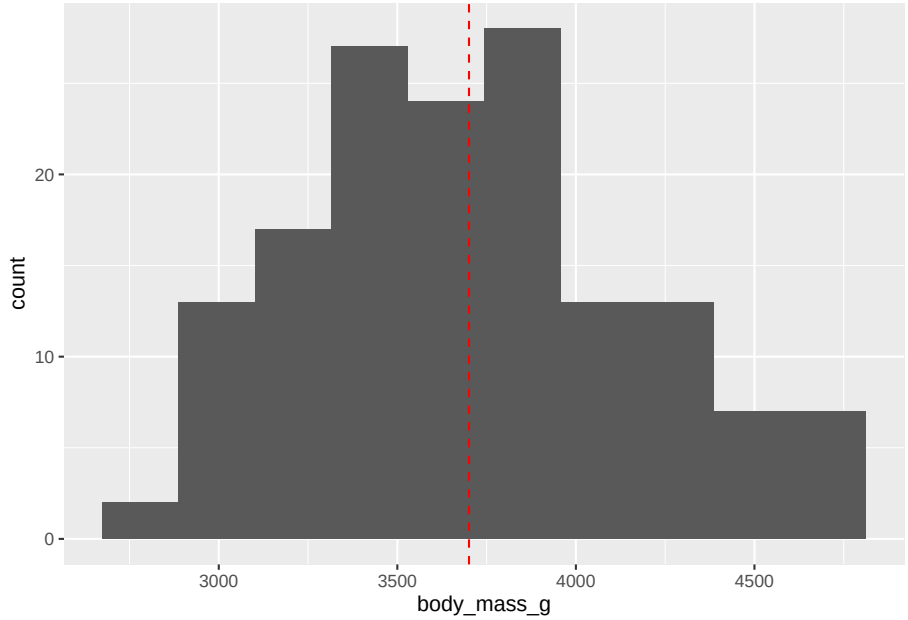
10.2.1 Graphing a numeric variable

```
adelie_penguins <- penguins %>%
  filter(species=="Adelie")

adelie_summary <- adelie_penguins %>%
  summarise(mean=mean(body_mass_g,
                      na.rm=TRUE))

ggplot()+
  geom_histogram(data= adelie_penguins,
                 aes(x=body_mass_g),
                 bins=10)+ #remember it is a good idea to try multiple bins of data
  geom_vline(data=adelie_summary,
```

```
            aes(xintercept=mean),
            colour="red",
            linetype="dashed")
```



Using this distribution, we can see that the data appears to fit a normal/gaussian distribution - mean body mass is slightly under 3750g. Penguins smaller than 3000g are rare.

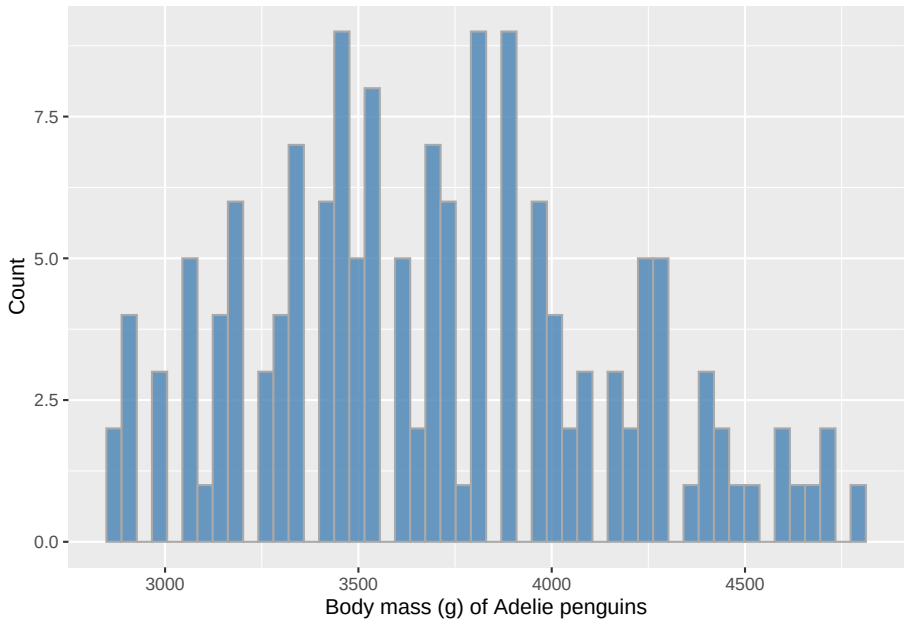
10.2.2 Insights about body mass

The histogram gives us a nice summary of the sample distribution of the body_mass_g variable. It reveals (1) the most common values, (2) the range of values and (3) the shape of the distribution. Remember it's a very good idea to play with the number of bins in order to see whether this changes the shape of the histogram.

Let's construct the histogram again with 30 bins. As well we will make some other tweaks.

```
ggplot()+
  geom_histogram(data=adelie_penguins,
                 aes(x=body_mass_g),
                 bins=50, # fifty bins
                 fill="steelblue",
```

```
                 colour="darkgrey",
                 alpha=0.8)+
  labs(x="Body mass (g) of Adelie penguins",
       y = "Count")
```



Changing the colours and labels is purely aesthetic, and not needed *at all* for data exploration, but look how pretty!

What is important is that increasing the number of bins indicates that their *might* be two peaks in our data. We should think carefully about the other variables (probably categorical) that could be used to subset our data to investigate this. This requires careful thought about our data - in this instance - it seems sensible to try and see whether splitting the data by sex accounts for this

```
adelie_penguins %>%
  drop_na(sex) %>%
ggplot()+
  geom_histogram(aes(x=body_mass_g,
                     fill=sex),
                 bins=50, # fifty bins
                 colour="darkgrey",
                 alpha=0.8,
                 position="identity")+
  labs(x="Body mass (g) of Adelie penguins",
```

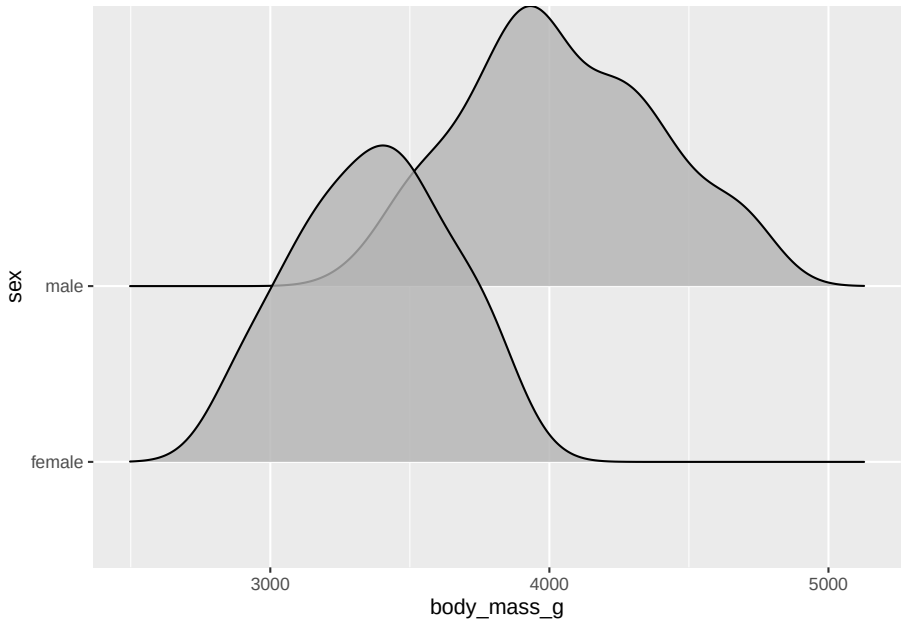
```
       y = "Count")
```

Let's finish this section by looking at density. This is sensible to use when we have "large datasets", and also allows us to make comparisons between groups with different sample sizes.

```
adelie_penguins %>%
  drop_na(sex) %>%
ggplot()+
  geom_density(aes(x=body_mass_g,
                       fill=sex),
                   colour="darkgrey",
                   alpha=0.8,
                   position="identity")+
  labs(x="Body mass (g) of Adelie penguins",
       y = "Count")
```

Or we can use one of my favourite packages `ggridges` Wilke (2021) which let's us separate out different groups along the y-axis.

```
adelie_penguins %>%
  drop_na(sex) %>%
ggplot()+
  ggridges::geom_density_ridges(aes(x=body_mass_g,
                                    y=sex),
                                    alpha=0.8)
```



10.3 Descriptive statistics

We have, so far, been spending our time describing the properties of data by examining graphs. Now we can start to build accurate and specific terms to the descriptions of our data.

- **central tendency** describes the typical (central) value of a distribution. The most well known description of central tendency is the arithmetic mean, however you should be comfortable with the idea that the median may be a better representation of the central tendency for some data distributions

- **dispersion** describes how a distribution is spread out. Dispersion measures the variability or scatter of a variable. If one distribution is more dispersed than another, this means that in some sense it encompasses a wider range of values. Basic statistics courses often tend to focus on variance adn the standard deviation as two ways to measure/sumamrise dispersion. However, the interquartile range is another method often used in exploratory analysis.

In the next section we will use the median and interquartile ranges as effective measures of central tendency and dispersion - we will use graphics for this - and more specifically look at boxplots.

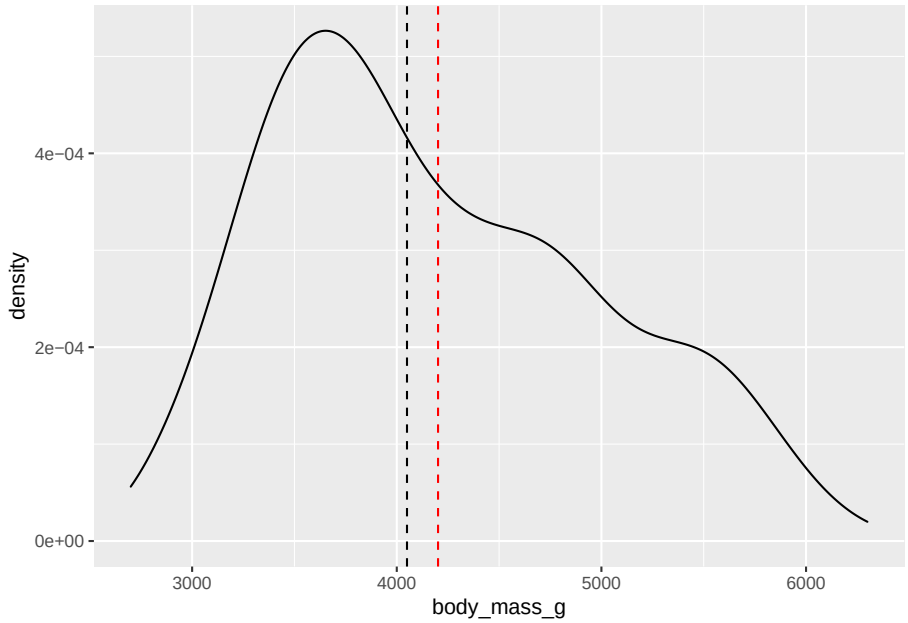
10.3.1 Central tendency

We can find both the mean and median easily with the summarise function.

```
penguin_body_mass_summary <- penguins %>%
  summarise(mean_body_mass=mean(body_mass_g, na.rm=T),
            median_body_mass=median(body_mass_g, na.rm=T))

penguin_body_mass_summary
```

```
## # A tibble: 1 x 2
##   mean_body_mass median_body_mass
##            <dbl>            <dbl>
## 1          4202.             4050
```



If we do this for the entire penguins dataset - we can clearly see that the mean value has bee significantly "right-shifted" by the long tail of the data distribution. However,this is much less apparent for the median. In this way we can say that the median is less sensitive to the distribution of the data than the mean is.

10.3.2 Dispersion

Dispersion (how spread out the data is) is an important component towards understanding any numeric variable. Important measures for statistics are **variance** and **standard deviation**. These are both non-negative, smaller values indicate observations tend to be similar in value, while high values indicate these

observations are more spread out. We can quickly calculate these with `var` and `sd` - but they probably aren't the best place to *start* exploring your data from.

Variance is not an intuitive measure - calculated from squaring the deviation of each data point from the sample mean - it gives no overall insight into the shape of the distribution, and it is on a different scale to the original measurements. Standard deviation is the square root of the variance. This means it is on the same scale as the observation, making it easier to interpret, but just like variance it doesn't provide much insight into the overall shape of the distribution, and like the mean can be affected by outliers.

When trying to understand data it can be simpler and easier to use a measure that does not suffer from these issues outlined above. Instead we can use the **interquartile range**.

The interquartile range (IQR) is the range that contains the "middle 50%" of our data sample. This is given as the difference between the third and first quartiles. The reason we like to use the IQR is that the more spread out the data is, the larger the IQR. It will also indicate the shape of the distribution and is less affected by outliers than variance.

We can use the IQR function to find the interquartile range of the body mass variable

```
penguins %>%
  summarise(IQR_body_mass = IQR(body_mass_g, na.rm=TRUE))
```

The IQR is also useful when applied to the summary plots 'box and whisker plots'. We can also calculate the values of the IQR margins, and add labels with `scales` Wickham and Seidel (2020).

```
penguins %>%
  summarise(q_body_mass = quantile(body_mass_g, c(0.25, 0.5, 0.75), na.rm=TRUE),
            quantile = scales::percent(c(0.25, 0.5, 0.75))) # scales package allows easy converti
```

```
# A tibble: 3 x 2
  q_body_mass quantile
        <dbl> <chr>
1        3550 25%
2        4050 50%
3        4750 75%
```

We can see for ourselves the IQR is obtained by subtracting the body mass at tht 75% quantile from the 25% quantile (4750-3550 = 1200).

10.3.3 Visualising dispersion

```
penguins %>%
  ggplot()+
  geom_boxplot(aes(x="",
                   y= body_mass_g),
               fill="darkorange",
               colour="steelblue",
               width=0.4)+
  labs(x= "Bodyweight",
       y = "Mass (g)")+
  theme_minimal()
```

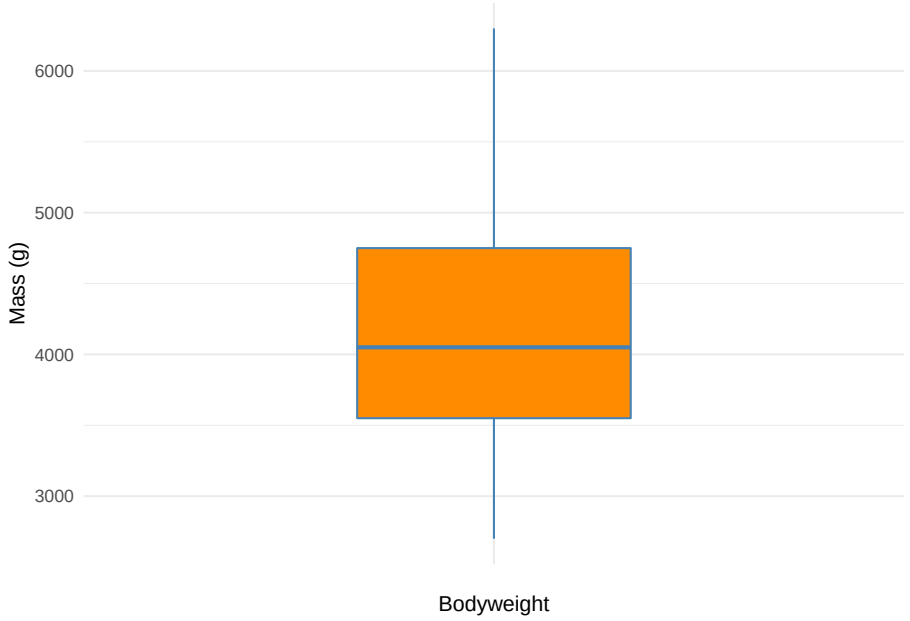


Figure 10.1: A boxplot of the body mass variable showing the medan and IQR

> **Note - we forced ggplot2 to hide the tick mark label on the x axis by coding a dummy label with x = " "

We now have several compact representations of the body_mass_g including a histogram, boxplot and summary calculations. You can *and should* generate the same summaries for your other numeric variables. These tables and graphs provide the detail you need to understand the central tendency and dispersion of numeric variables.

10.3.4 Combining histograms and boxplots

```
library(patchwork) # put this at the TOP of your script

penguins_na_sex <- penguins %>%
  drop_na(sex)

colours <-  c("darkorange", "cyan") # set colour scheme here to save on repeating code
lims <- c(3000,6000) # set axis limits here to save on repeating code

  p1 <- ggplot(data = penguins_na_sex,
    aes(x = species,
             y = body_mass_g,
             fill = sex))+
      geom_boxplot()+
    scale_fill_manual(values = colours)+
    scale_y_continuous(limits=lims)+
    labs(x="",
         y="")+
        coord_flip()+ # rotate box plot 90 degrees
    theme_minimal()+
    theme(legend.position="none")

  p2 <- ggplot(data = penguins_na_sex,
          aes(x = body_mass_g,
              y = species,
              fill = sex))+
    ggridges::stat_density_ridges(quantile_lines = TRUE)+
        scale_fill_manual(values = colours)+
    scale_x_continuous(limits=lims)+
    labs(y="",
         x = "Body Mass (g)")+
    theme_minimal()

  (p1/p2) # patchwork command to layer one plot above the other
```

10.3.5 Missing values

We first met `NA` back in Chapter 3.7.8 and you will hopefully have noticed, either here or in those previous chapters, that missing values `NA` can really mess up our calculations. There are a few different ways we can deal with missing data:

- `drop_na()` on everything before we start. This runs the risk that we lose **a lot** of data as *every* row, with an NA in *any column* will be removed

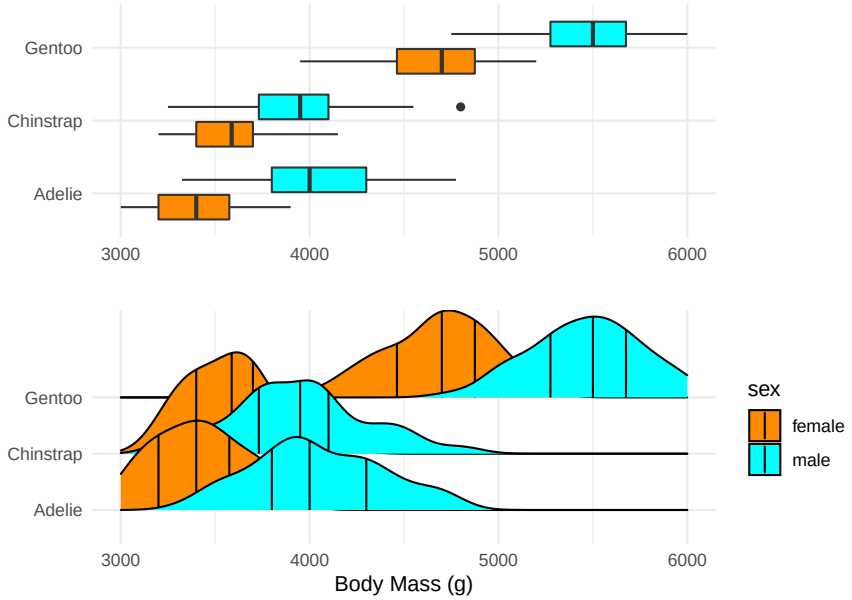


Figure 10.2: This figure shows the use of patchwork to combine two ggplots, demonstrating the boxplot and geom density figures show the same distributions of the data

- `drop_na()` on a particular variable. This is fine, but we should approach this cautiously - if we do this in a way where we write this data into a new object e.g. `penguins <- penguins %>% drop_na(body_mass_g)` then we have removed this data forever - perhaps we only want to drop those rows for a specific calculation - again they might contain useful information in other variables.
- `drop_na()` for a specific task - this is a more cautious approach **but** we need to be aware of another phenomena. Is the data **missing at random**? You might need to investigate *where* your missing values are in a dataset. Data that is truly **missing at random** can be removed from a dataset without introducing bias. However, if bad weather conditions meant that researchers could not get to a particular island to measure one set of penguins that data is **missing not at random** this should be treated with caution. If that island contained one particular species of penguin, it might mean we have complete data for only two out of three penguin species. There is nothing you can do about incomplete data other than be aware that data not missing at random could influence your distributions.

10.4 Categorical variables

Ok that was a lot of information - well done - have some praise.

```
praise::praise()
# have some praise
```

Now let's look at categorical variables -remember these can be **ordinal** or **nominal**. We don't have anything we would classify as ordinal data in the penguins dataset.

We can look at species with the function `distinct`:

```
penguins %>%
  distinct(species)
```

```
# A tibble: 3 x 1
  species
  <chr>
1 Adelie
2 Gentoo
3 Chinstrap
```

We can clearly see there would be no sense to applying an order to these three categories, we can order them in whatever way suits best when presenting our data.

10.4.1 Summaries

```
penguins %>%
  count(species, sort=TRUE)
```

```
# A tibble: 3 x 2
  species       n
  <chr>     <int>
1 Adelie      152
2 Gentoo      124
3 Chinstrap    68
```

It might be useful for us to make some quick data summaries here

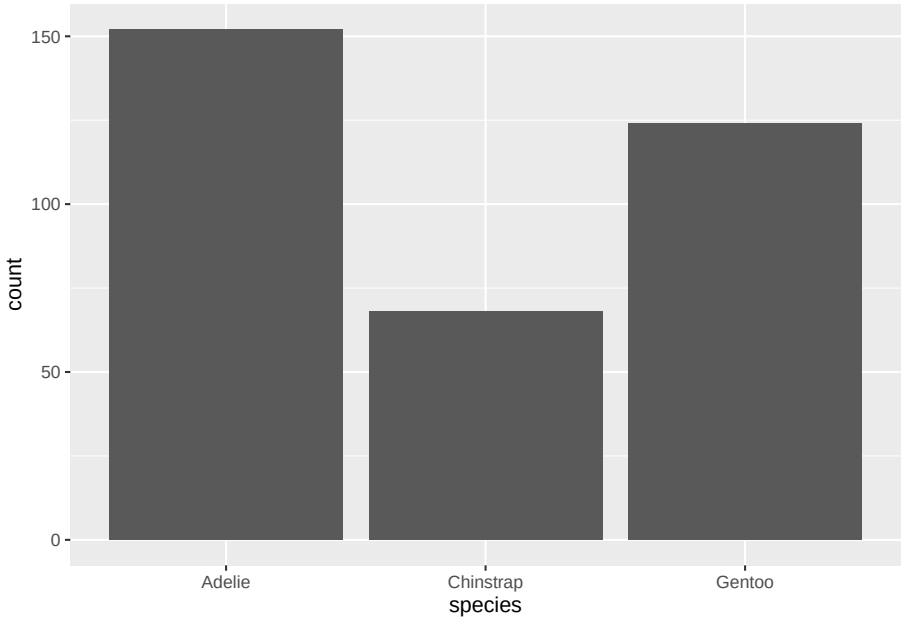
```
prob_obs_species <- penguins %>%
  count(species, sort=TRUE) %>%
  mutate(prob_obs = n/sum(n))

prob_obs_species
```

```
## # A tibble: 3 x 3
##   species       n prob_obs
##   <chr>     <int>    <dbl>
## 1 Adelie      152    0.442
## 2 Gentoo      124    0.360
## 3 Chinstrap    68    0.198
```

So about 44% of our sample is made up of observations from Adelie penguins. When it comes to making summaries about categorical data, that's about the best we can do, we can make observations about the most common categorical observations, and the relative proportions.

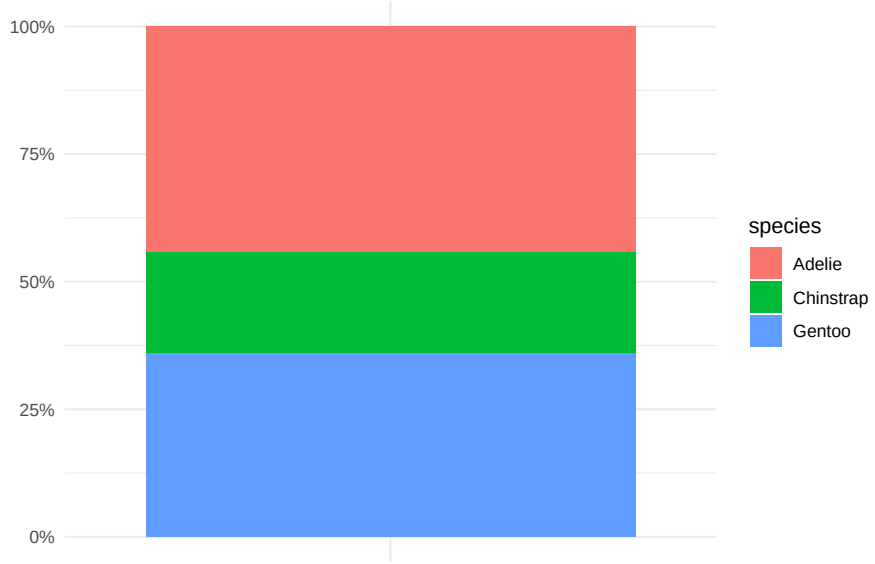
```
penguins %>%
  ggplot()+
  geom_bar(aes(x=species))
```



This chart is ok - but can we make anything better?

We could go for a stacked bar approach

```
penguins %>%
  ggplot(aes(x="",
             fill=species))+ # specify fill = species to ensure colours are defined by species
  geom_bar(position="fill")+ # specify fill forces geom_bar to calculate percentages
  scale_y_continuous(labels=scales::percent)+ #use scales package to turn y axis into percentages
  labs(x="",
       y="")+
  theme_minimal()
```

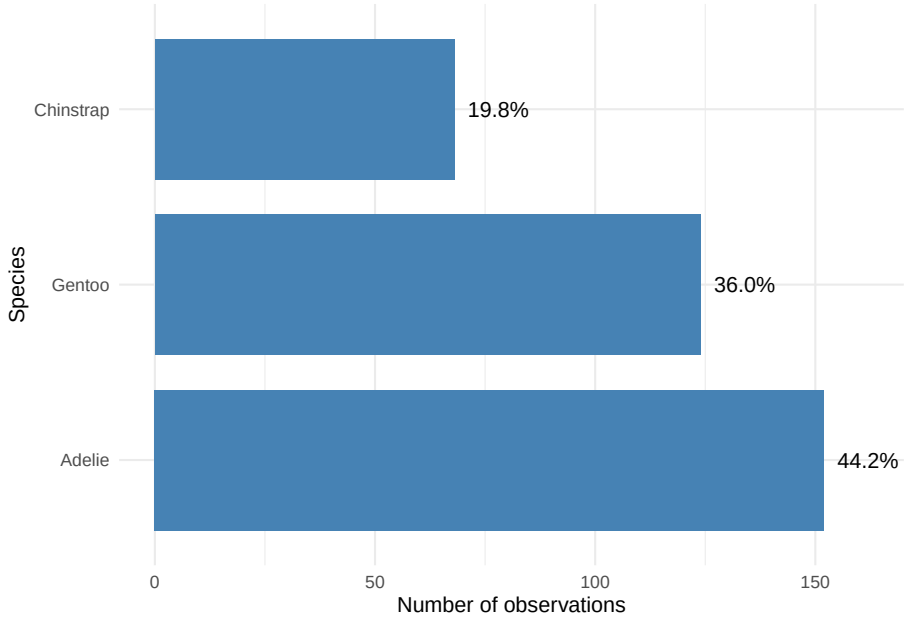


This graph is OK but not great, the height of each section of the bar represents the relative proportions of each species in the dataset, but this type of chart becomes increasingly difficult to read as more categories are included. Colours become increasingly samey,and it is difficult to read where on the y-axis a category starts and stops, you then have to do some subtraction to work out the values.

The best graph is then probably the first one we made - with a few minor tweak we can rapidly improve this.

```
penguins %>%
  mutate(species=factor(species, levels=c("Adelie",
                                          "Gentoo",
                                          "Chinstrap"))) %>% # set as factor and provi
  ggplot()+
  geom_bar(aes(x=species),
           fill="steelblue",
           width=0.8)+
  labs(x="Species",
       y = "Number of observations")+
  geom_text(data=prob_obs_species,
            aes(y=(n+10),
                x=species,
                label=scales::percent(prob_obs)))+
  coord_flip()+
```

```
  theme_minimal()
```



10.5 Summing up

In this chapter we have really focused on single variables, understanding variable types and their distributions. We learned

- About different types of data
- How to estimate central tendencies
- Dispersions of numeric and categorical variables
- How to visualise metrics

You have primarily used `tidyverse` packages, but also:

- `ggridges` Wilke (2021)
- `scales` Wickham and Seidel (2020)

In the next chapter we will look at generating insights around the relationships between variables. Now is a good time to review earlier chapters such as Chapter 3, and think about how we ask questions of our data.

10.5.1 Reward

Haven't you done well???

Enjoy this stupid reward from ggcats

> **Note - ggcats is sadly not available on CRAN library (yet) if you want to install your own version you can, you just need the package 'remotes' - then to follow the instructions on the ggcats github page.

```
library(Ecdat)
data(incomeInequality)

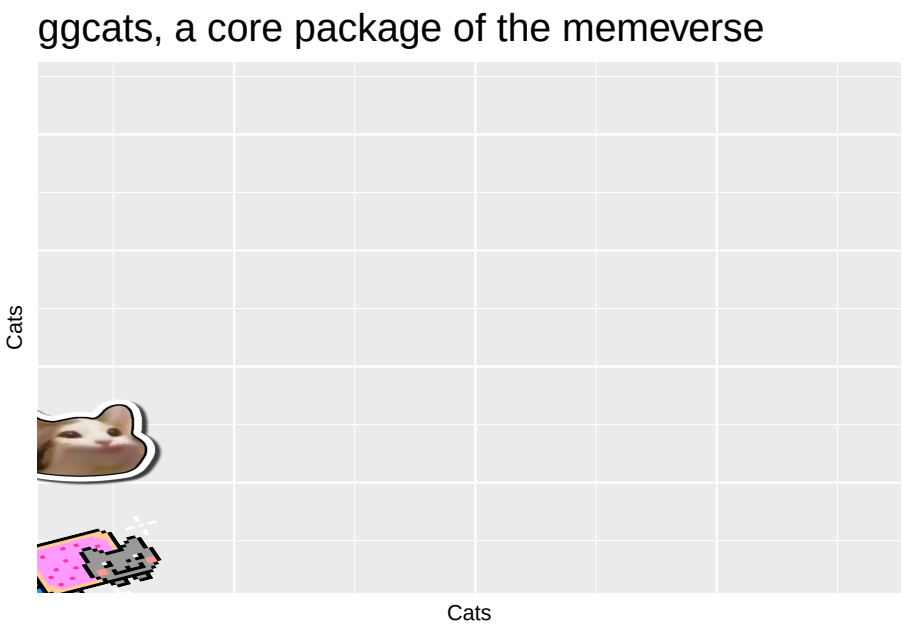
library(ggcats)
library(gganimate)

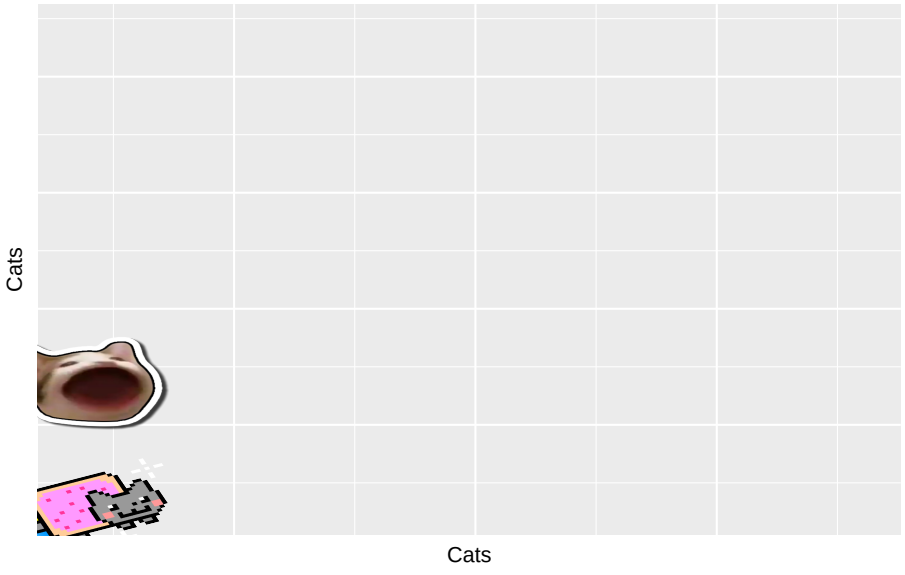
 dat <-
   incomeInequality %>%
   select(Year, P99, median) %>%
   rename(income_median = median,
          income_99percent = P99) %>%
   pivot_longer(cols = starts_with("income"),
                names_to = "income",
                names_prefix = "income_")

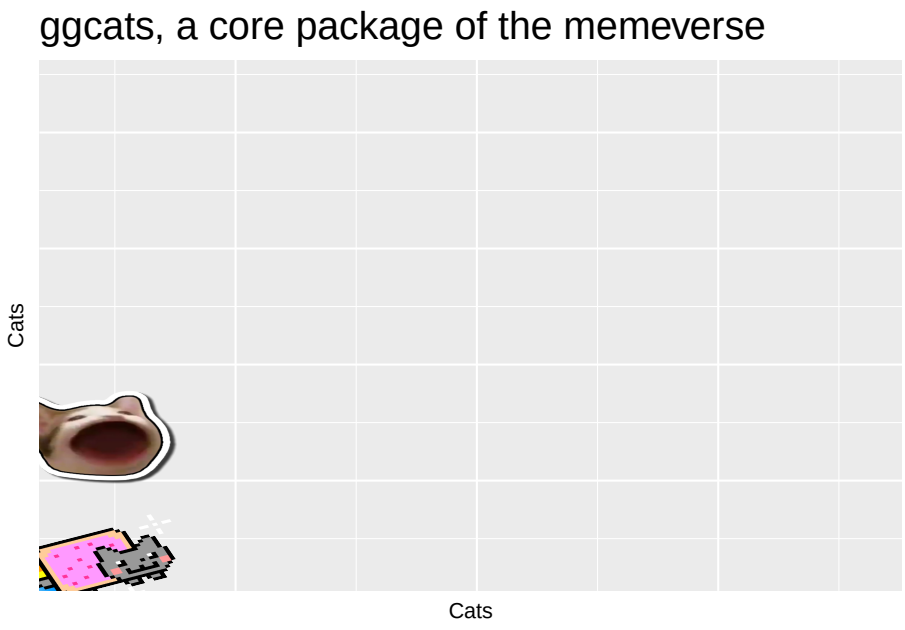
dat$cat <- rep(NA, 132)

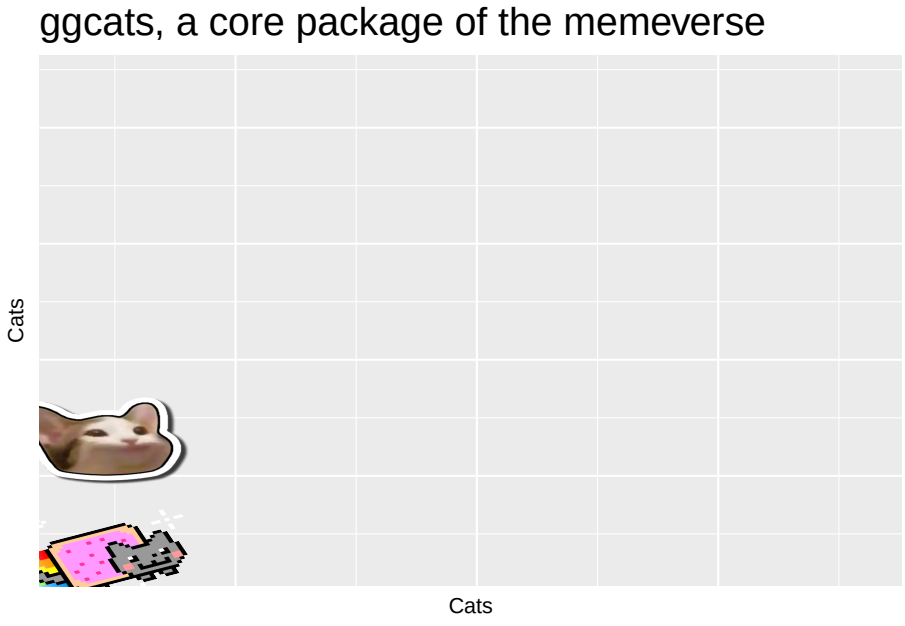
dat$cat[which(dat$income == "median")] <- "nyancat"
dat$cat[which(dat$income == "99percent")] <- rep(c("pop_close", "pop"), 33)

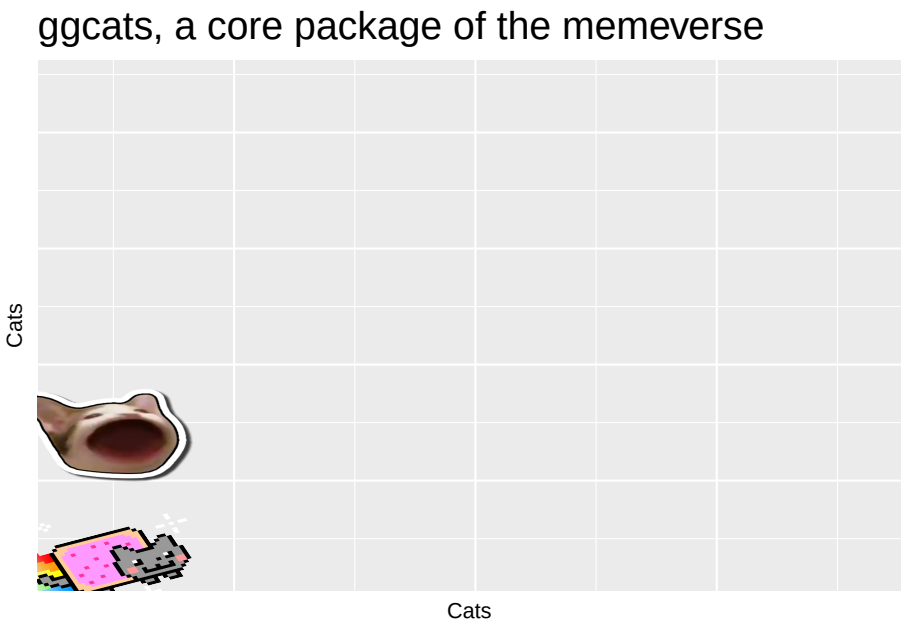
ggplot(dat, aes(x = Year, y = value, group = income, color = income)) +
   geom_line(size = 2) +
   ggtitle("ggcats, a core package of the memeverse") +
   geom_cat(aes(cat = cat), size = 5) +
   xlab("Cats") +
   ylab("Cats") +
   theme(legend.position = "none",
         plot.title = element_text(size = 20),
         axis.text = element_blank(),
         axis.ticks = element_blank()) +
   transition_reveal(Year)
```

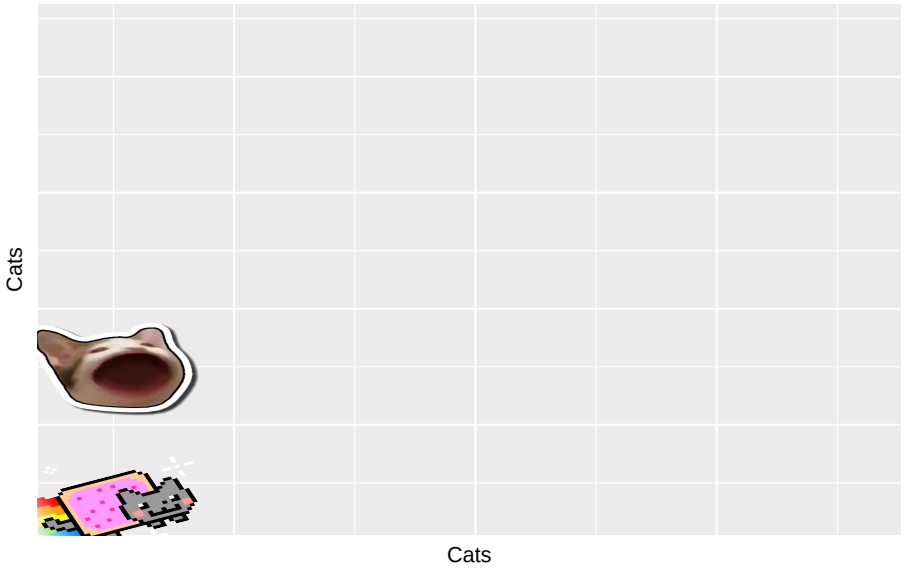

ggcats, a core package of the memeverse
Cats
Cats

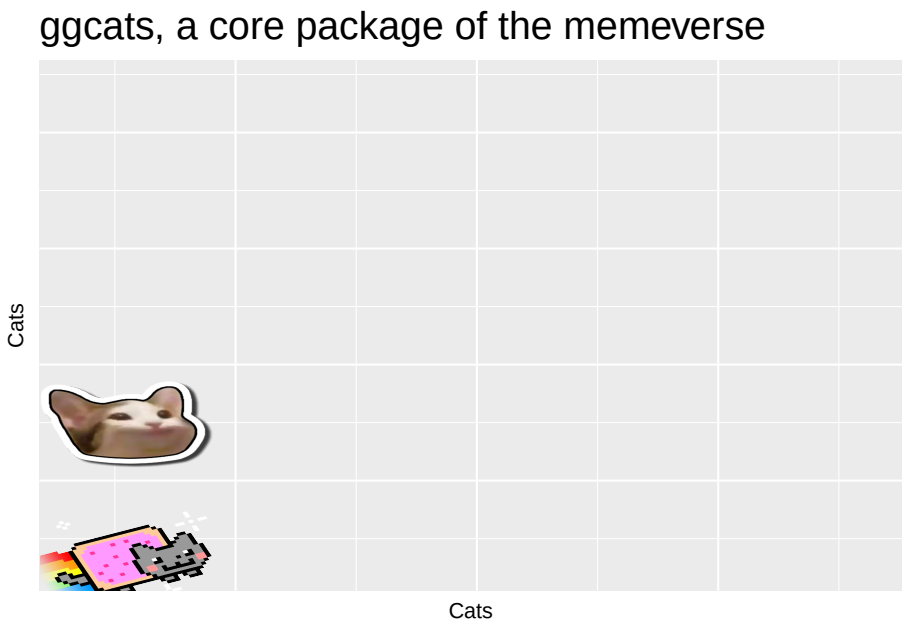

ggcats, a core package of the memeverse
Cats
Cats

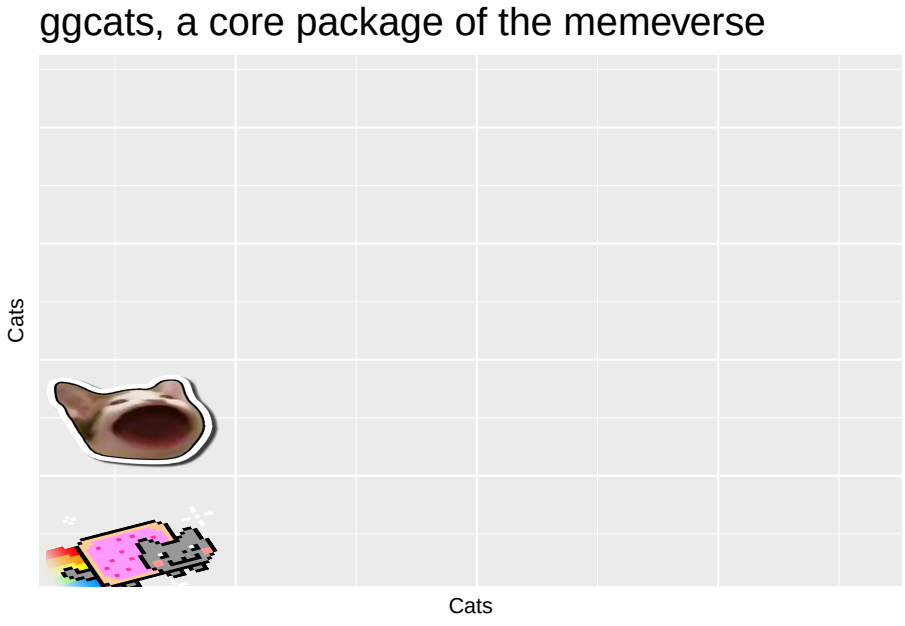

ggcats, a core package of the memeverse
Cats
Cats

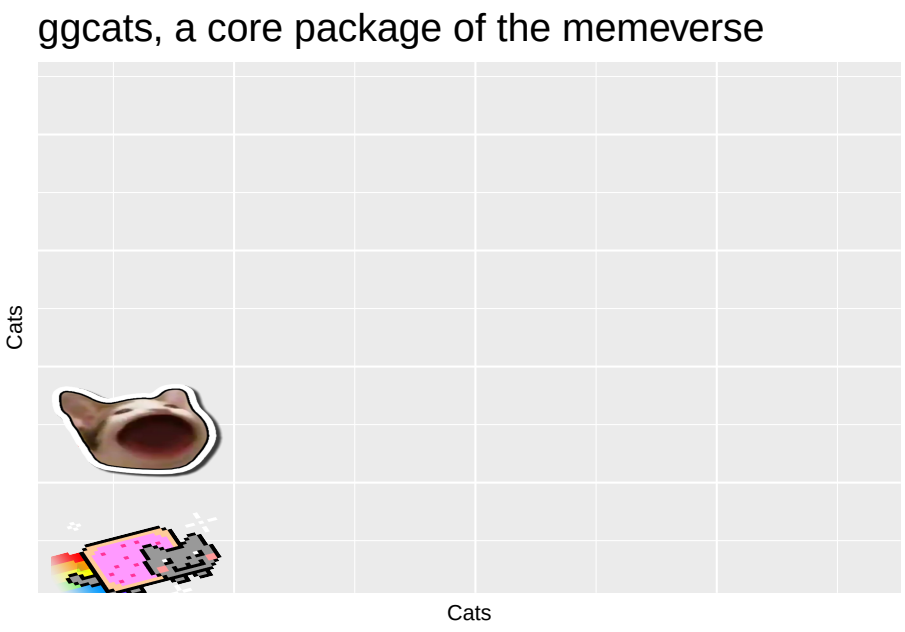

ggcats, a core package of the memeverse
Cats
Cats

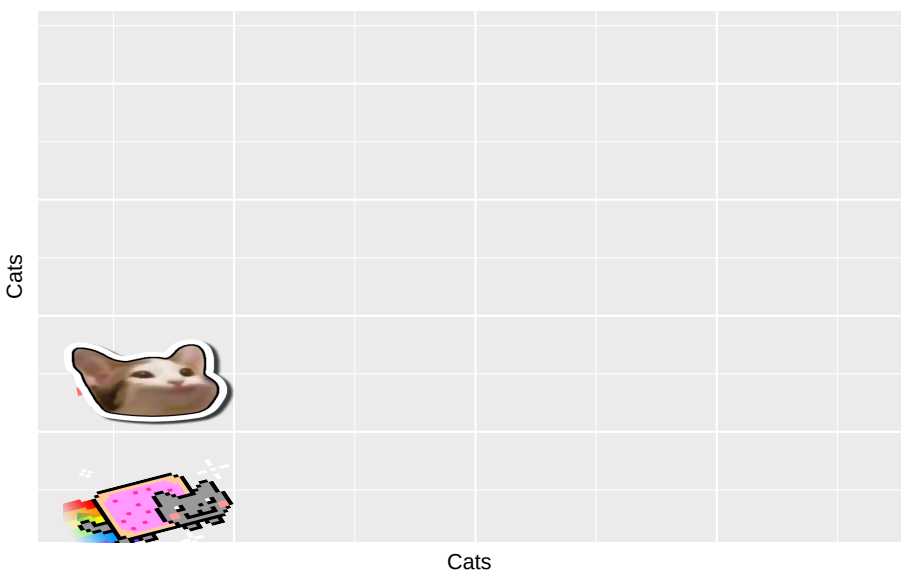

ggcats, a core package of the memeverse
Cats
Cats

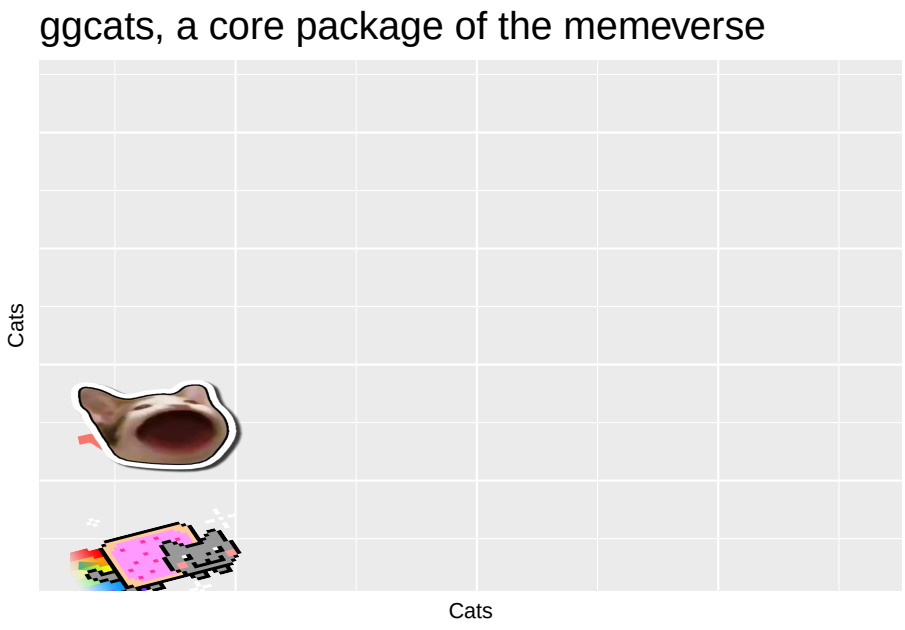

ggcats, a core package of the memeverse
Cats
Cats

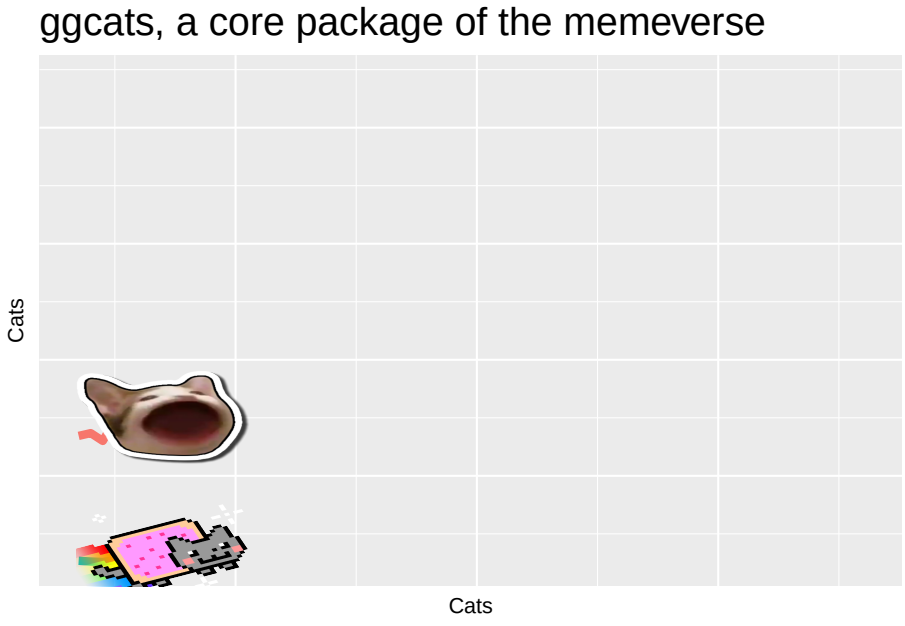

ggcats, a core package of the memeverse
Cats
Cats

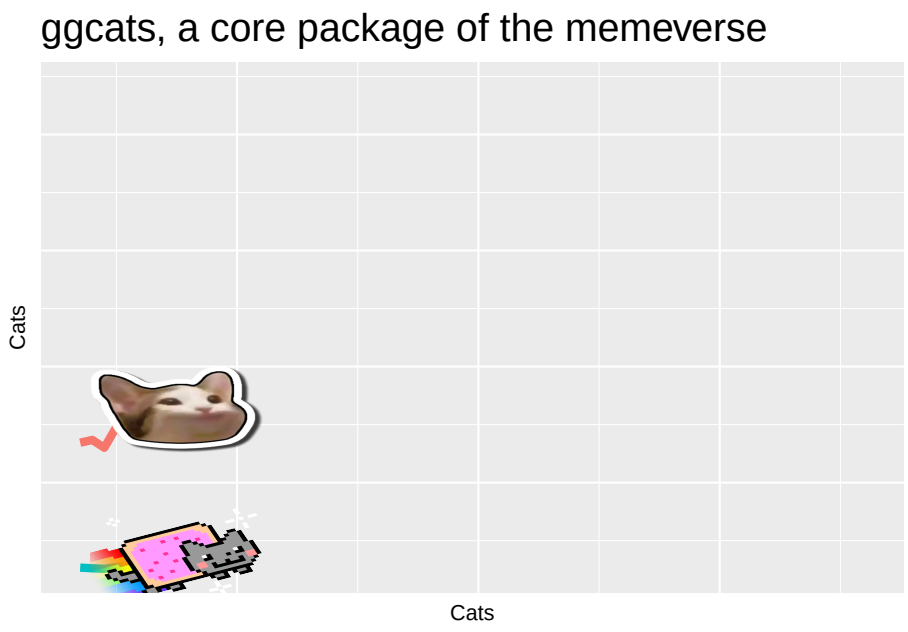

ggcats, a core package of the memeverse
Cats
Cats

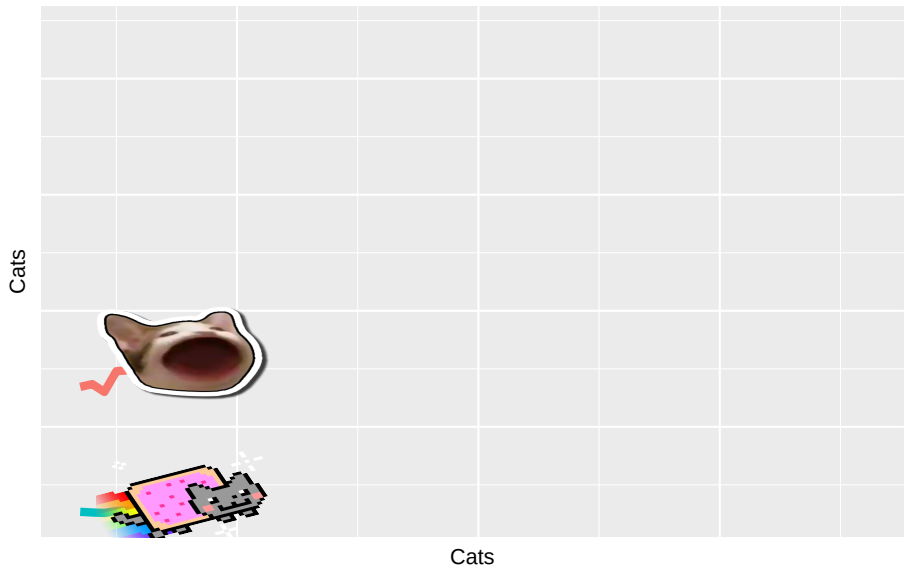

ggcats, a core package of the memeverse
Cats
Cats

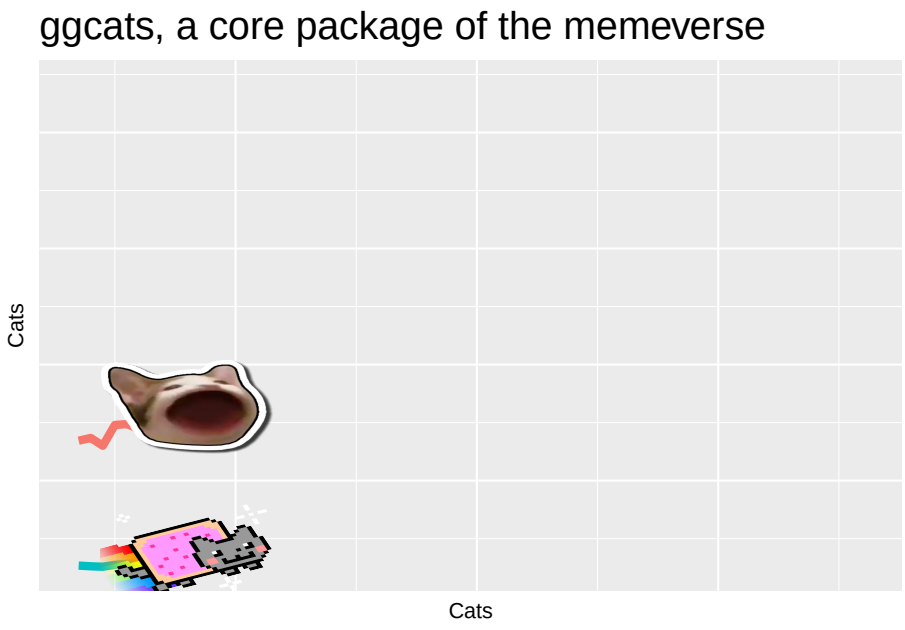

ggcats, a core package of the memeverse
Cats
Cats

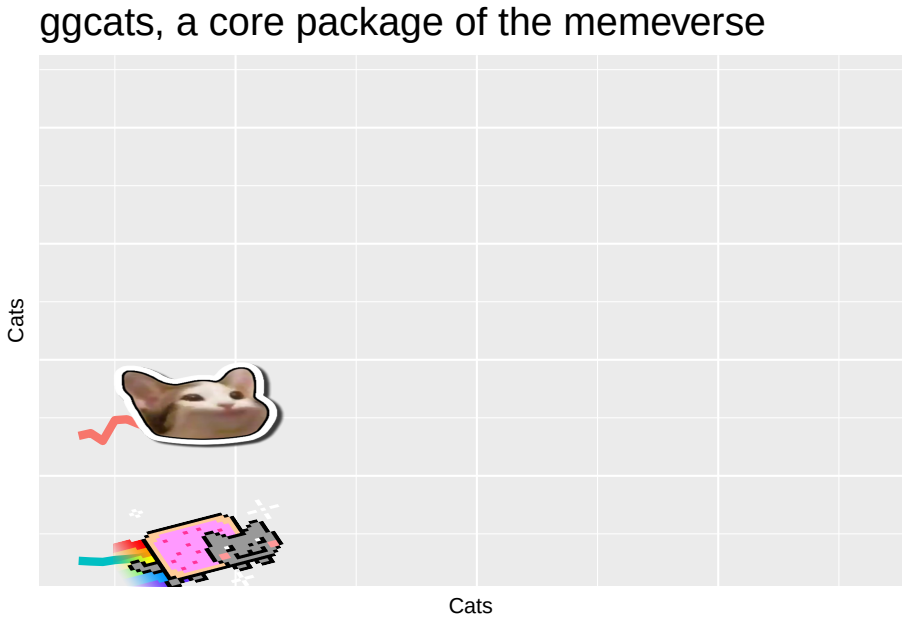

ggcats, a core package of the memeverse
Cats
Cats

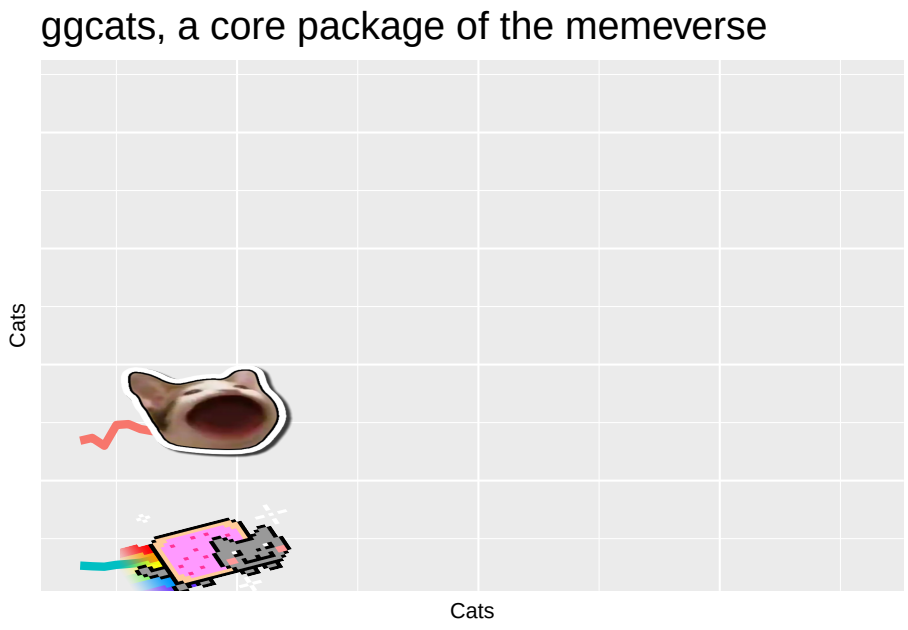

ggcats, a core package of the memeverse
Cats
Cats

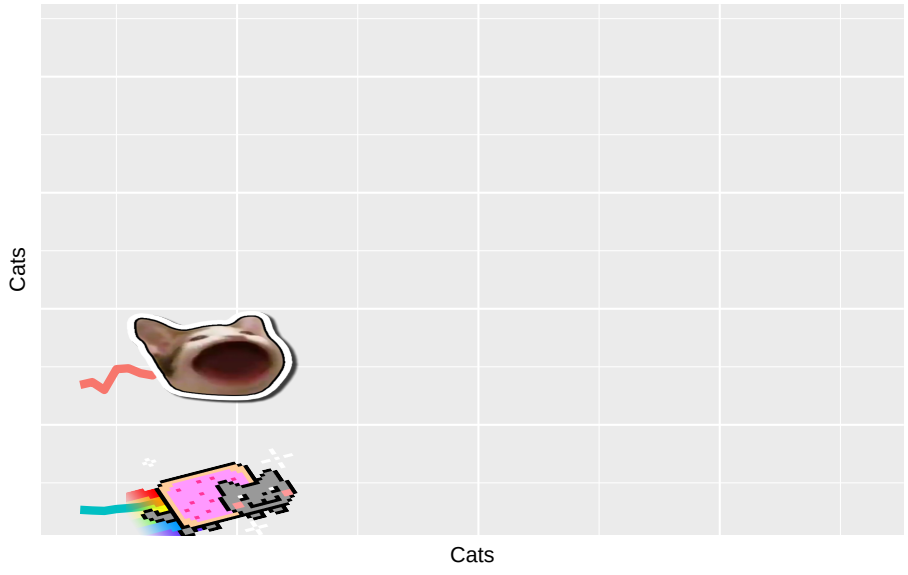

ggcats, a core package of the memeverse
Cats
Cats

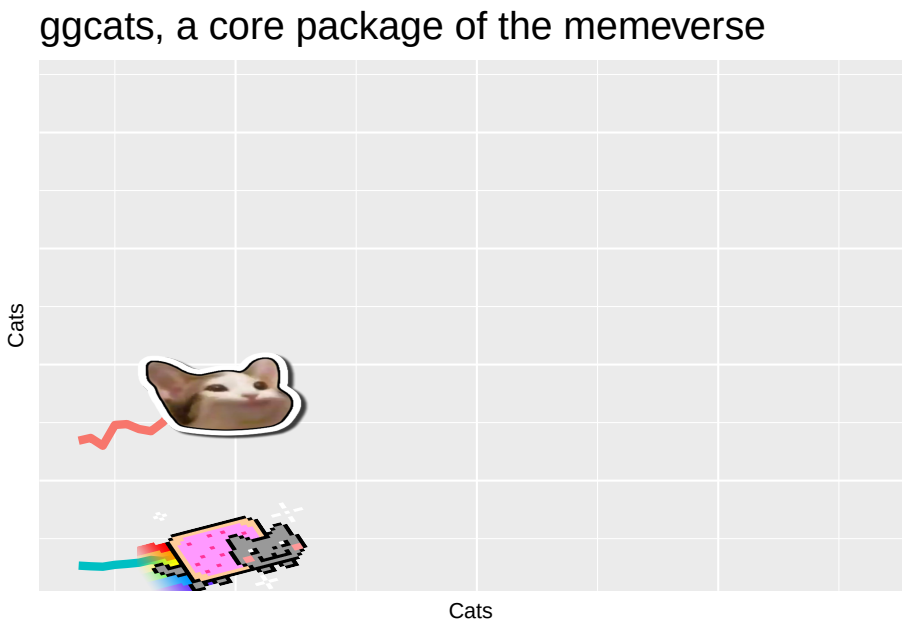

ggcats, a core package of the memeverse
Cats
Cats

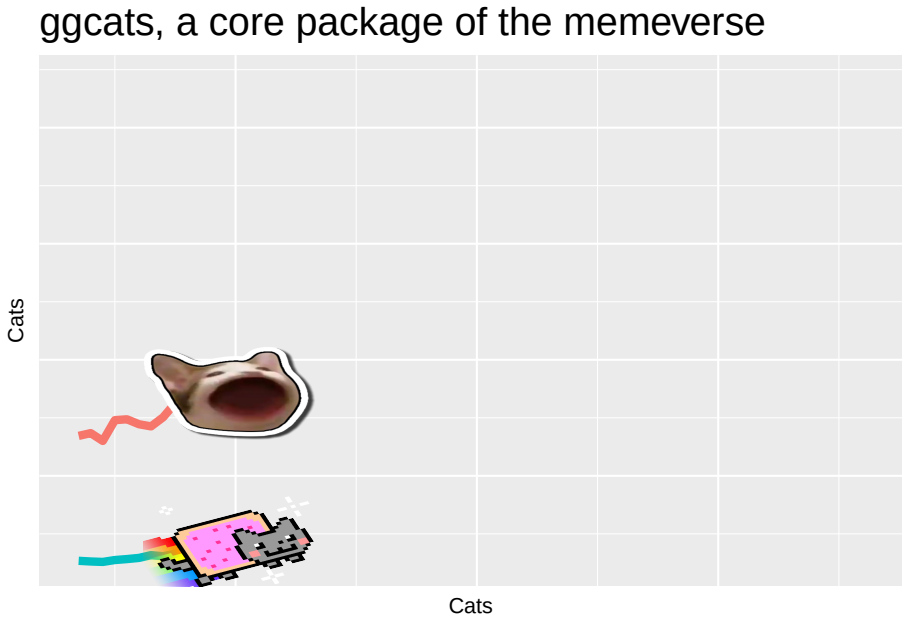

ggcats, a core package of the memeverse
Cats
Cats

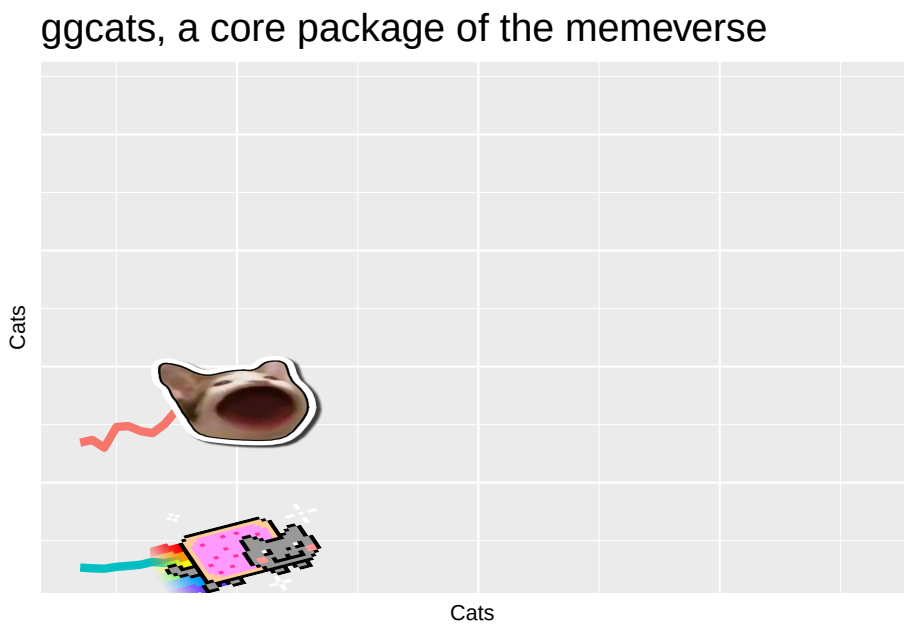

ggcats, a core package of the memeverse
Cats
Cats

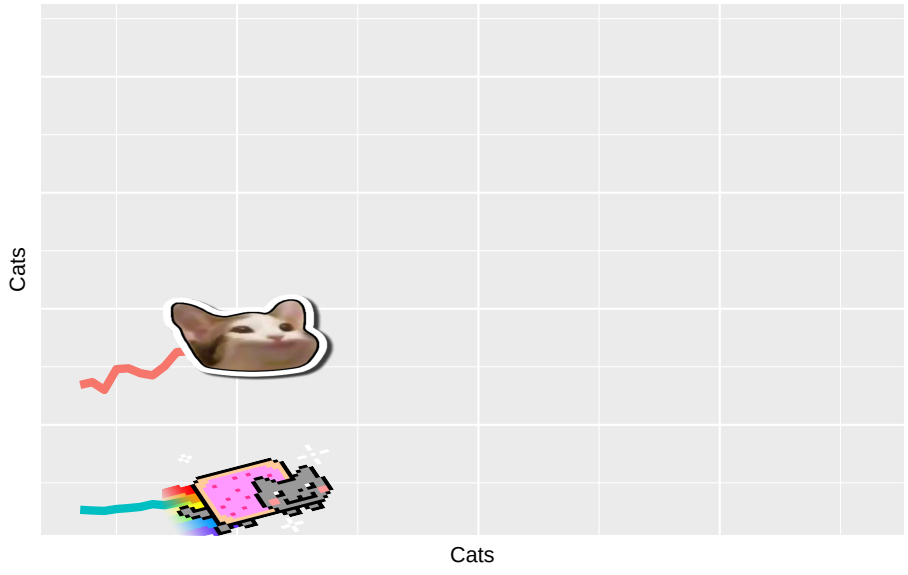

ggcats, a core package of the memeverse
Cats
Cats

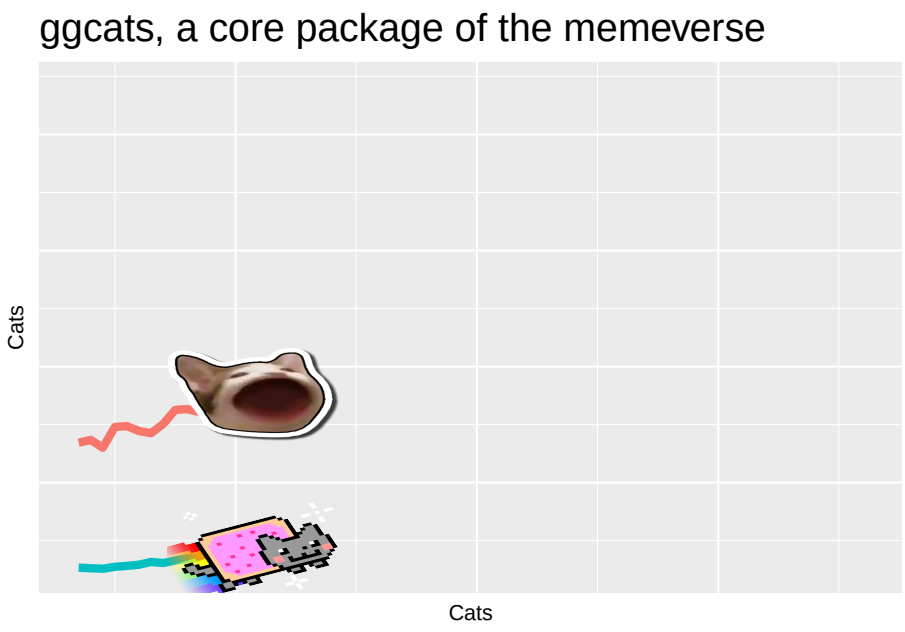

ggcats, a core package of the memeverse
Cats
Cats

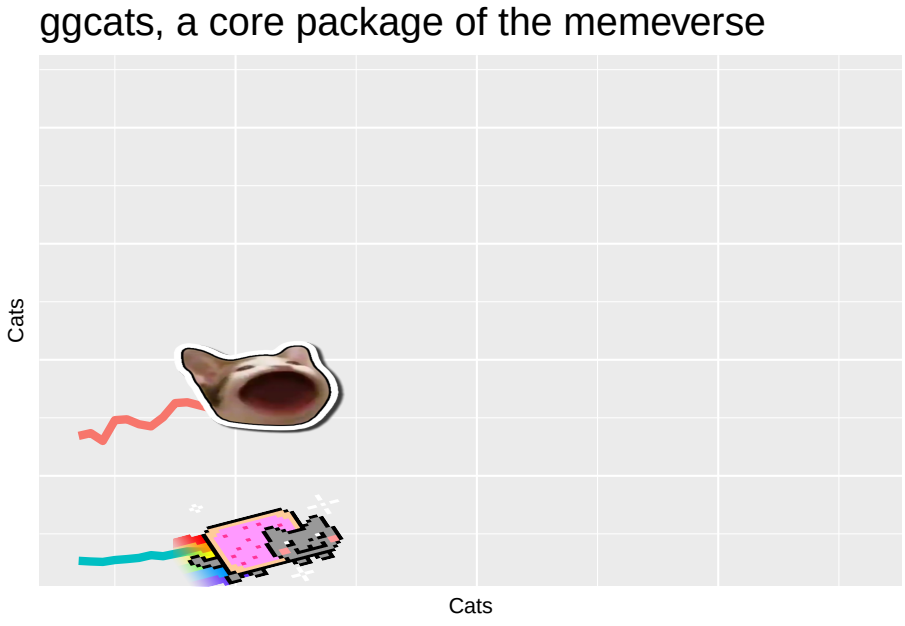

ggcats, a core package of the memeverse
Cats
Cats

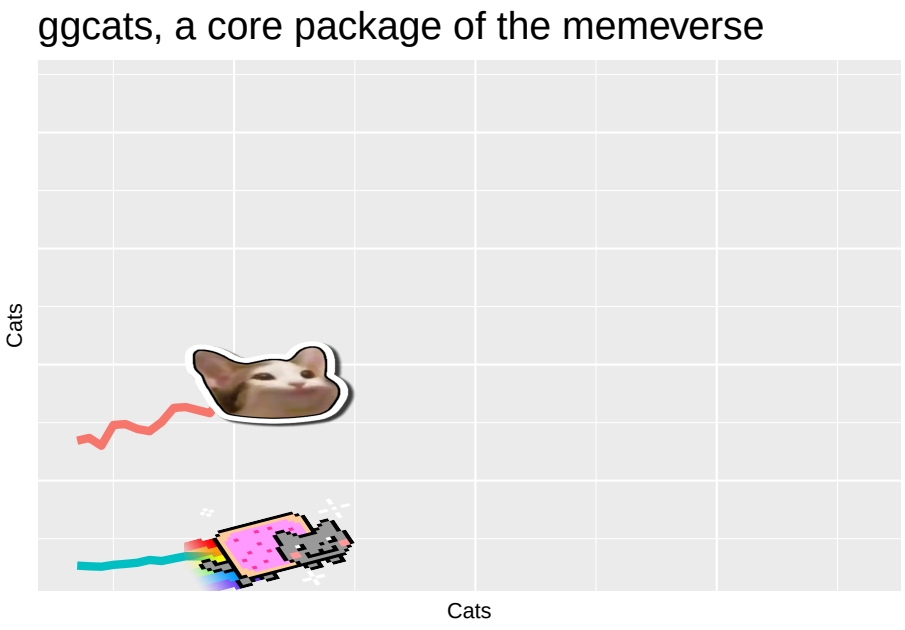

ggcats, a core package of the memeverse
Cats
Cats

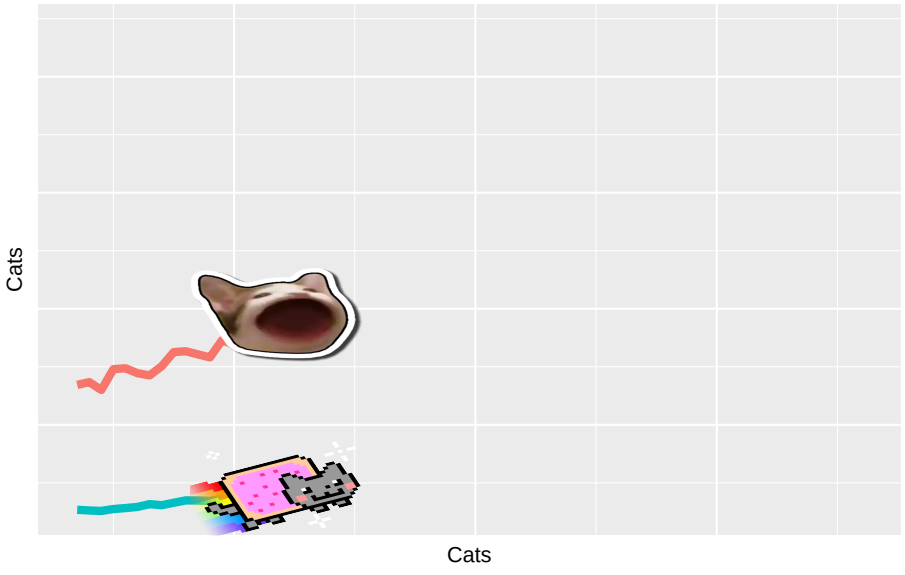

ggcats, a core package of the memeverse
Cats
Cats

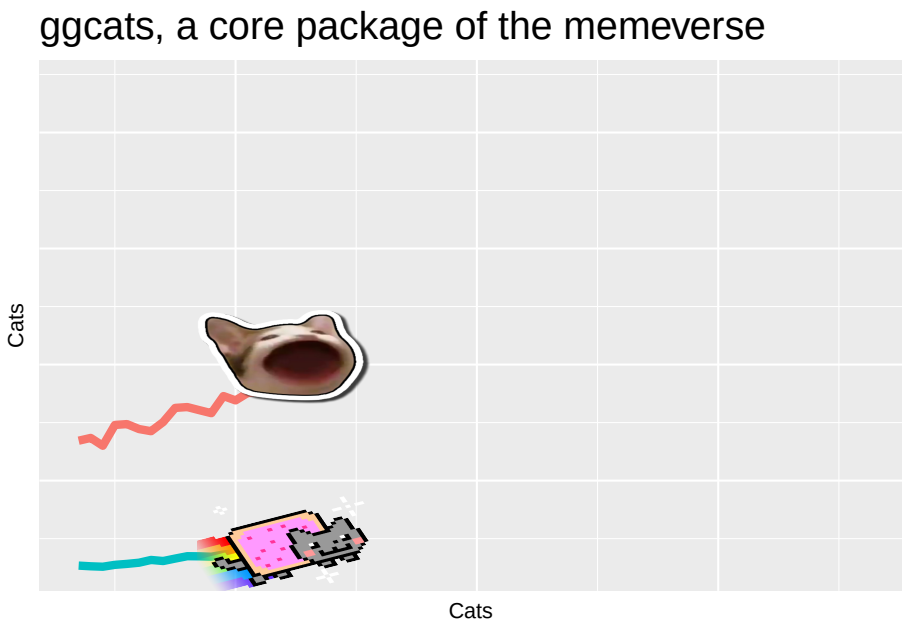

ggcats, a core package of the memeverse
Cats
Cats

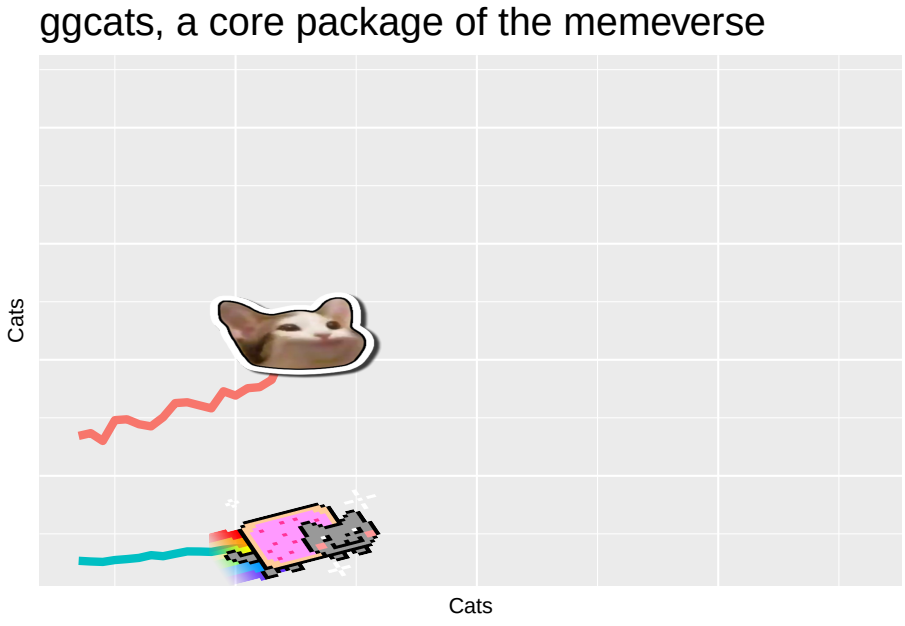

ggcats, a core package of the memeverse
Cats
Cats

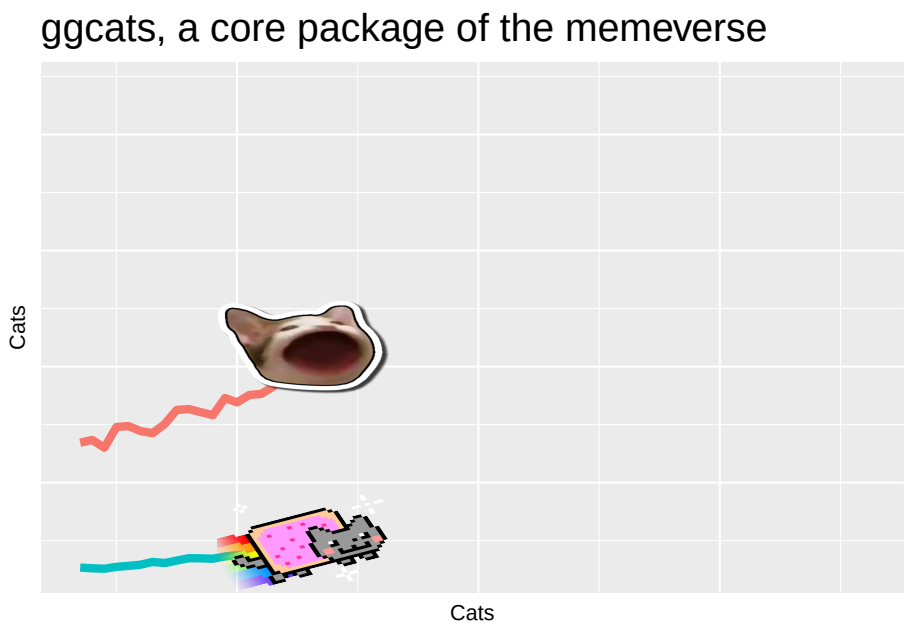

ggcats, a core package of the memeverse
Cats
Cats

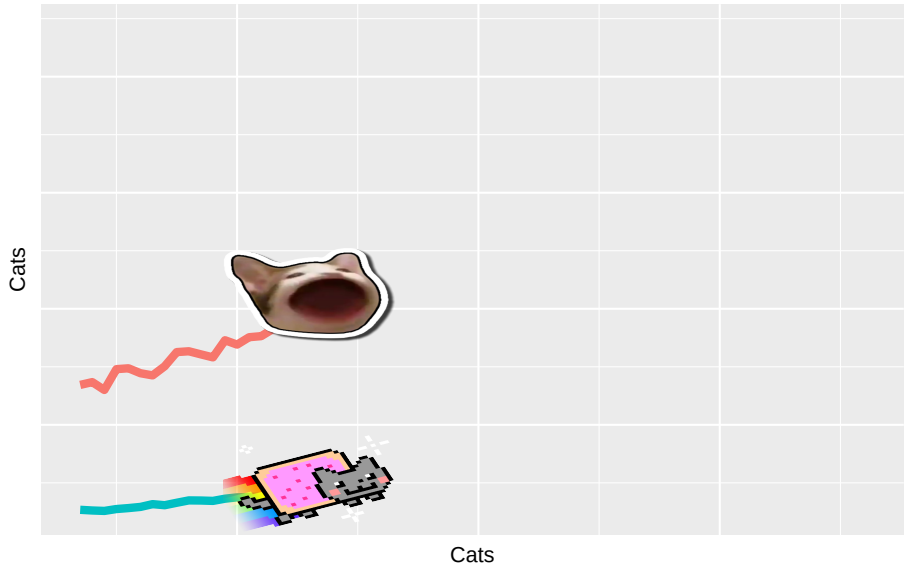

ggcats, a core package of the memeverse
Cats
Cats

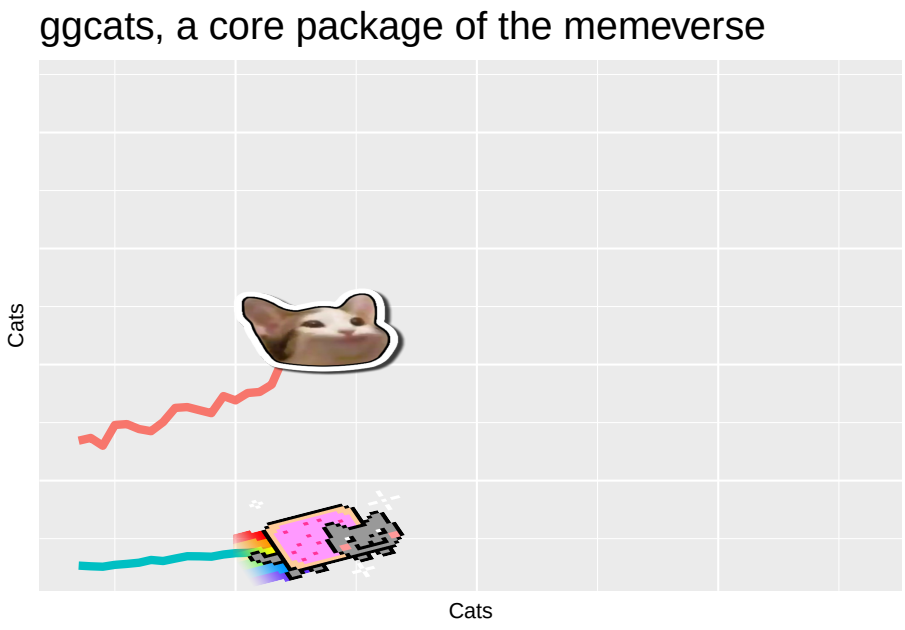

ggcats, a core package of the memeverse
Cats
Cats

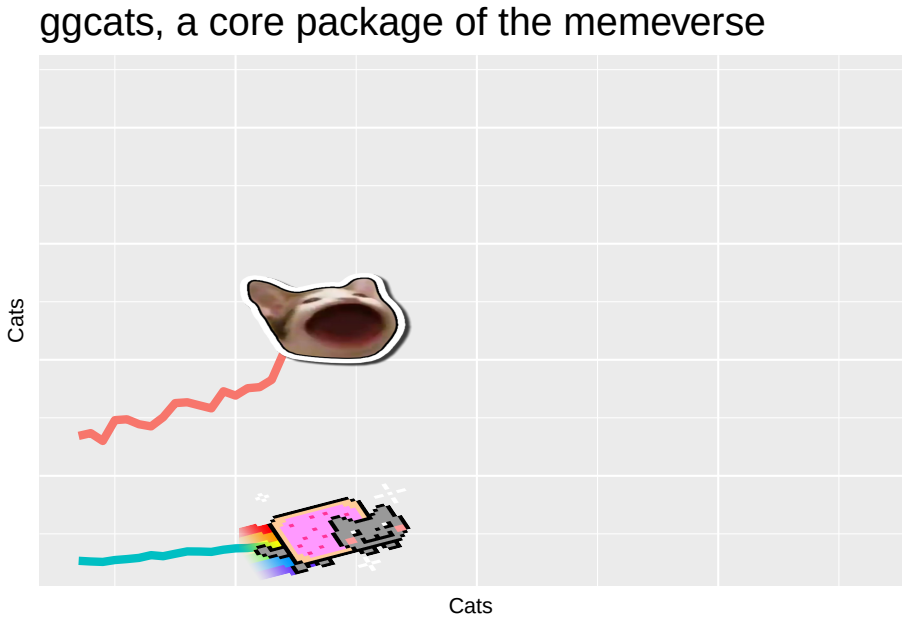

ggcats, a core package of the memeverse
Cats
Cats

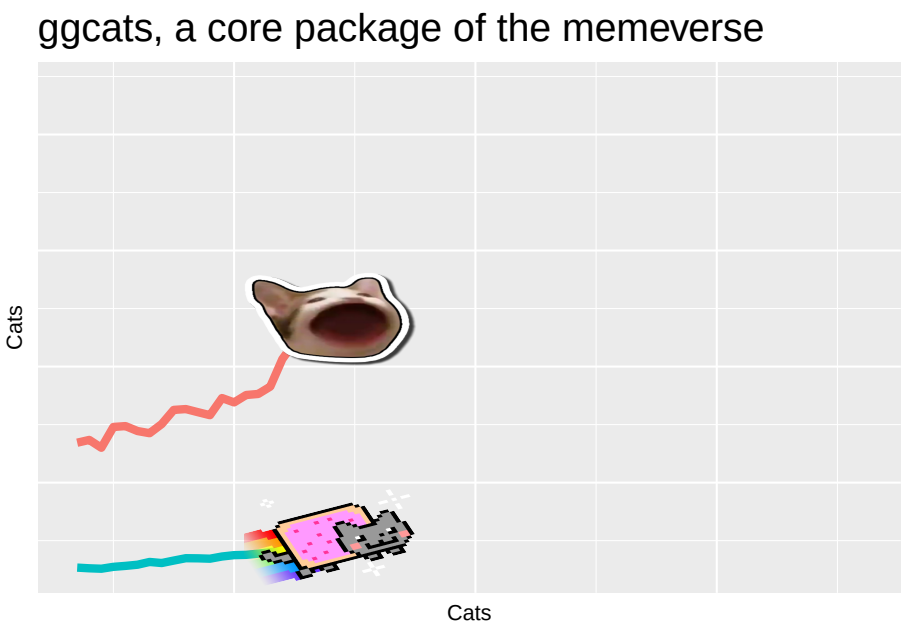

ggcats, a core package of the memeverse
Cats
Cats

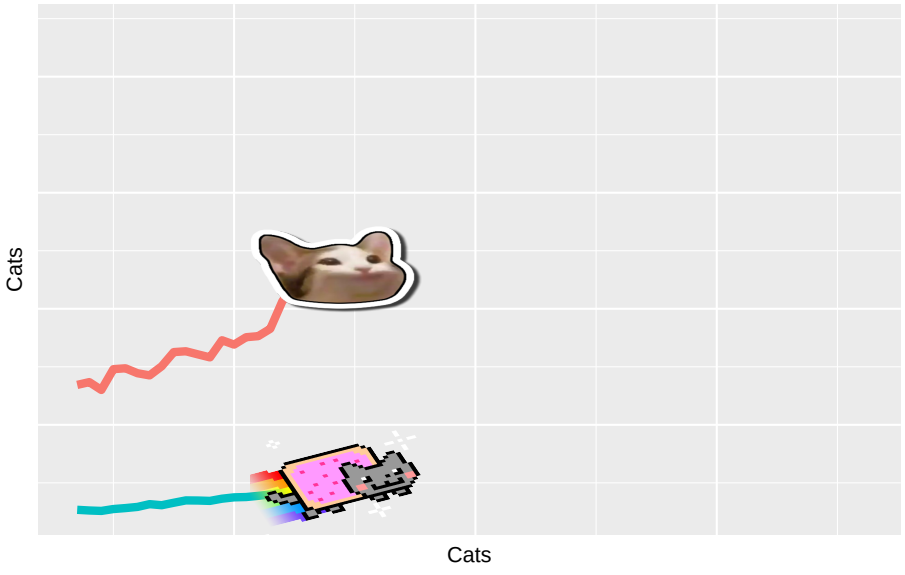

ggcats, a core package of the memeverse
Cats
Cats

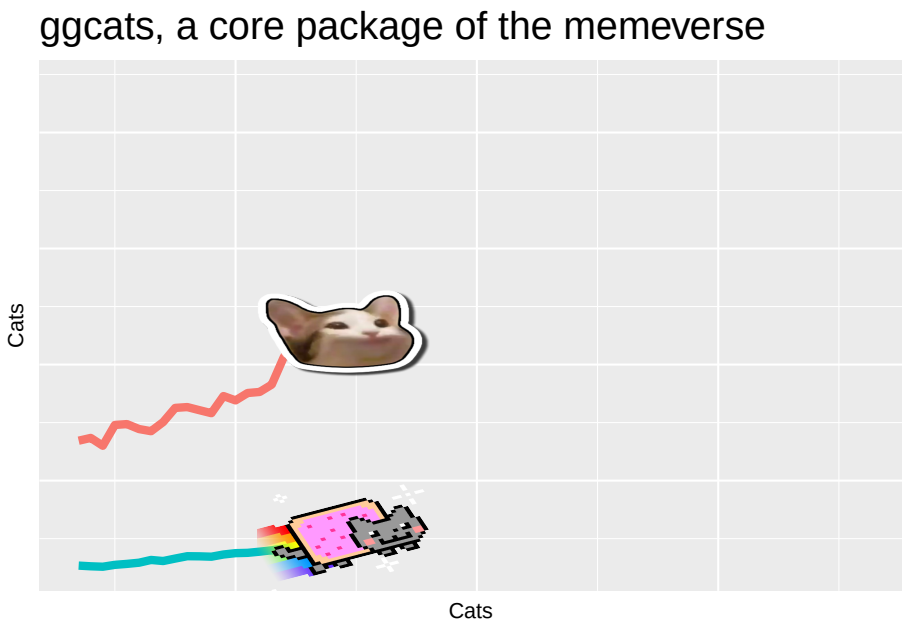

ggcats, a core package of the memeverse
Cats
Cats

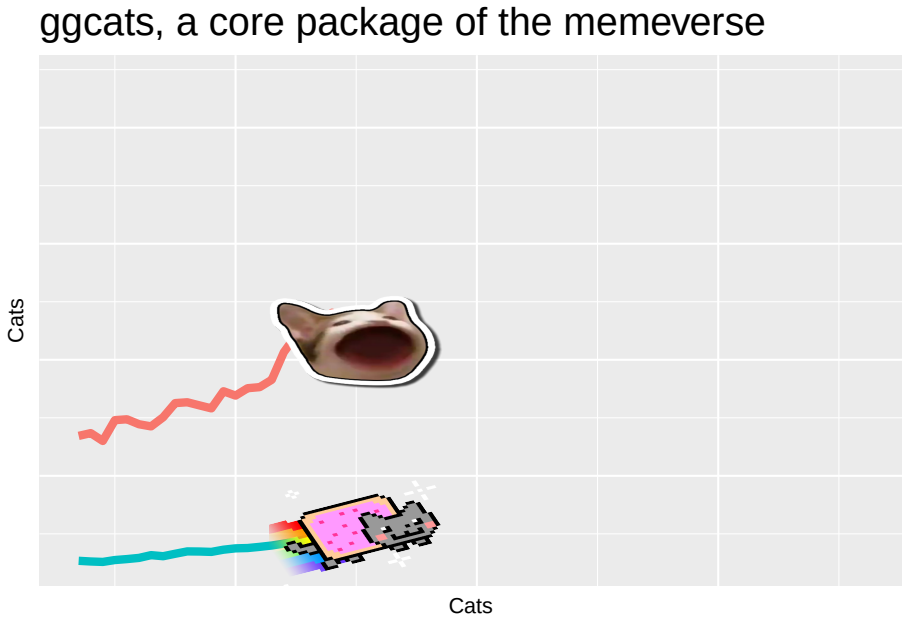

ggcats, a core package of the memeverse
Cats
Cats

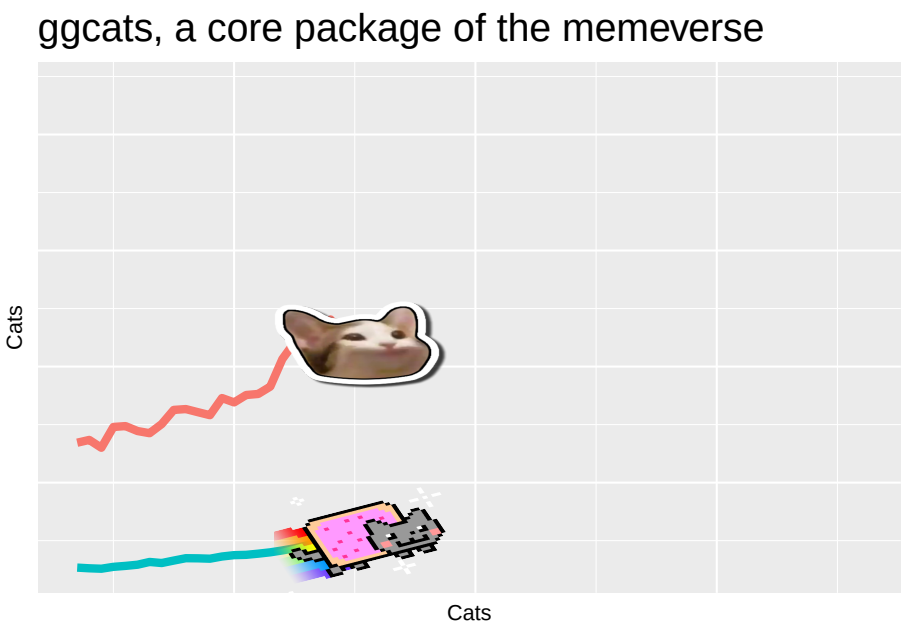

ggcats, a core package of the memeverse
Cats
Cats

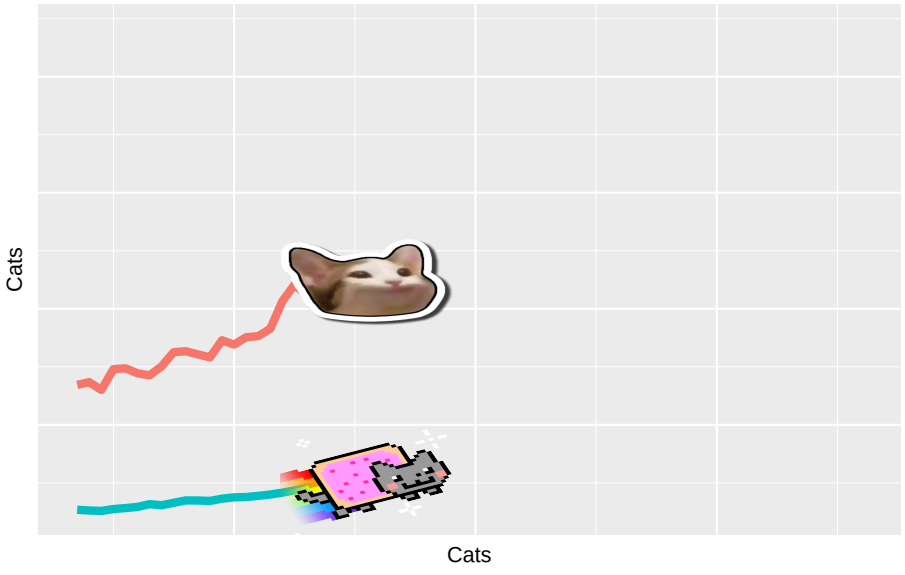

ggcats, a core package of the memeverse
Cats
Cats

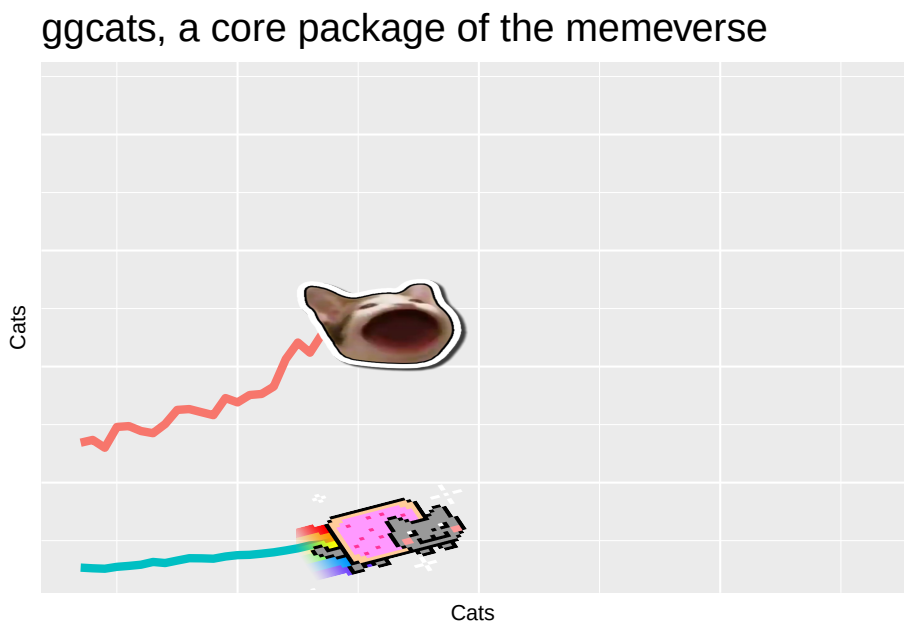

ggcats, a core package of the memeverse
Cats
Cats

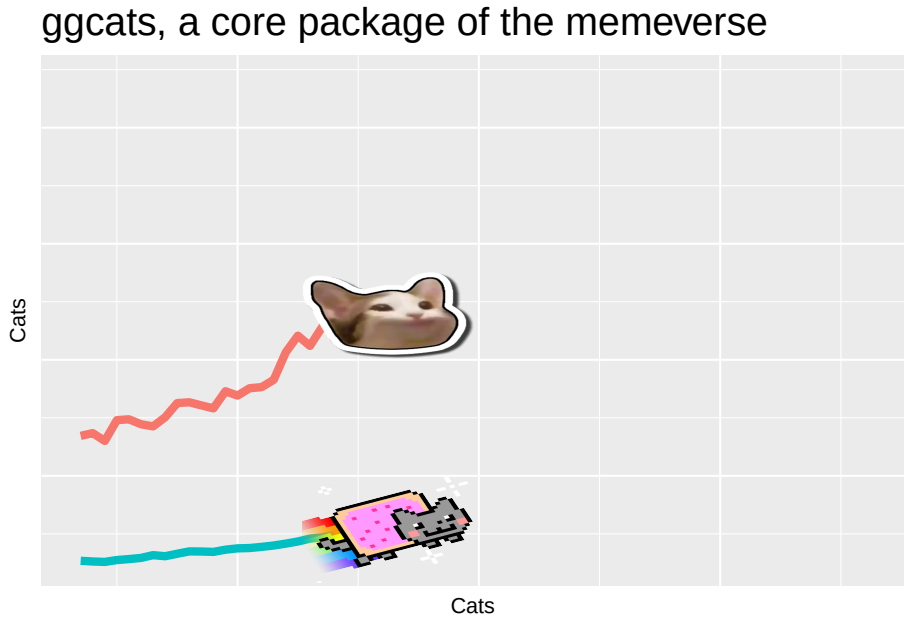

ggcats, a core package of the memeverse
Cats
Cats

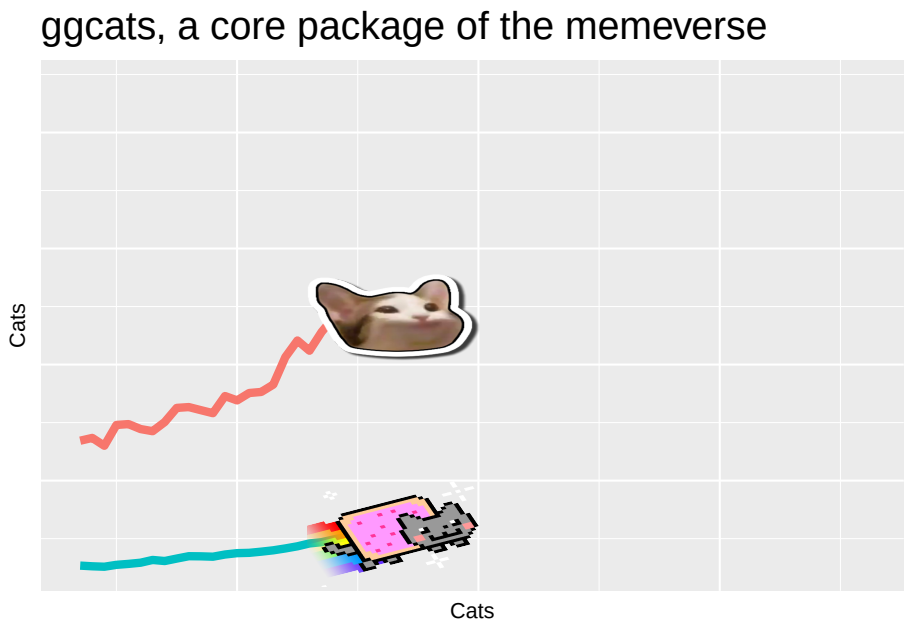

ggcats, a core package of the memeverse
Cats
Cats

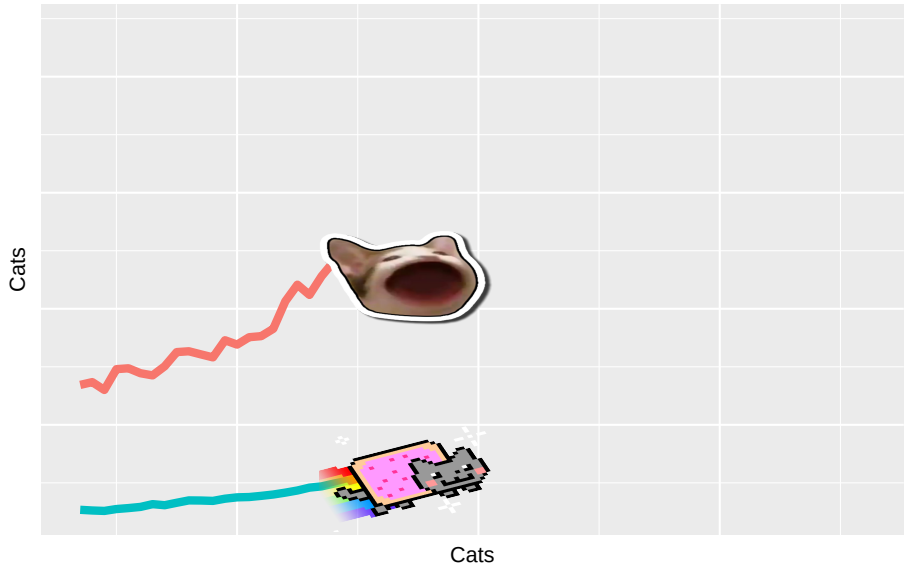

ggcats, a core package of the memeverse
Cats
Cats

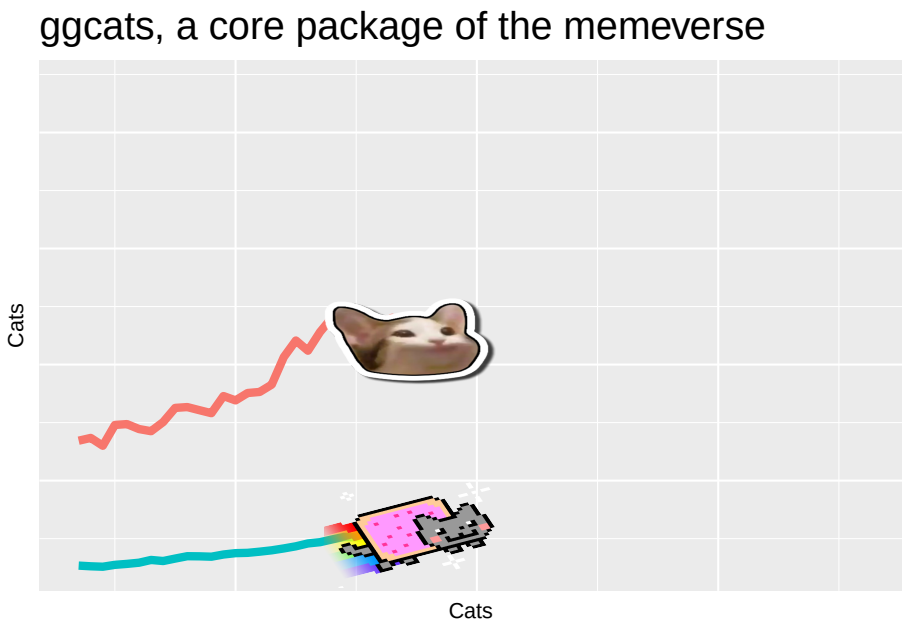

ggcats, a core package of the memeverse
Cats
Cats

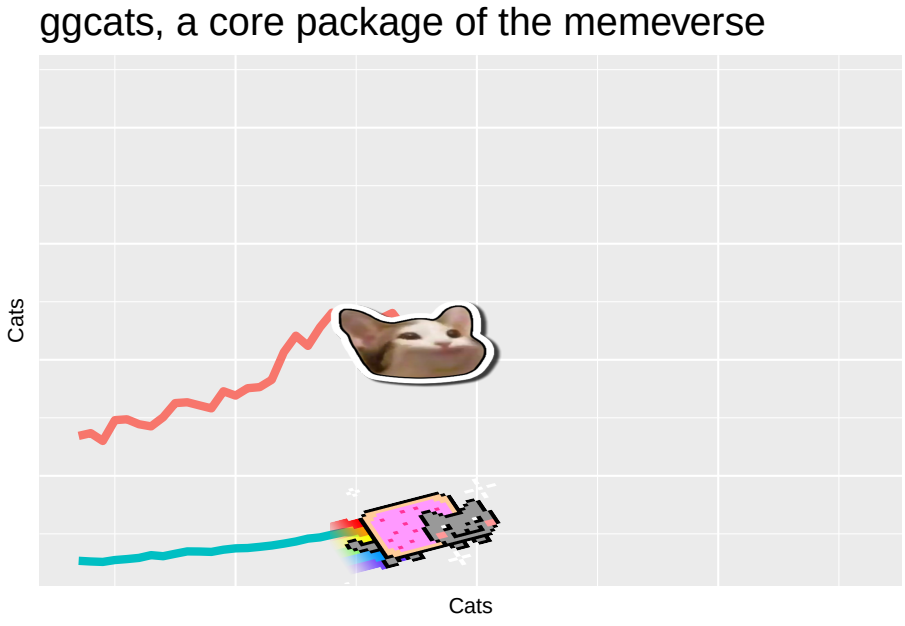

ggcats, a core package of the memeverse
Cats
Cats

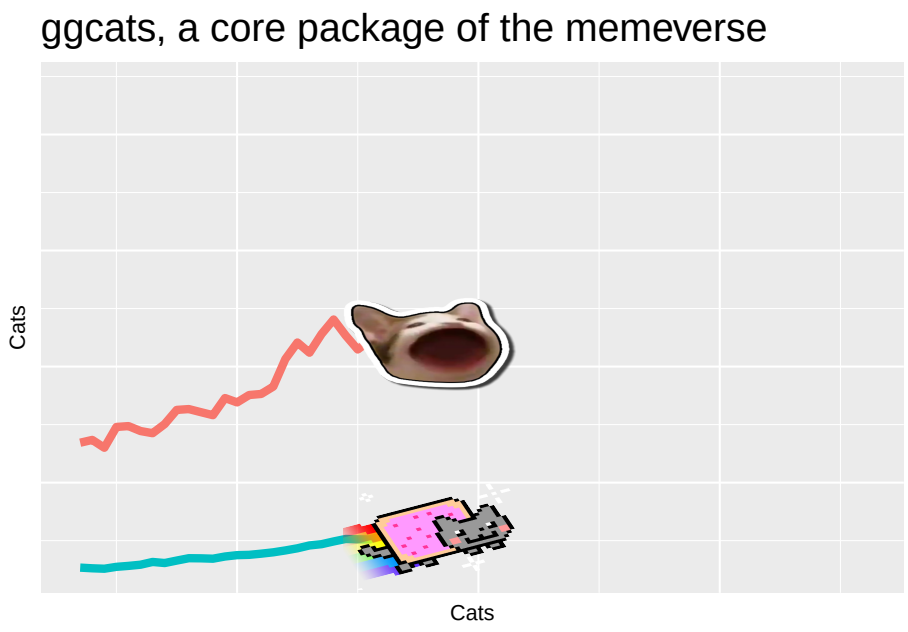

ggcats, a core package of the memeverse
Cats
Cats

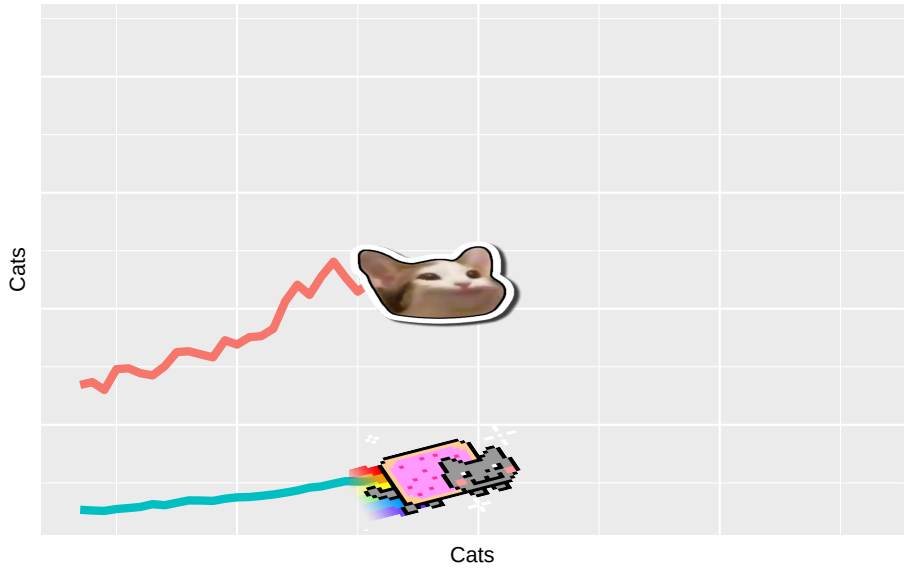

ggcats, a core package of the memeverse
Cats
Cats

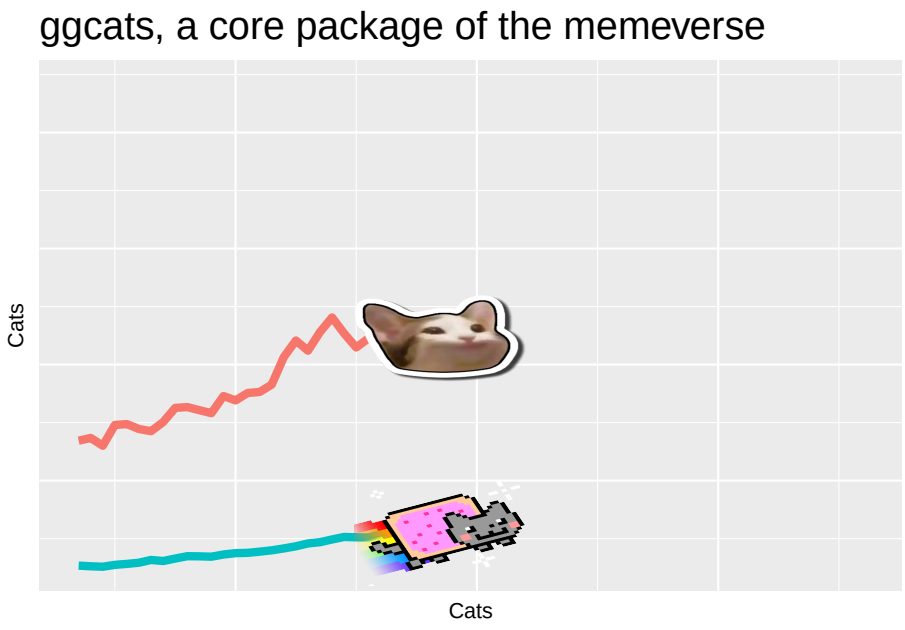

ggcats, a core package of the memeverse
Cats
Cats

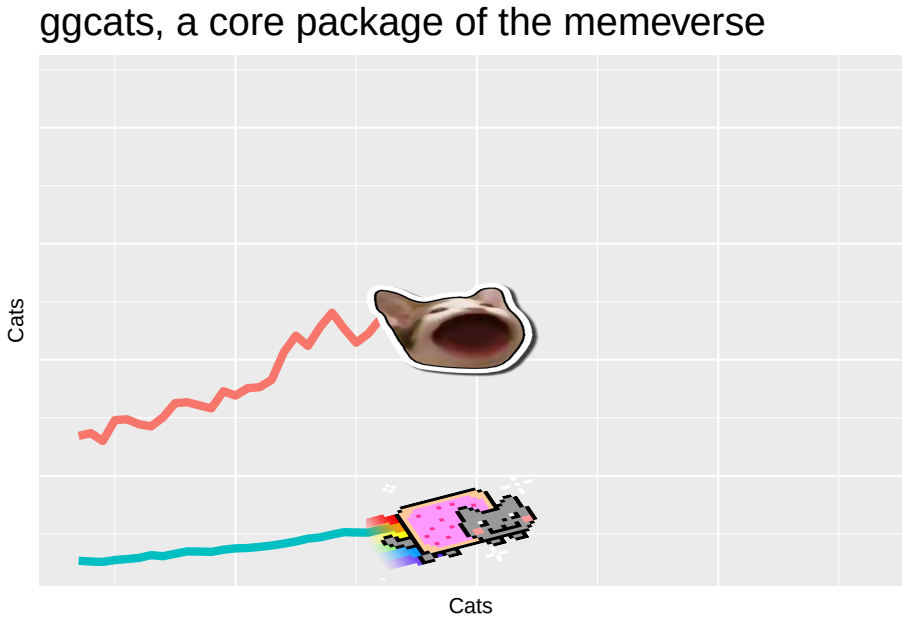

ggcats, a core package of the memeverse
Cats
Cats

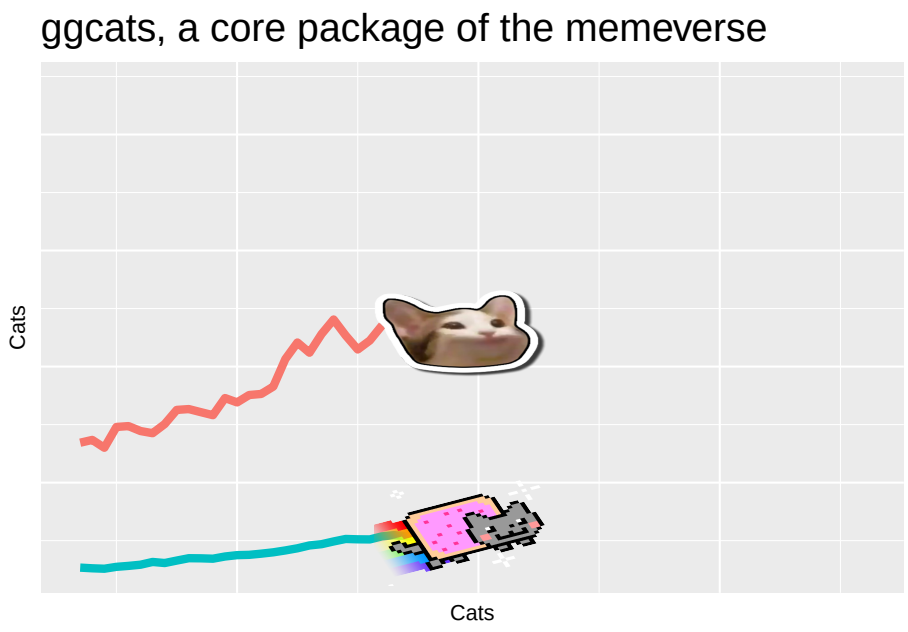

ggcats, a core package of the memeverse
Cats
Cats

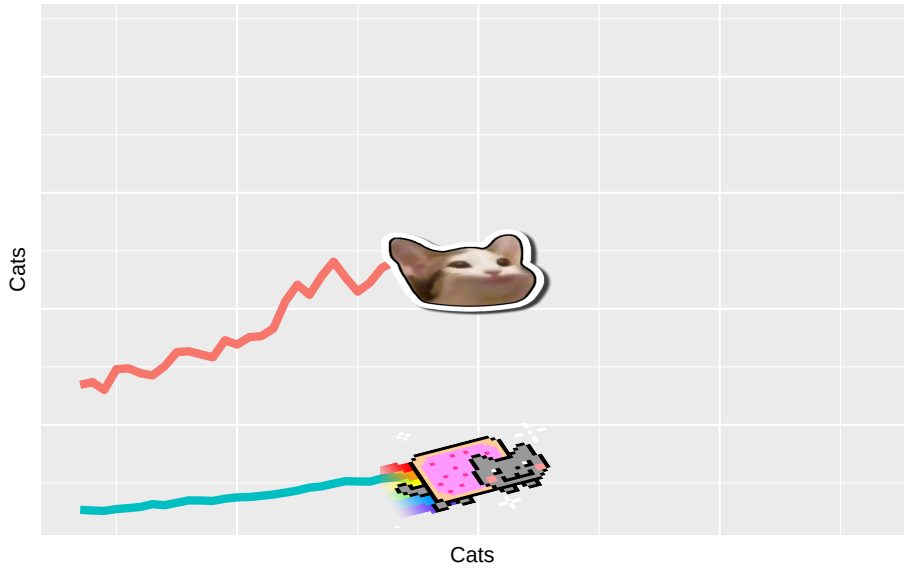

ggcats, a core package of the memeverse
Cats
Cats

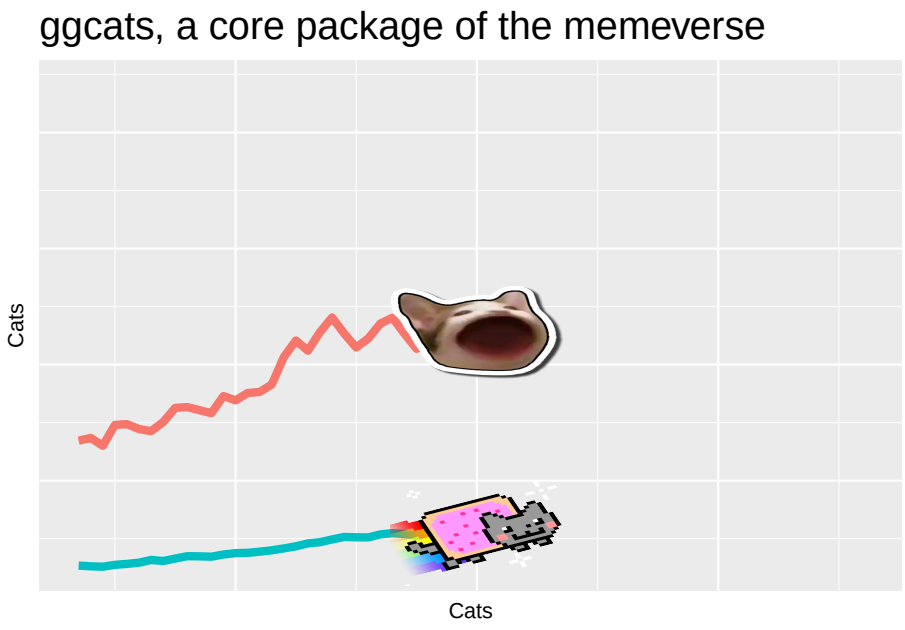

ggcats, a core package of the memeverse
Cats
Cats

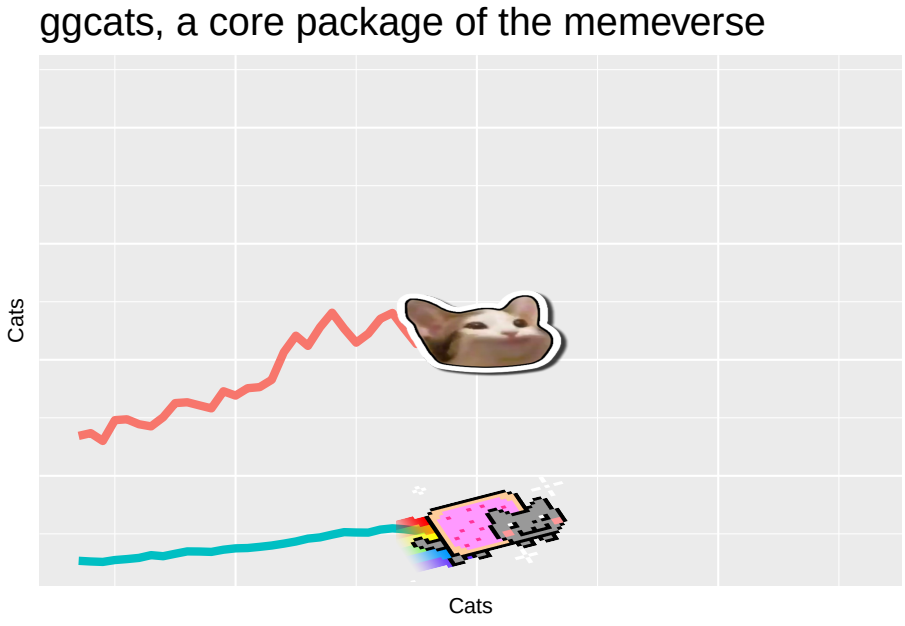

ggcats, a core package of the memeverse
Cats
Cats

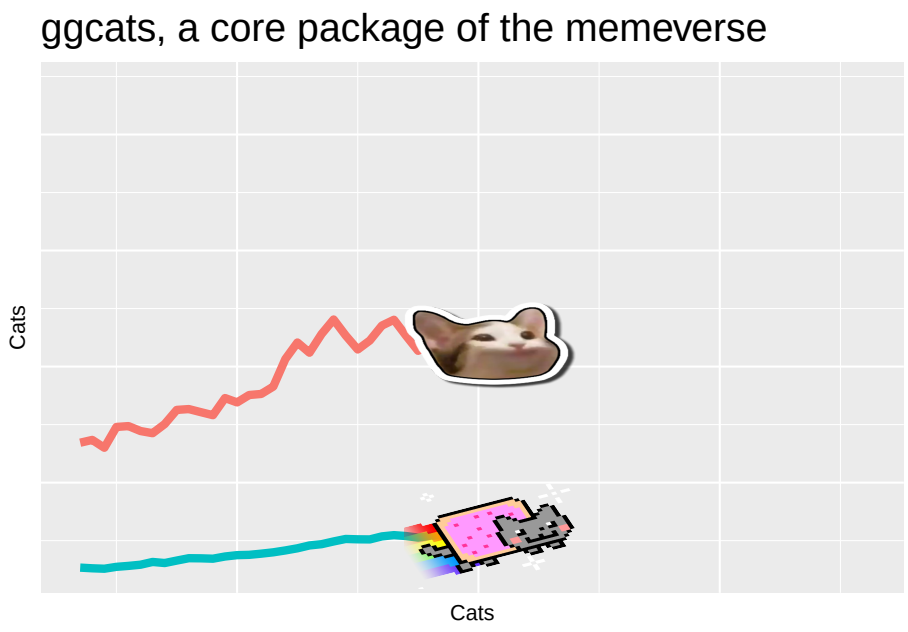

ggcats, a core package of the memeverse
Cats
Cats

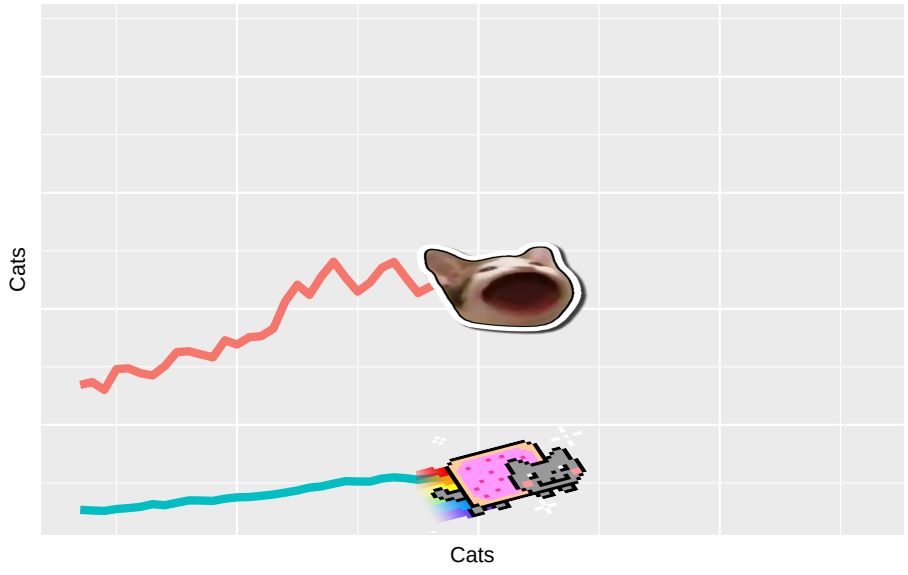

ggcats, a core package of the memeverse
Cats
Cats

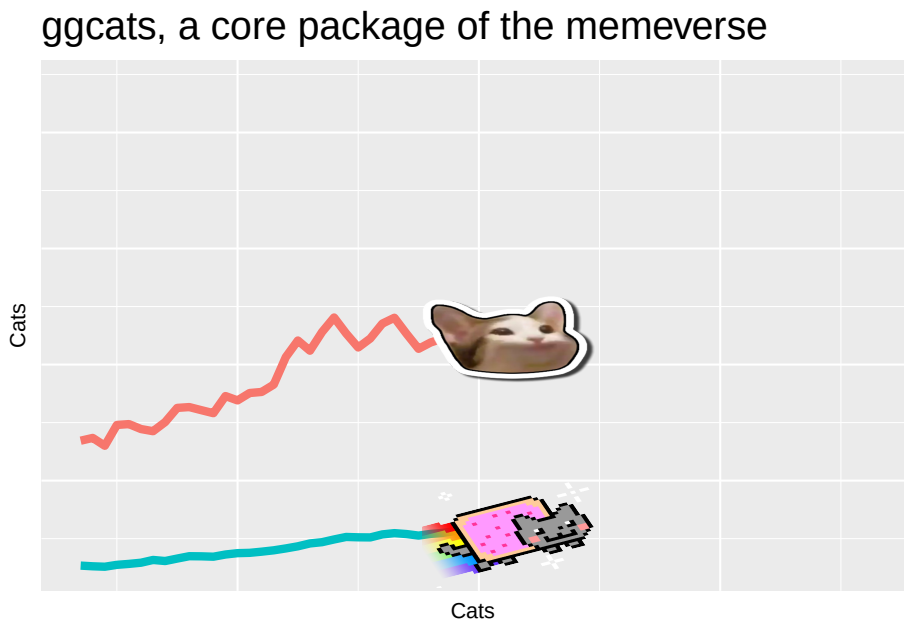

ggcats, a core package of the memeverse
Cats
Cats

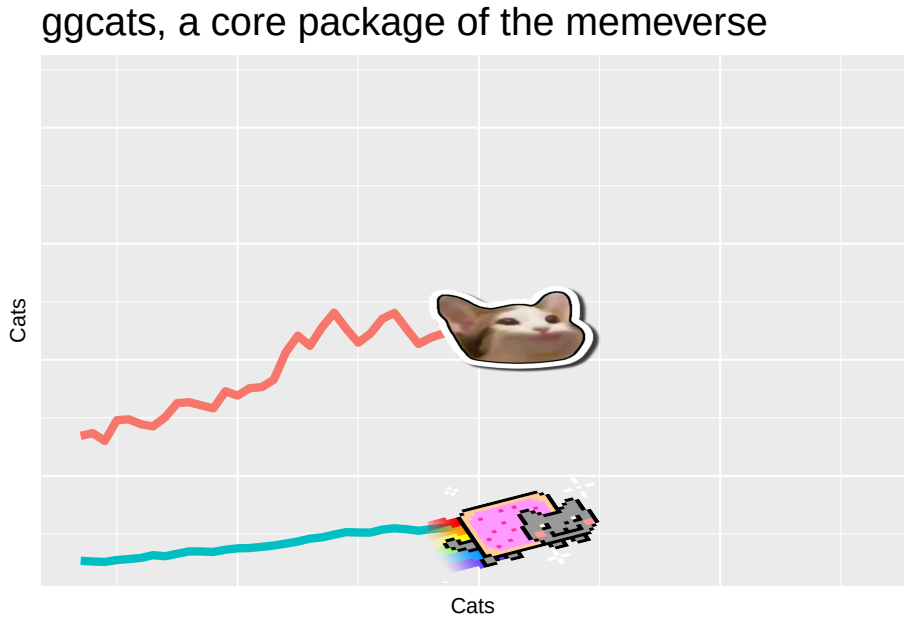

ggcats, a core package of the memeverse
Cats
Cats

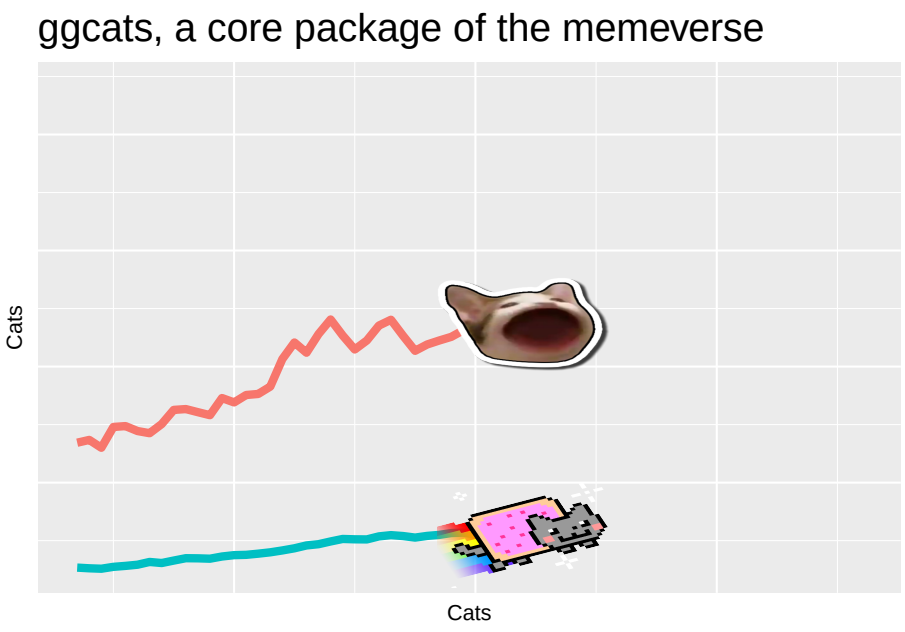

ggcats, a core package of the memeverse
Cats
Cats

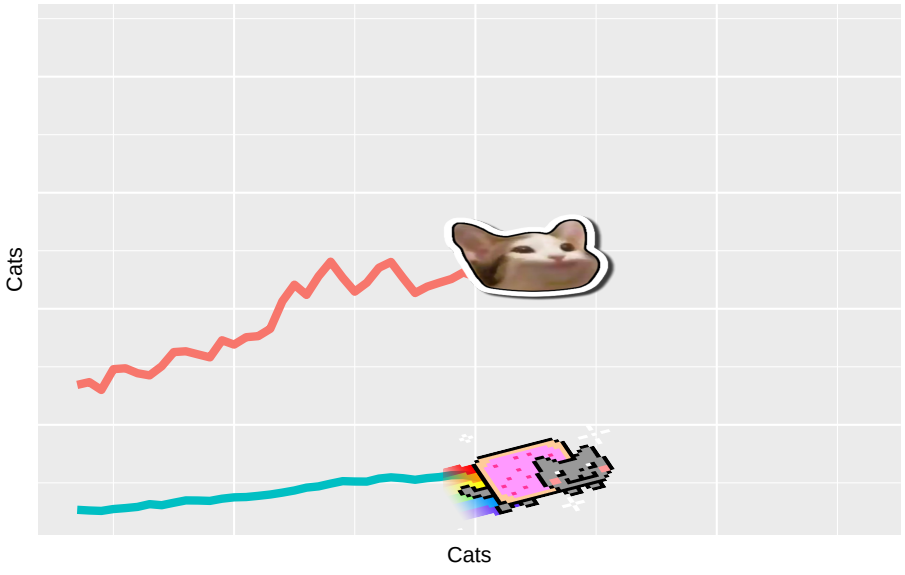

ggcats, a core package of the memeverse
Cats
Cats

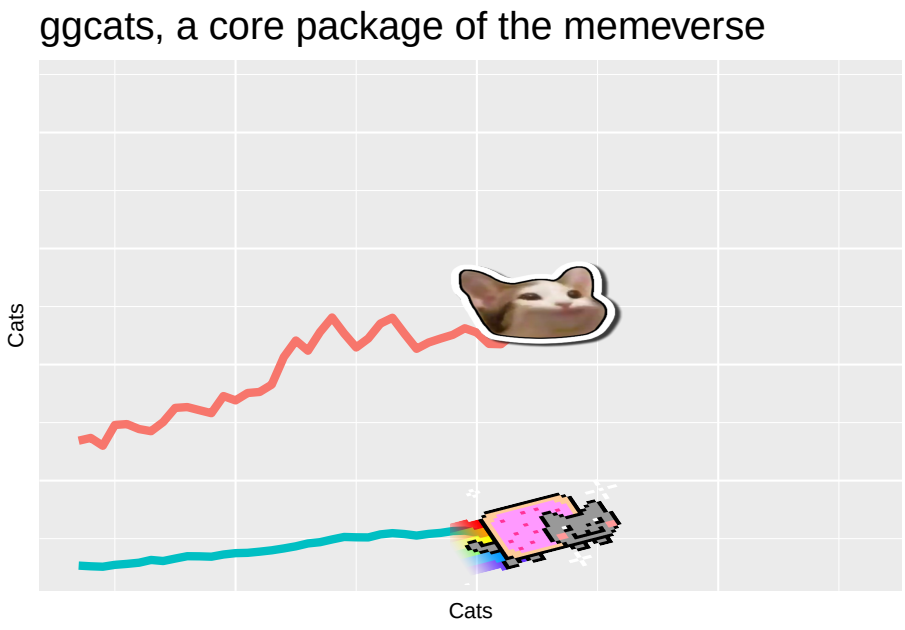

ggcats, a core package of the memeverse
Cats
Cats

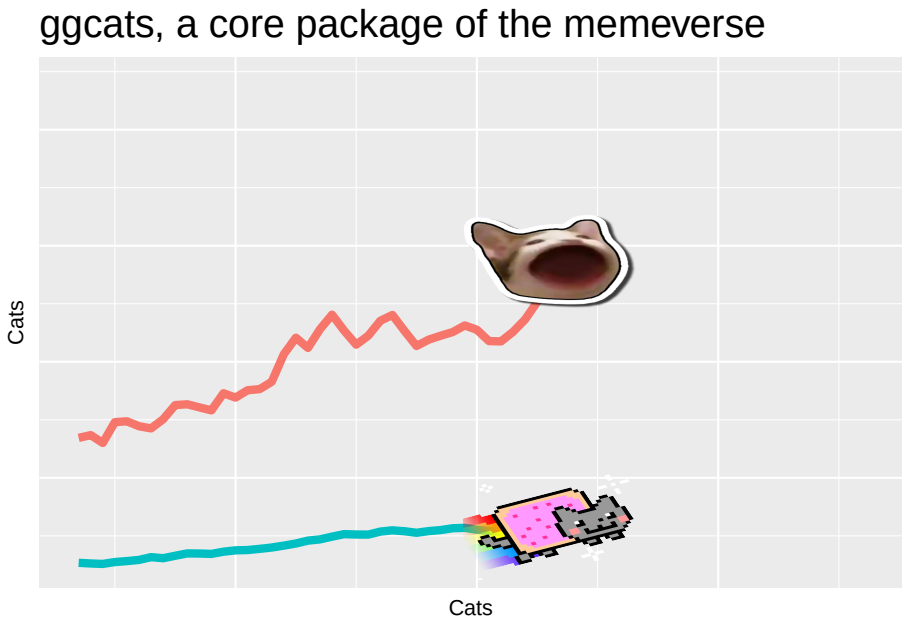

ggcats, a core package of the memeverse
Cats
Cats

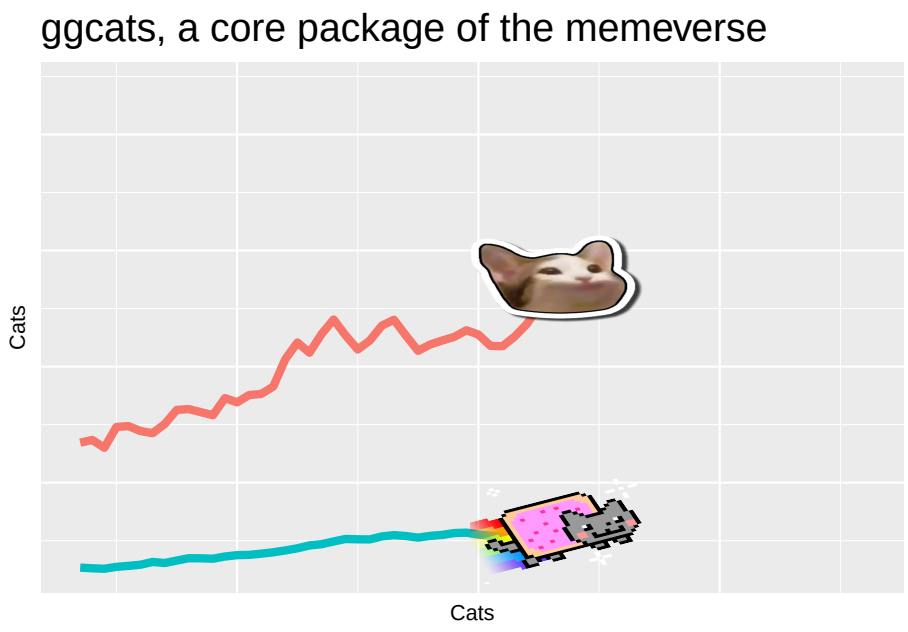

ggcats, a core package of the memeverse
Cats
Cats

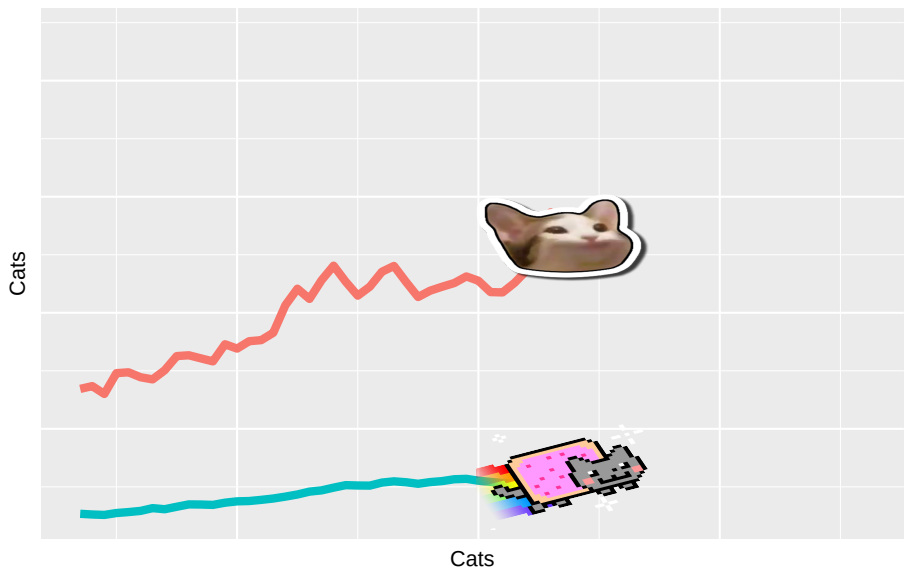

ggcats, a core package of the memeverse
Cats
Cats

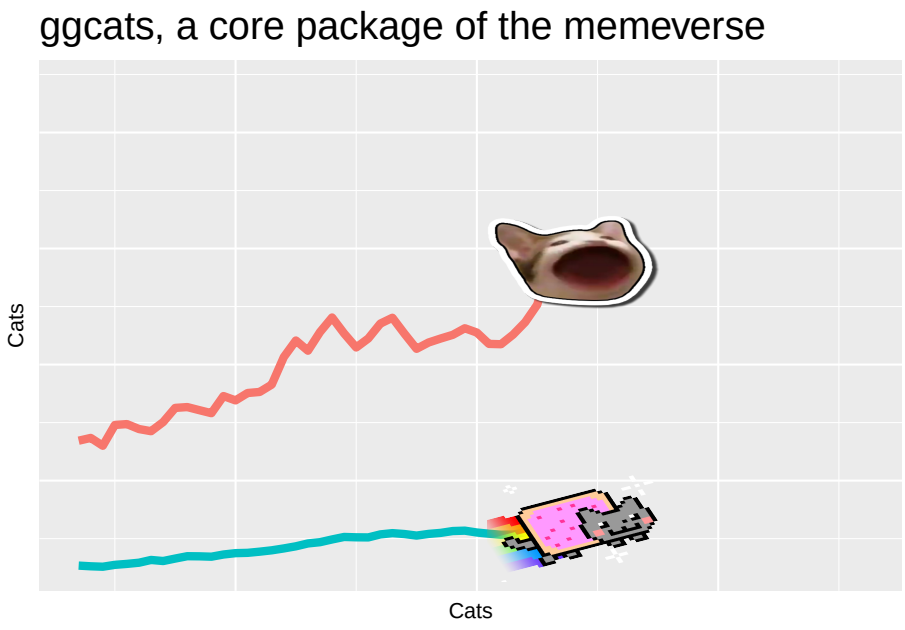

ggcats, a core package of the memeverse
Cats
Cats

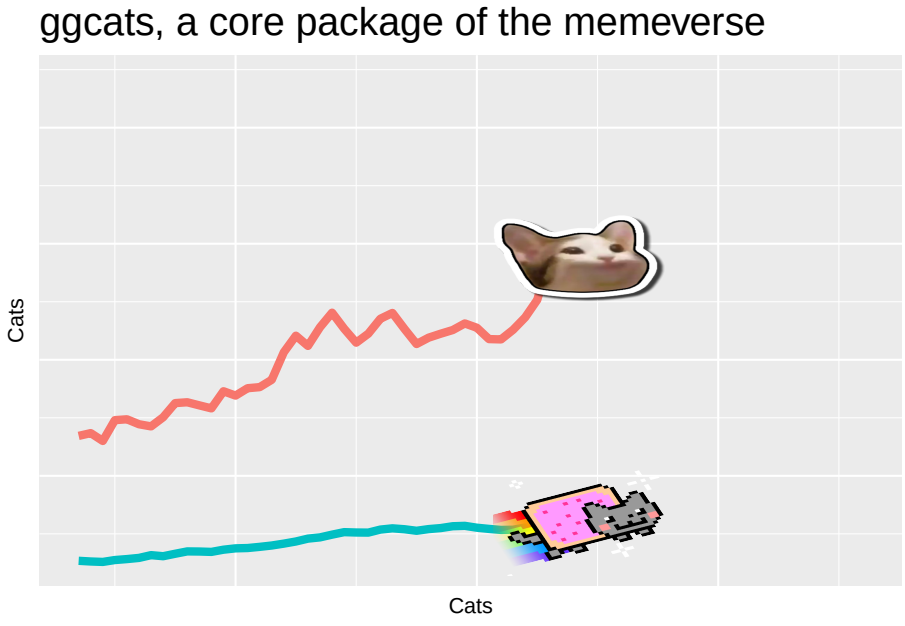

ggcats, a core package of the memeverse
Cats
Cats

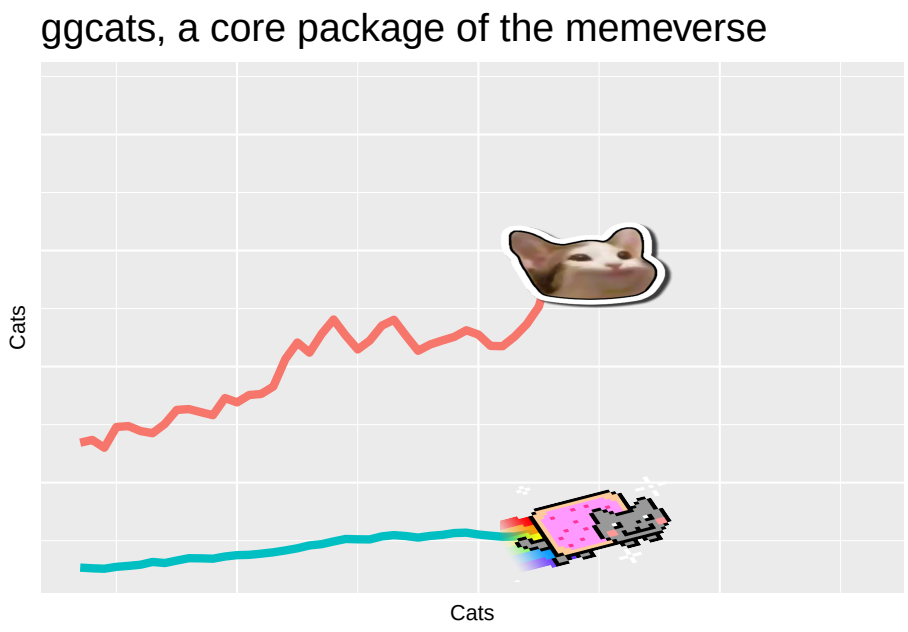

ggcats, a core package of the memeverse
Cats
Cats

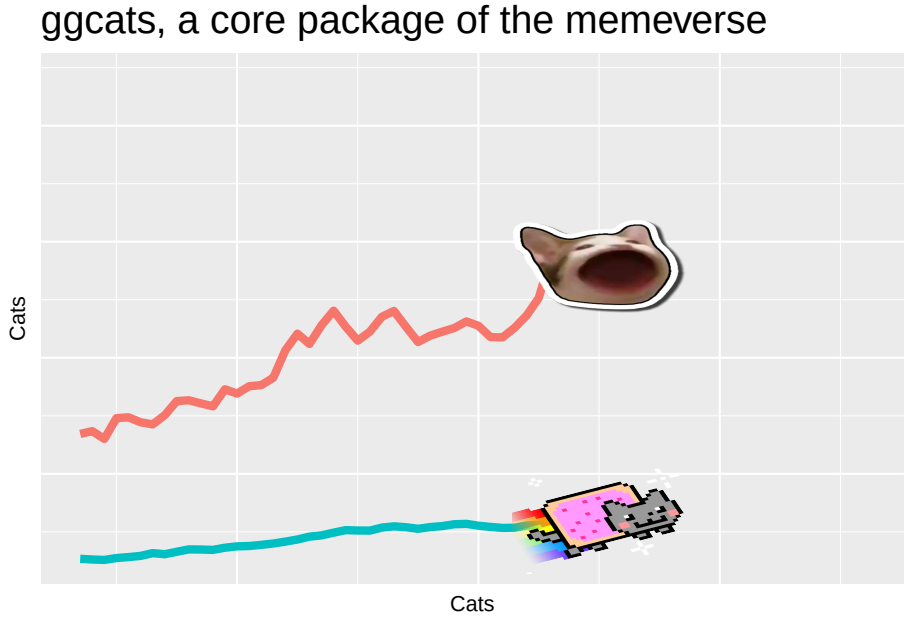

ggcats, a core package of the memeverse
Cats
Cats

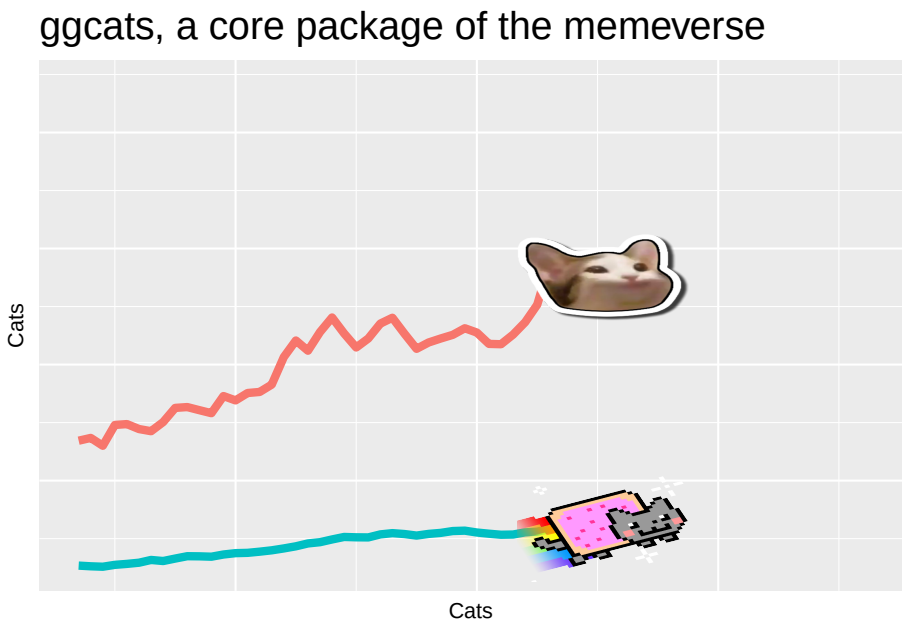

ggcats, a core package of the memeverse
Cats
Cats

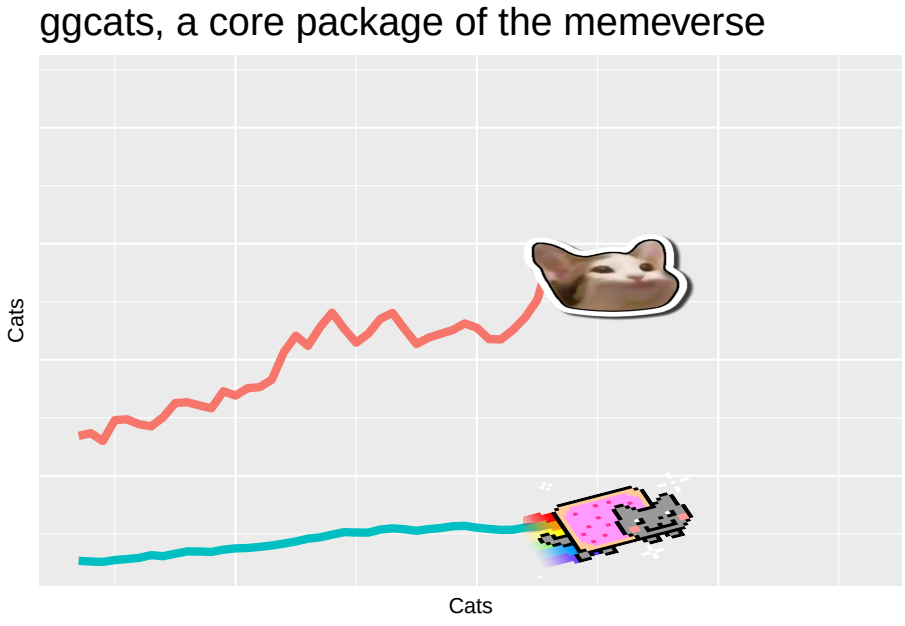

ggcats, a core package of the memeverse
Cats
Cats

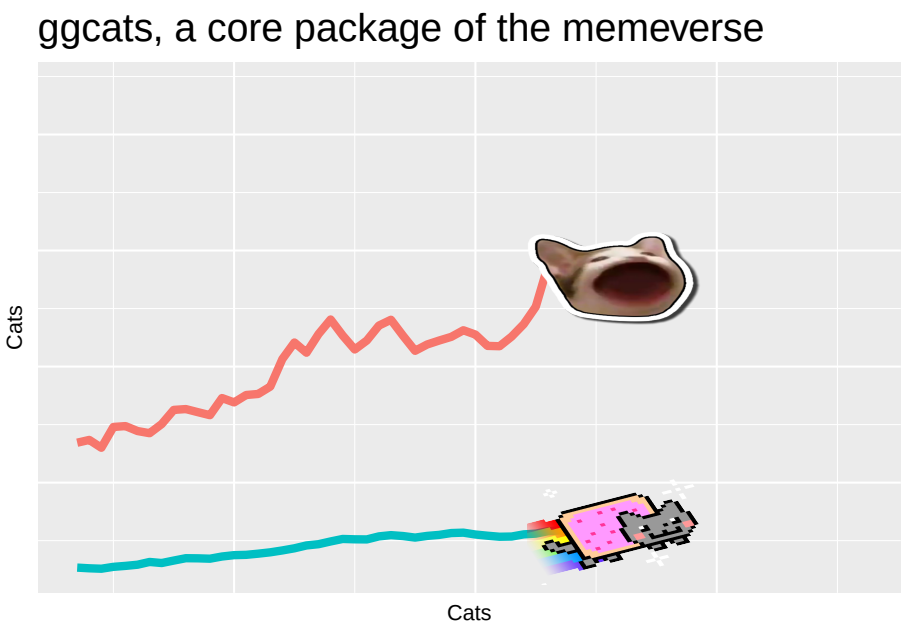

ggcats, a core package of the memeverse
Cats
Cats

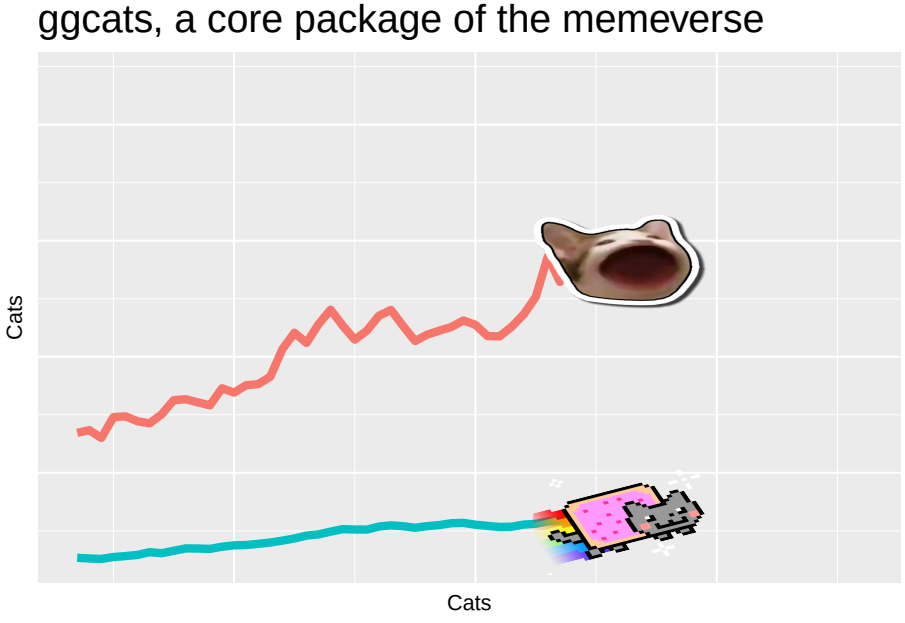

ggcats, a core package of the memeverse
Cats
Cats

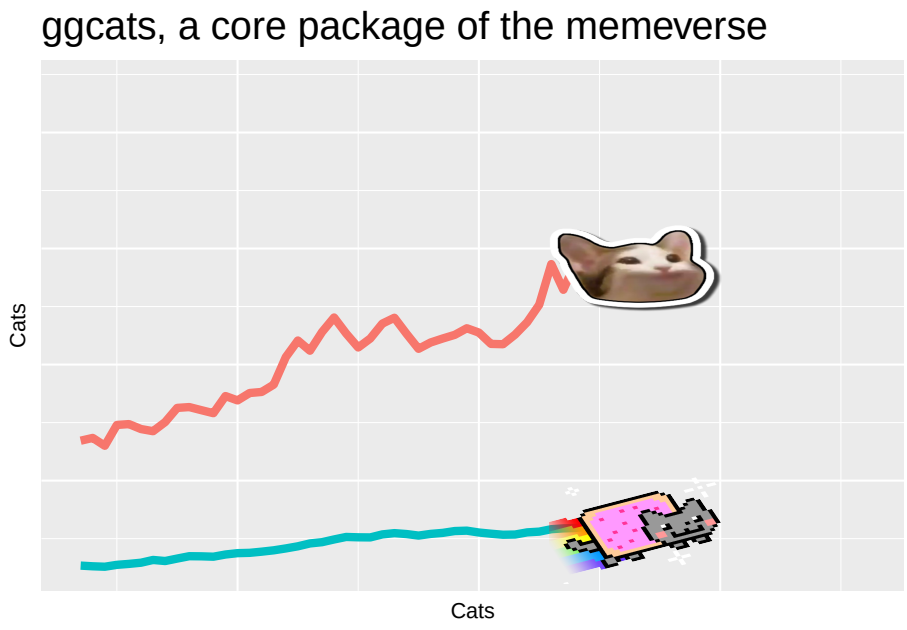

ggcats, a core package of the memeverse
Cats
Cats

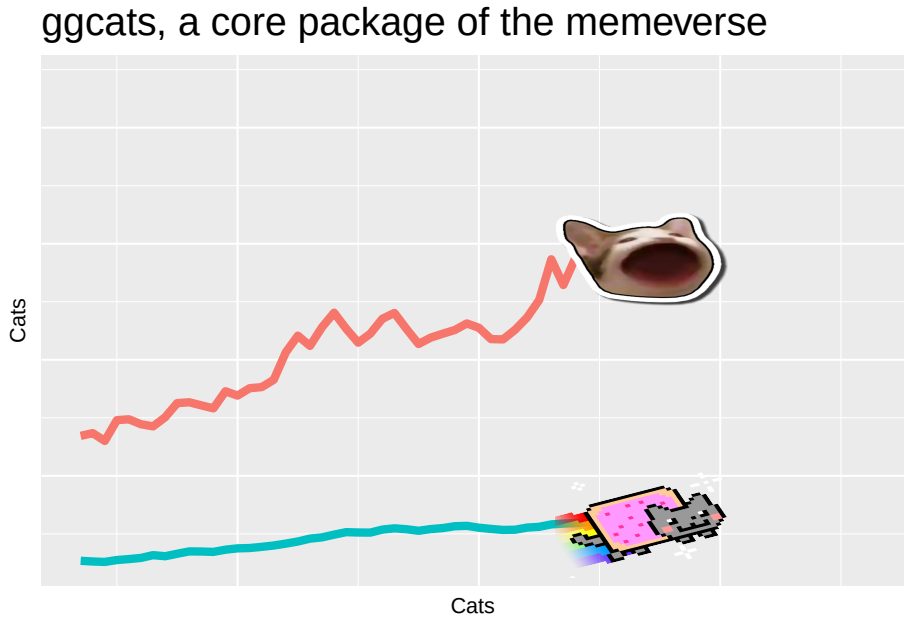

ggcats, a core package of the memeverse
Cats
Cats

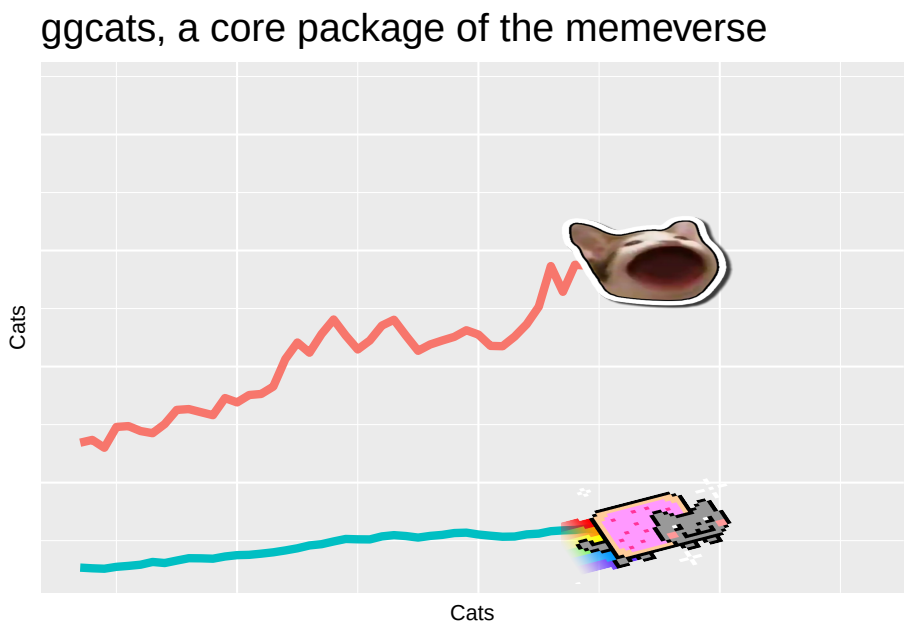

ggcats, a core package of the memeverse
Cats
Cats

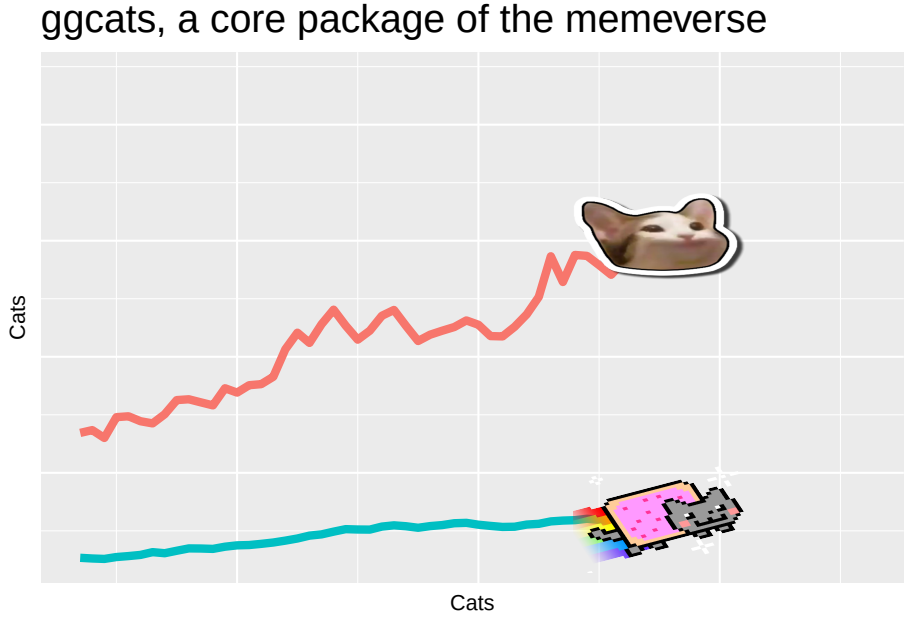

ggcats, a core package of the memeverse
Cats
Cats

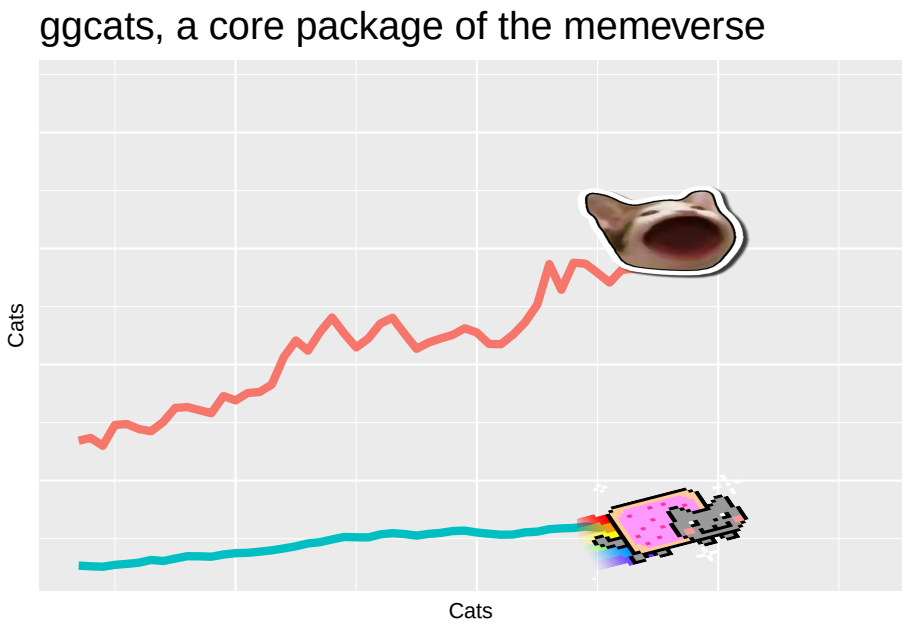

ggcats, a core package of the memeverse
Cats
Cats

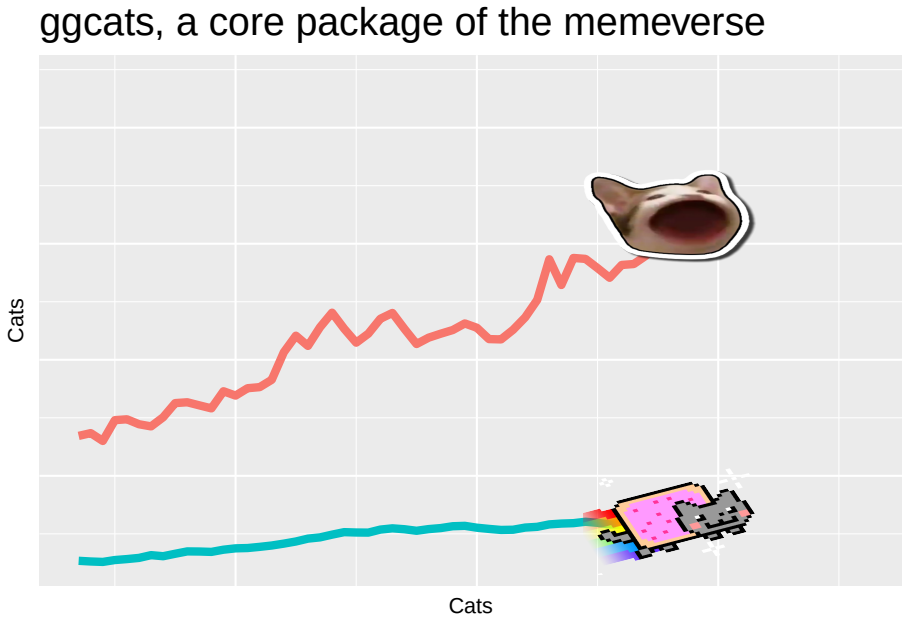

ggcats, a core package of the memeverse
Cats
Cats

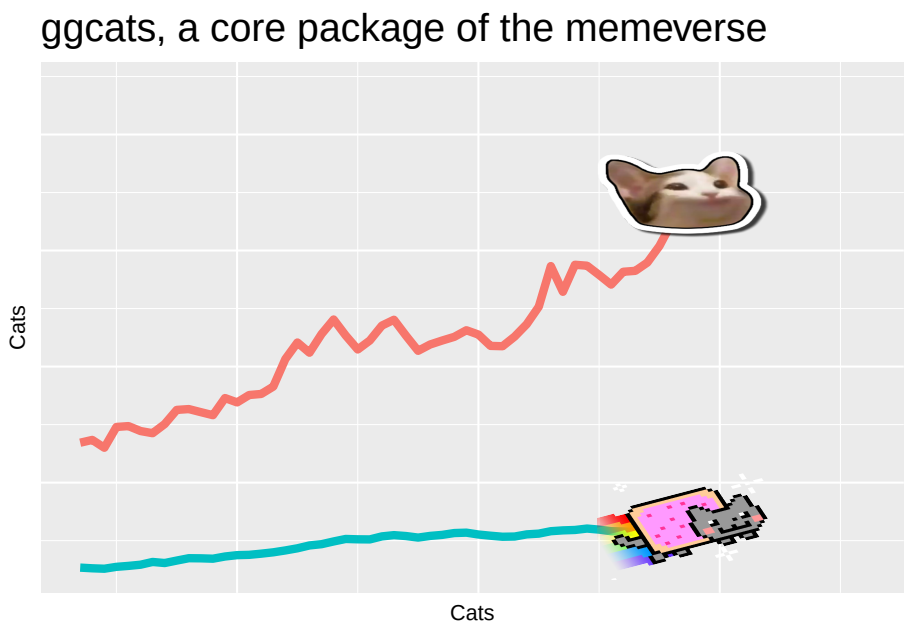

ggcats, a core package of the memeverse
Cats
Cats

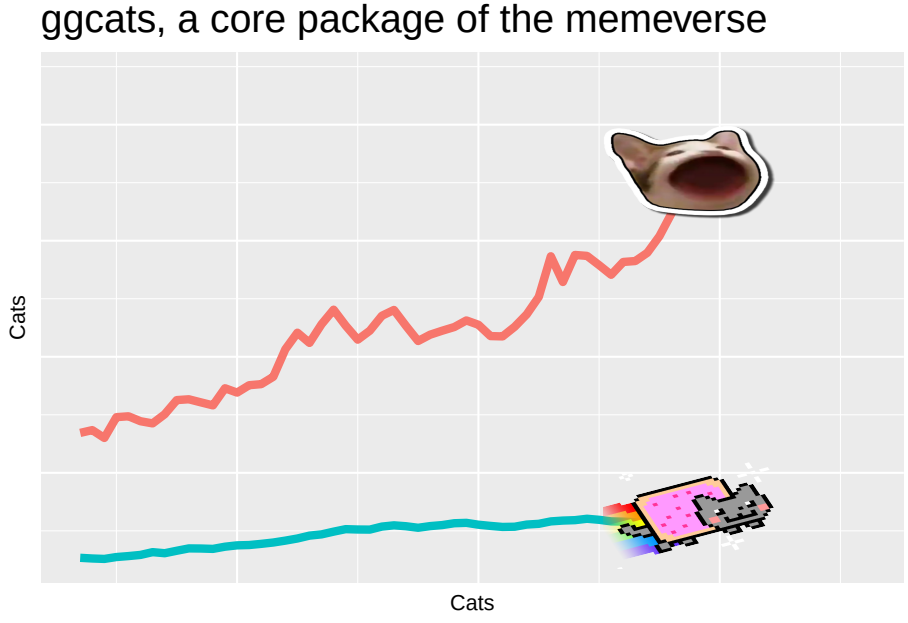

ggcats, a core package of the memeverse
Cats
Cats

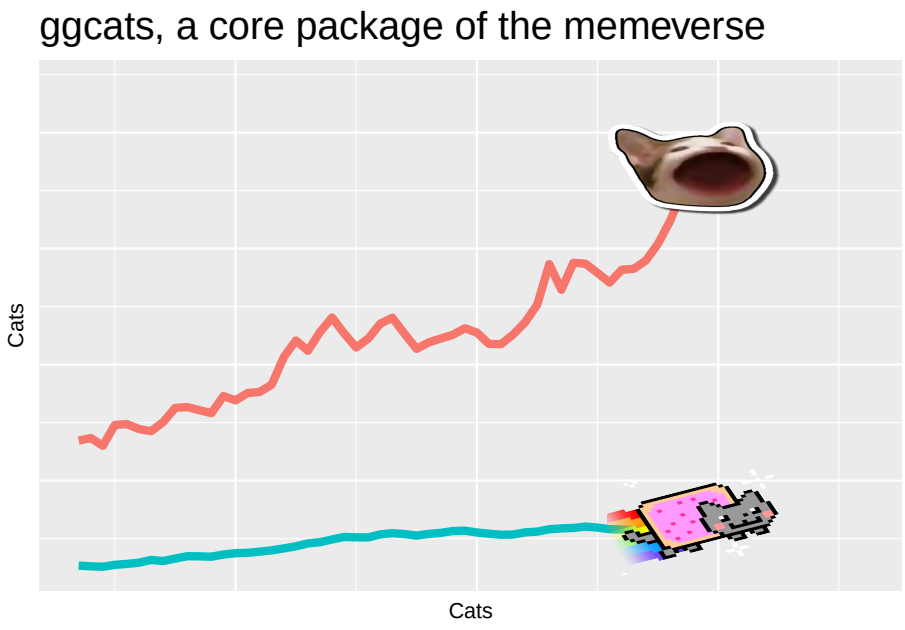

ggcats, a core package of the memeverse
Cats
Cats

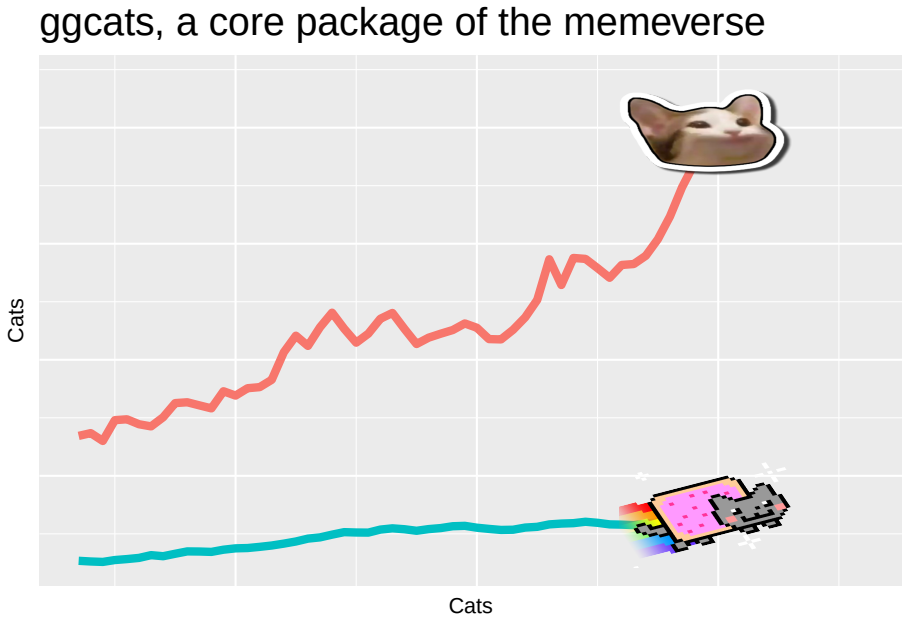

ggcats, a core package of the memeverse
Cats
Cats

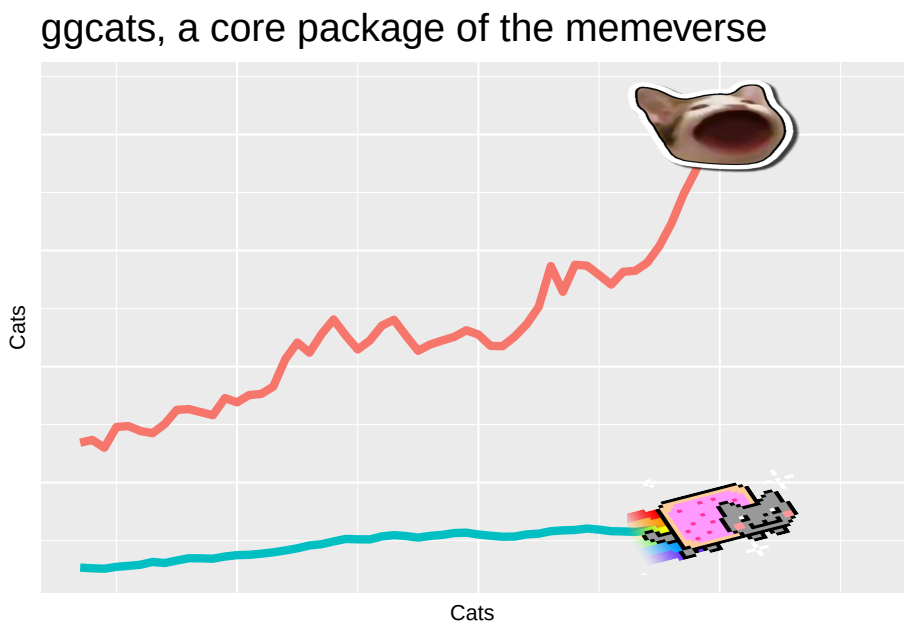

ggcats, a core package of the memeverse
Cats
Cats

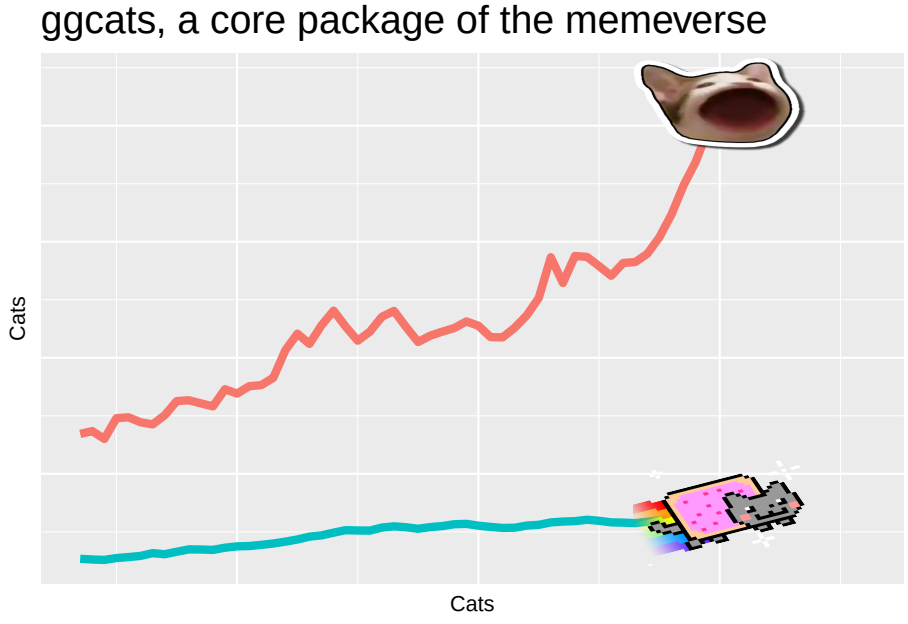

ggcats, a core package of the memeverse
Cats
Cats

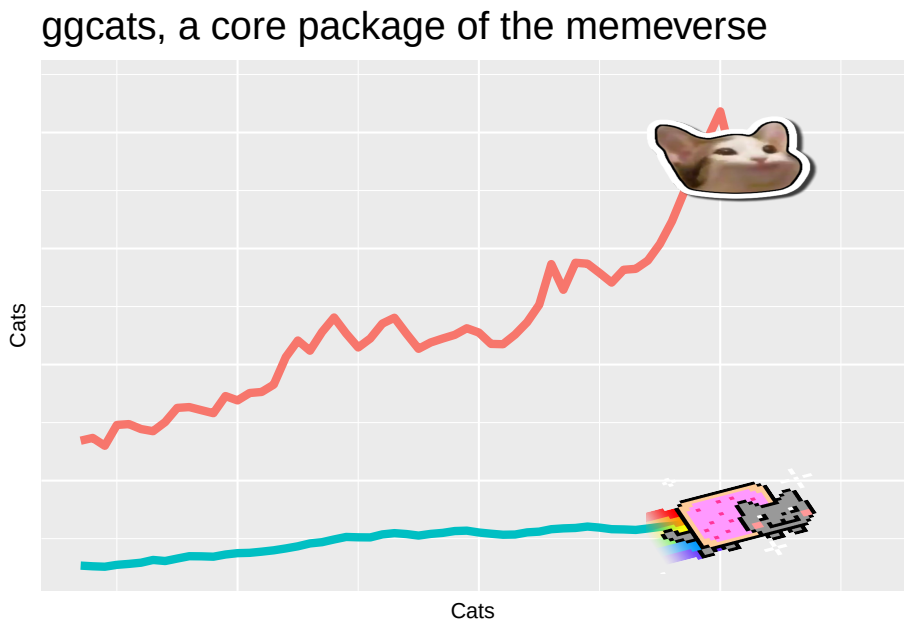

ggcats, a core package of the memeverse
Cats
Cats

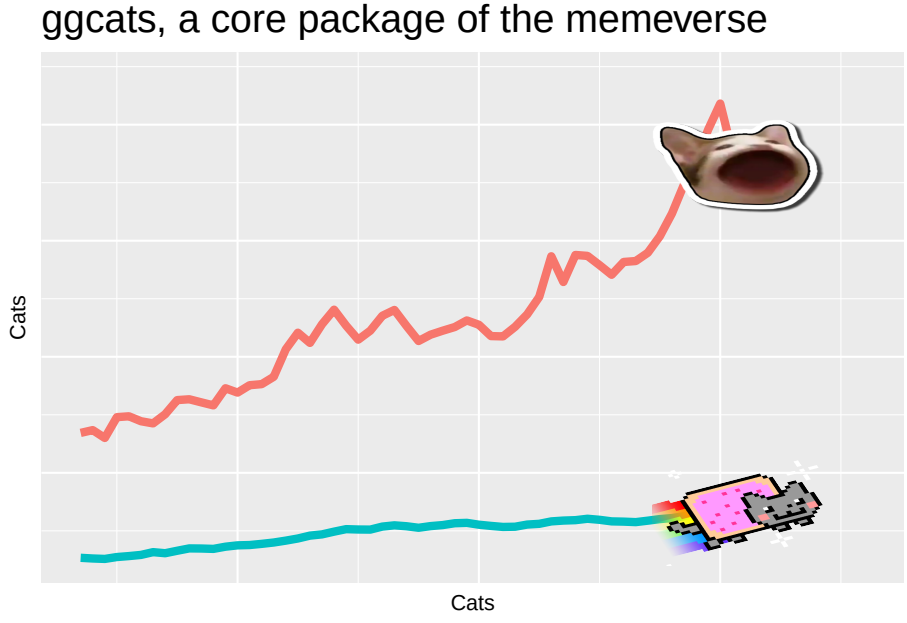

ggcats, a core package of the memeverse
Cats
Cats

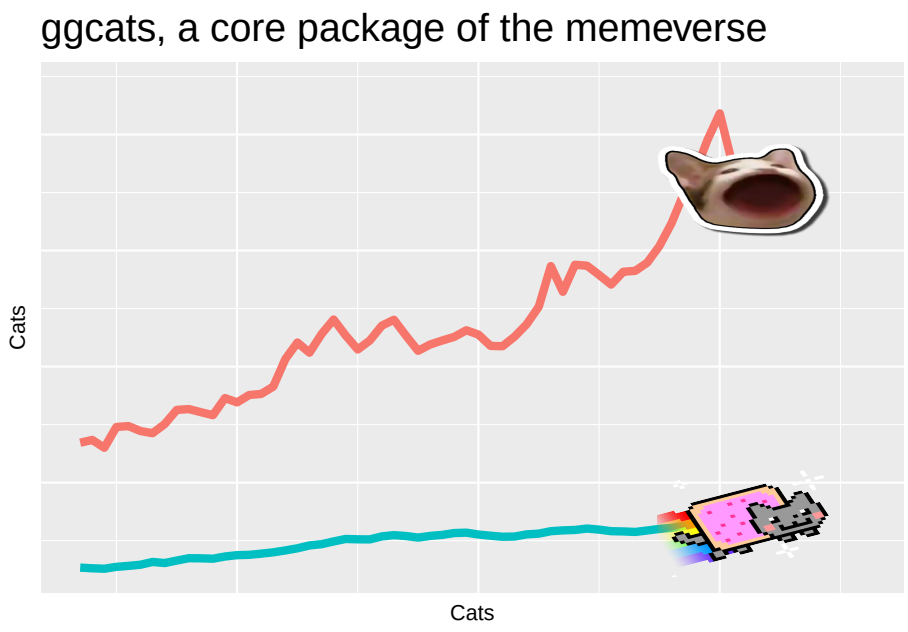

ggcats, a core package of the memeverse
Cats
Cats

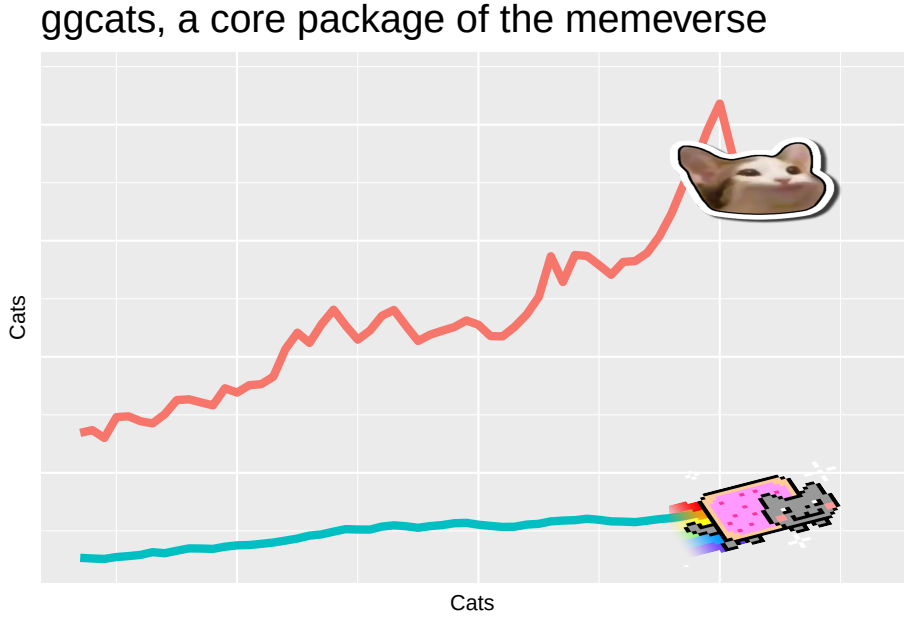

ggcats, a core package of the memeverse
Cats
Cats

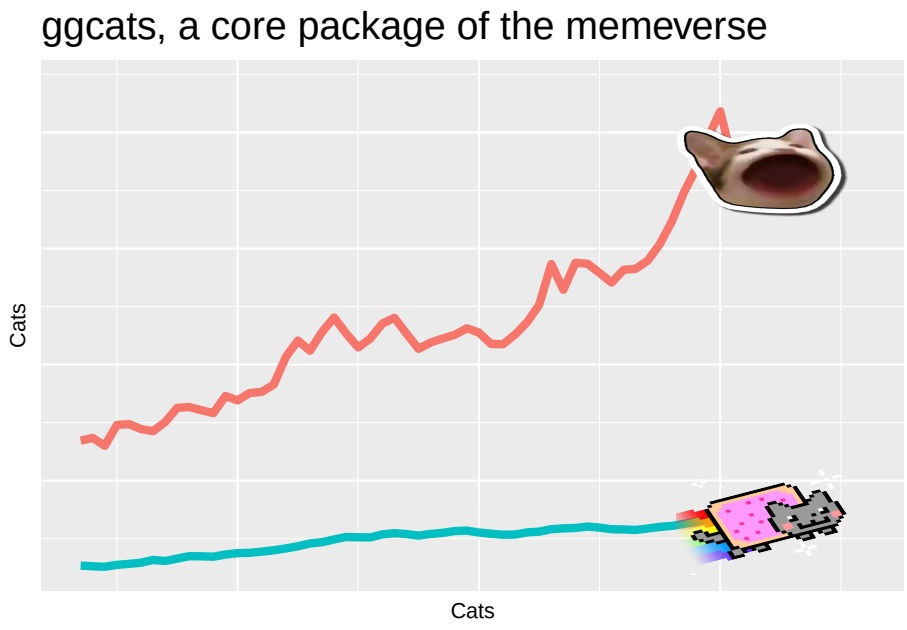

ggcats, a core package of the memeverse
Cats
Cats

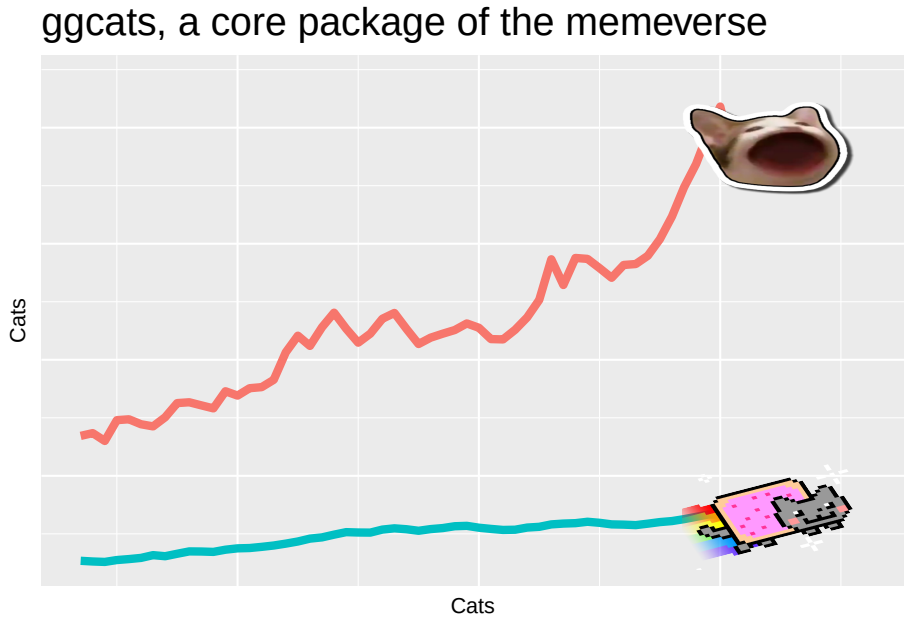

ggcats, a core package of the memeverse
Cats
Cats

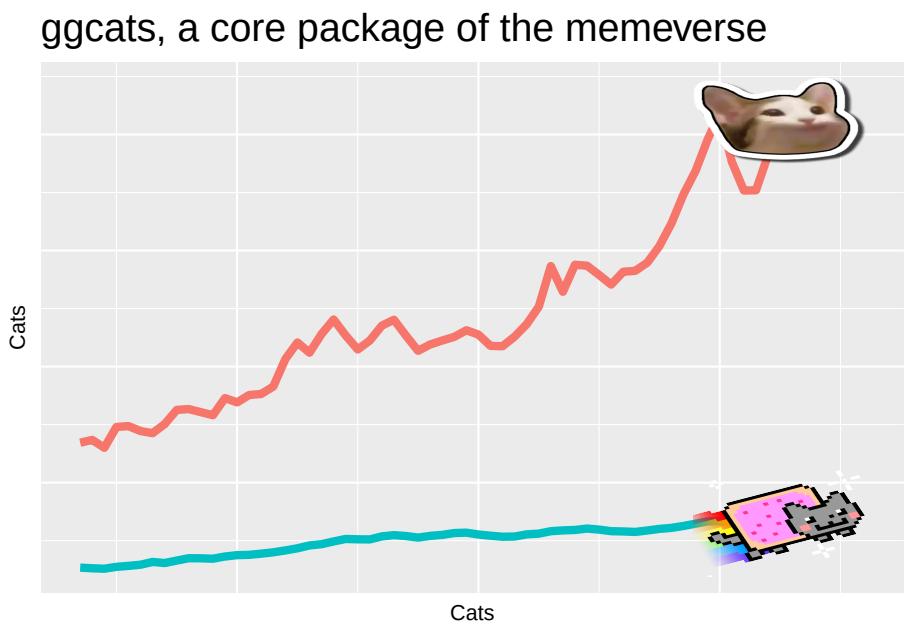

ggcats, a core package of the memeverse
Cats
Cats

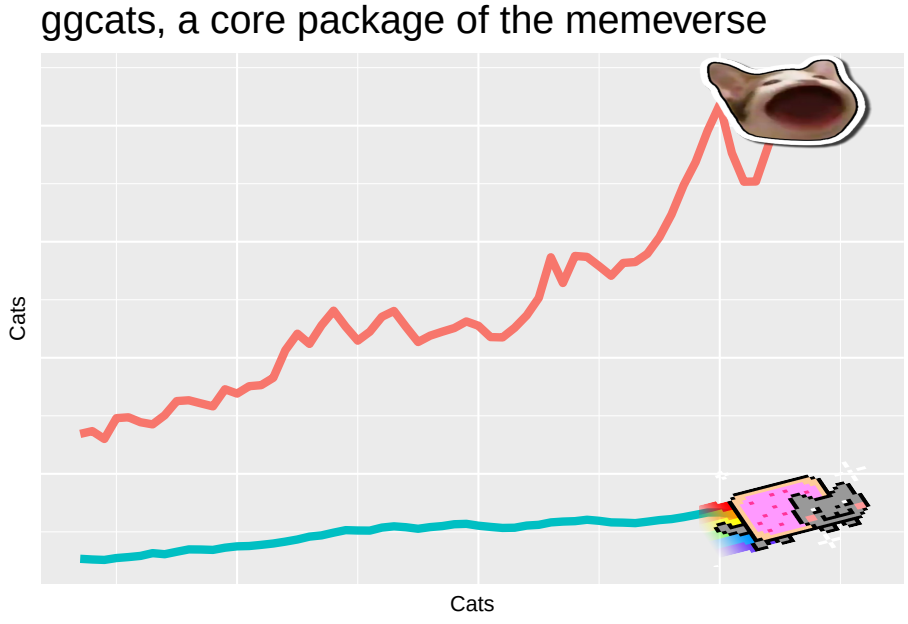

ggcats, a core package of the memeverse
Cats
Cats

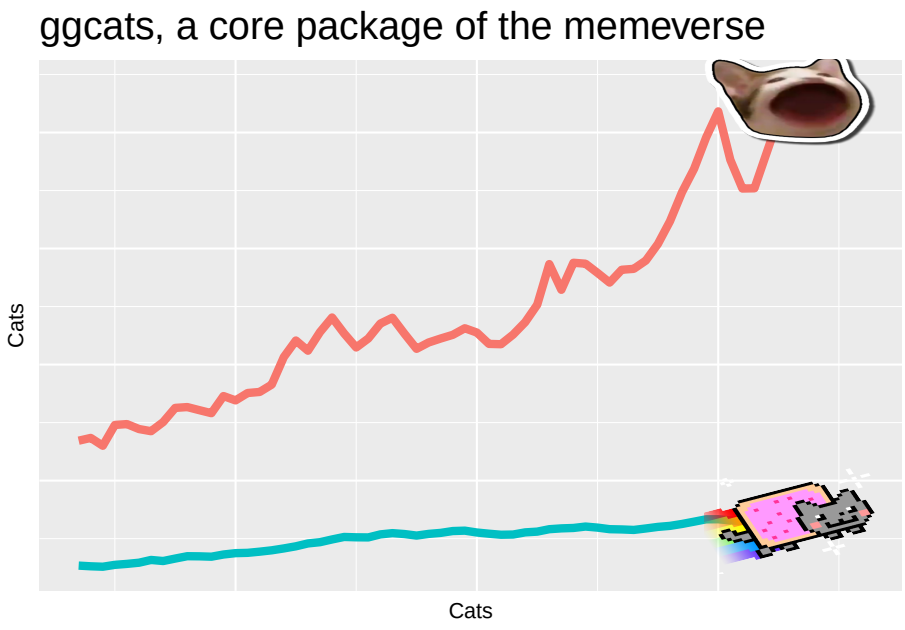

ggcats, a core package of the memeverse
Cats
Cats

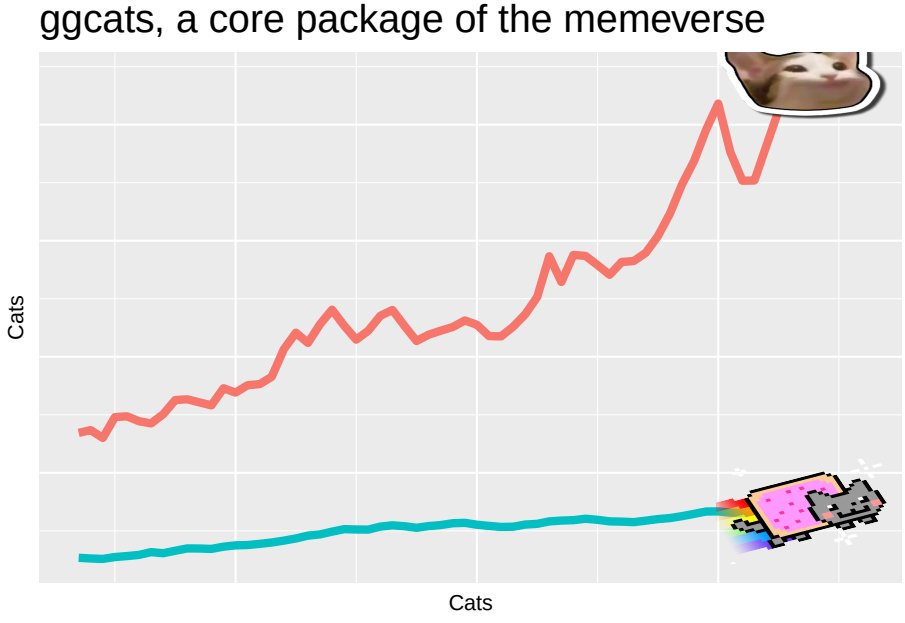

ggcats, a core package of the memeverse
Cats
Cats

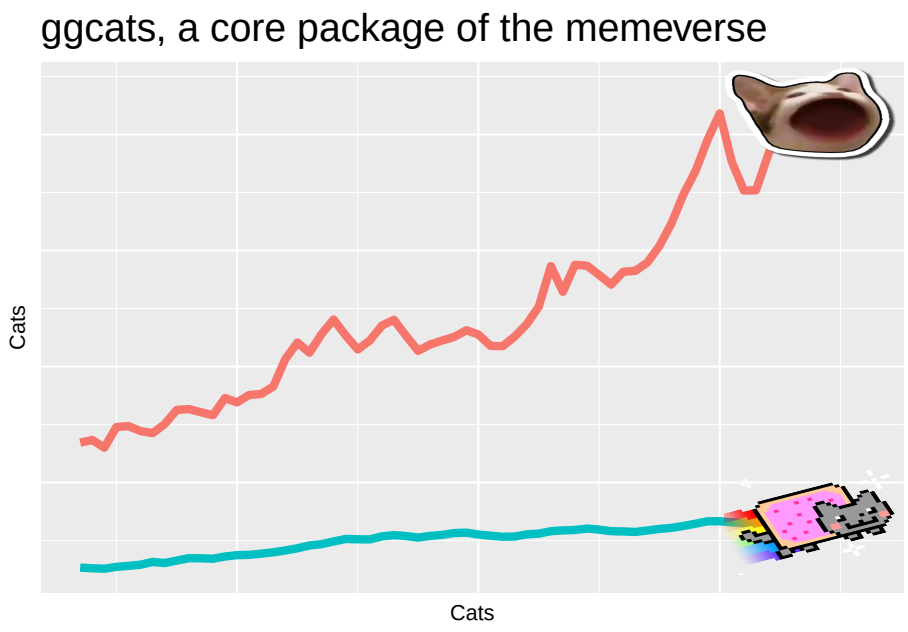

ggcats, a core package of the memeverse
Cats
Cats

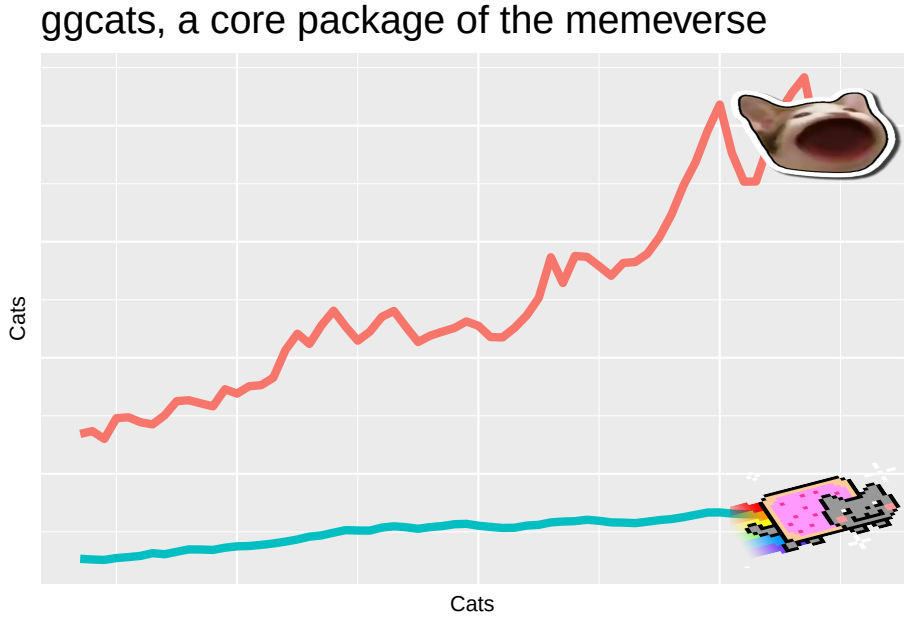

ggcats, a core package of the memeverse
Cats
Cats

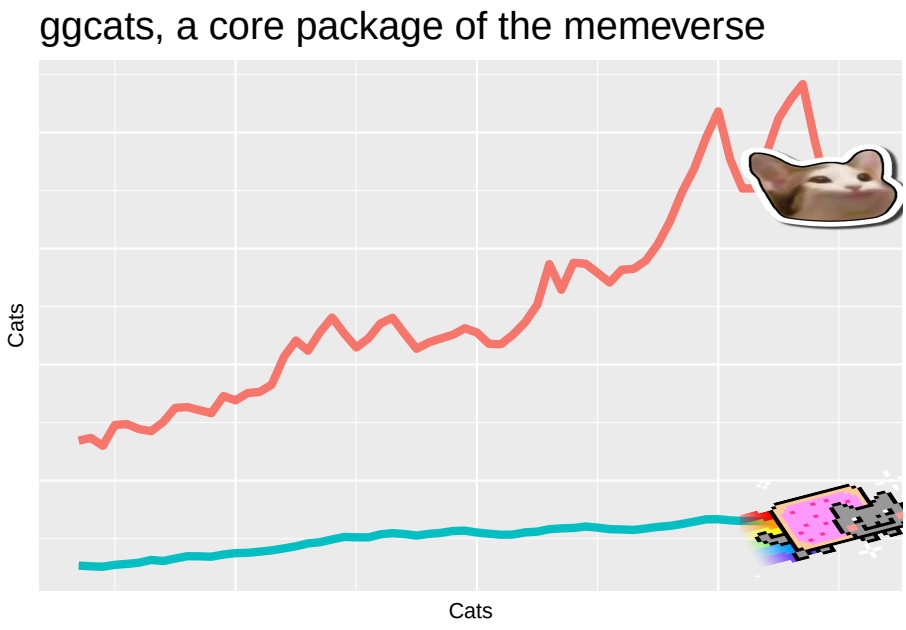

ggcats, a core package of the memeverse
Cats
Cats

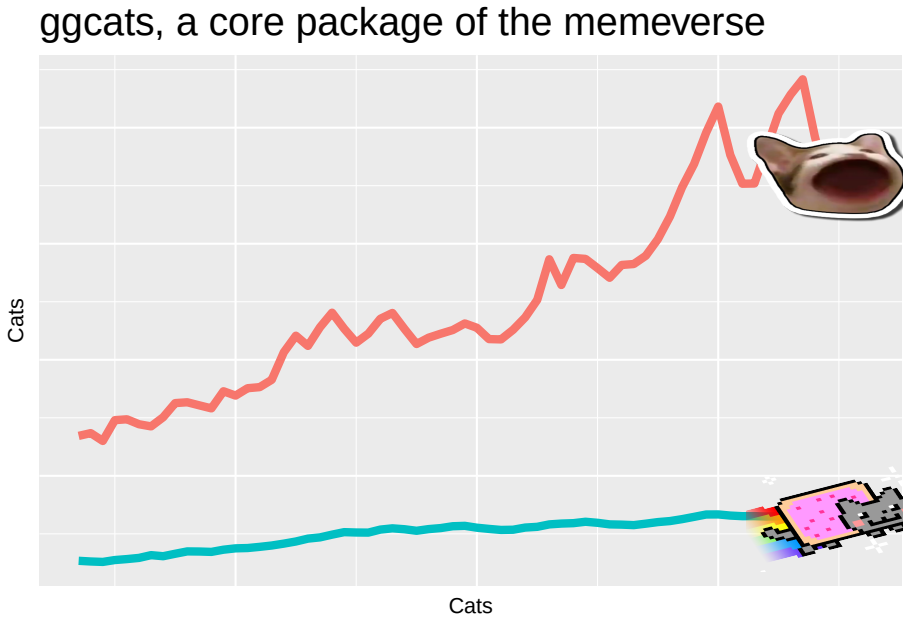

ggcats, a core package of the memeverse
Cats
Cats

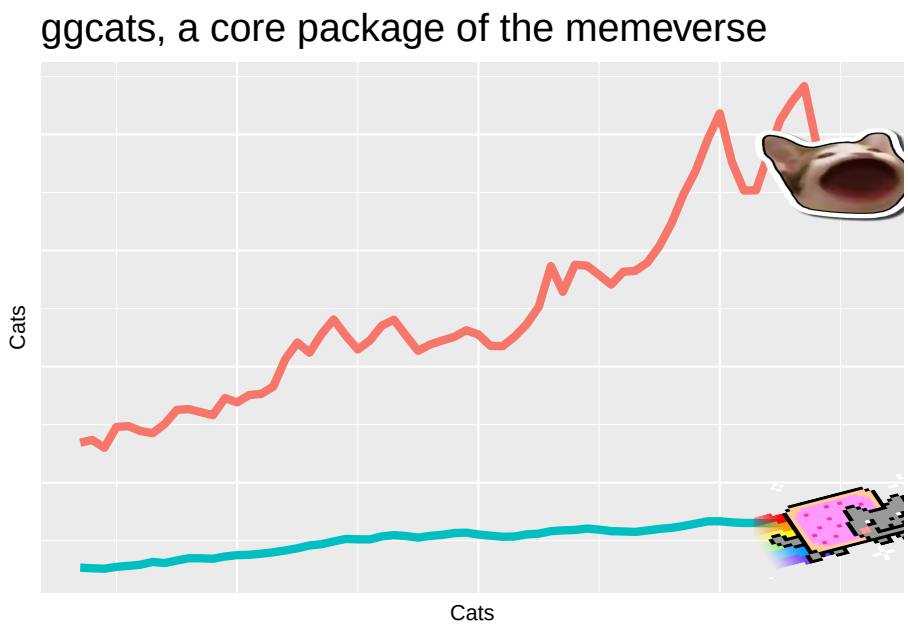

ggcats, a core package of the memeverse
Cats
Cats

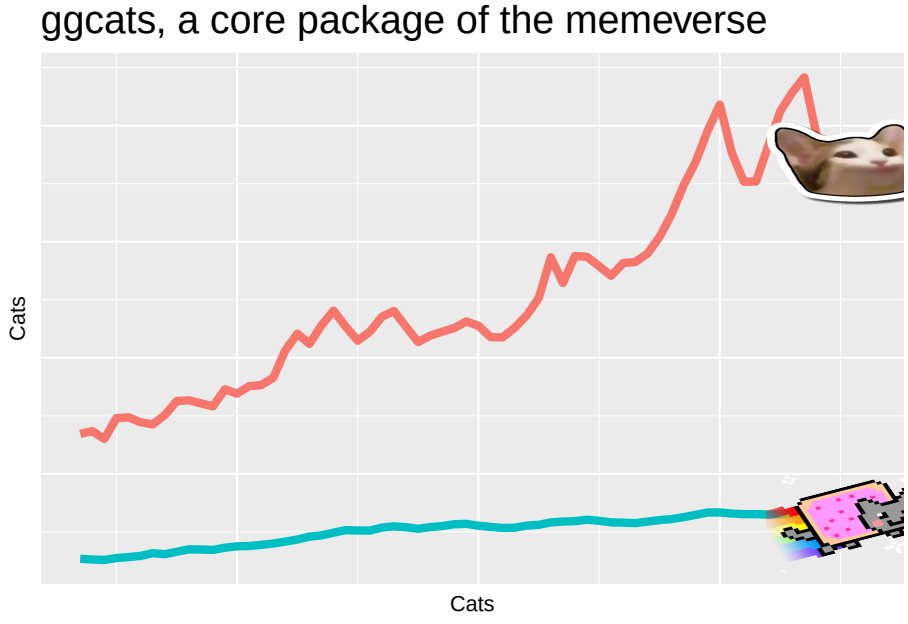

ggcats, a core package of the memeverse
Cats
Cats

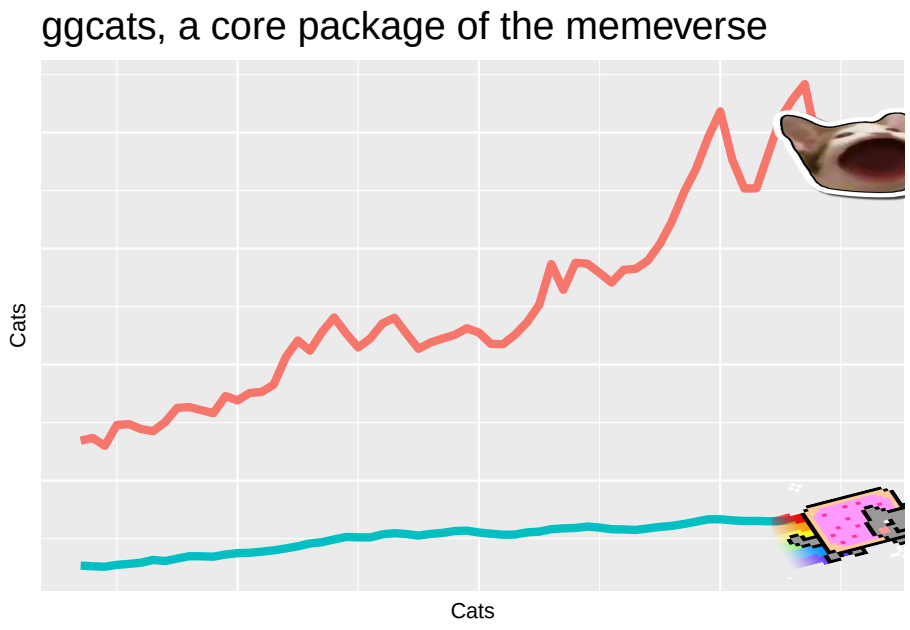

ggcats, a core package of the memeverse
Cats
Cats

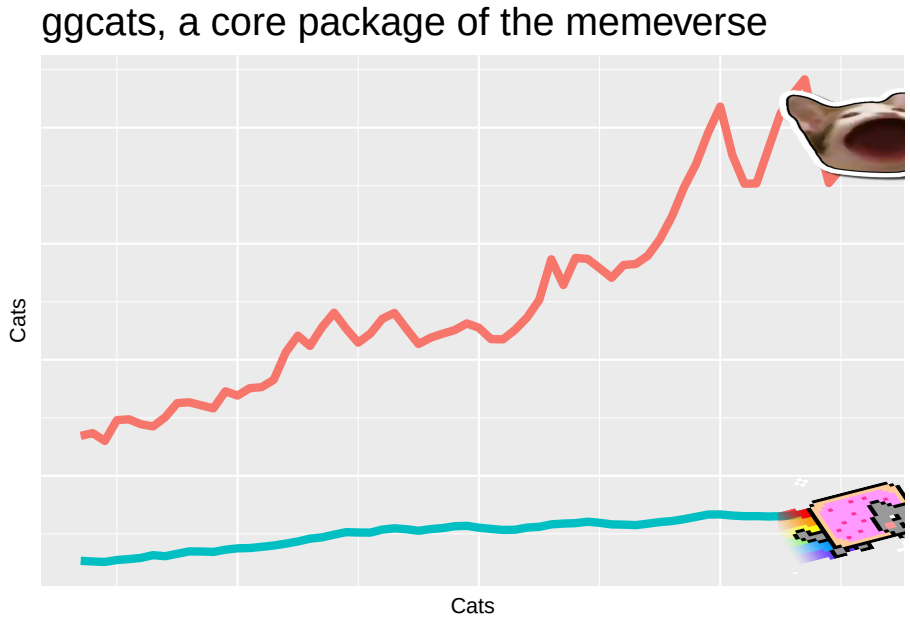

ggcats, a core package of the memeverse
Cats
Cats


ggcats, a core package of the memeverse
Cats
Cats


ggcats, a core package of the memeverse
Cats
Cats


ggcats, a core package of the memeverse
Cats
Cats


ggcats, a core package of the memeverse
Cats
Cats


ggcats, a core package of the memeverse
Cats
Cats

Chapter 11

Deeper data insights part 2: Week Eleven

In the previous chapter we looked at individual variables, and understanding the different types of data. We made numeric and graphical summaries of the distributions of features within each variable. This week we will continue to work in the same space, and extend our understanding to include relationships between variables.

Understanding the relationship between two or more variables is often the basis of most of our scientific questions. These might include comparing variables of the same type (numeric against numeric) or different types (numeric against categorical). In this chapter we will see how we can use descriptive statistics and visuals to explore associations

11.1 Associations between numerical variables

11.1.1 Correlations

A common measure of association between two numerical variables is the **correlation coefficient**. The correlation metric is a numerical measure of the *strength of an association*

There are several measures of correlation including:

- **Pearson's correlation coefficient** : good for describing linear associations
- **Spearman's rank correlation coefficient**: a rank ordered correlation - good for when the assumptions for Pearson's correlation is not met.

Pearson's correlation coefficient r is designed to measure the strength of a linear (straight line) association. Pearson's takes a value between -1 and 1.

- A value of 0 means there is no linear association between the variables
- A value of 1 means there is a perfect *positive* association between the variables
- A value of -1 means there is a perfect *negative* association between the variables

A *perfect* association is one where we can predict the value of one variable with complete accuracy, just by knowing the value of the other variable.

We can use the `cor` function in R to calculate Pearson's correlation coefficient.

```
library(rstatix)

penguins %>%
  cor_test(bill_length_mm, bill_depth_mm)
```

```
## # A tibble: 1 x 8
##   var1           var2           cor statistic       p conf.low conf.high method
##   <chr>          <chr>        <dbl>     <dbl>   <dbl>    <dbl>     <dbl> <chr>
## 1 bill_length_mm bill_depth_mm -0.24     -4.46 1.12e-5   -0.333    -0.132 Pears~
```

This tells us two features of the association. It's *sign* and *magnitude*. The coefficient is negative, so as bill length increases, bill depth decreases. The value -0.22 indicates that only about 22% of the variation in bill length can be explained by changes in bill depth (and *vice-versa*), suggesting that the variables are not closely related.

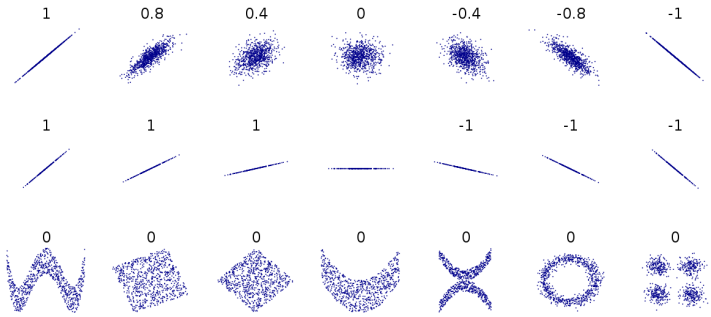


Figure 11.1: Different relationships between two numeric variables. Each number represents the Pearson's correlation coefficient of each association

- Because Pearson's coefficient is designed to summarise the strength of a linear relationship, this can be misleading if the relationship is *not linear*

e.g. curved or humped. This is why it's always a good idea to plot the relationship *first* (see above).

- Even when the relationship is linear, it doesn't tell us anything about the steepness of the association (see above). It *only* tells us how often a change in one variable can predict the change in the other *not* the value of that change.

This can be difficult to understand at first, so carefully consider the figure above.

- The first row above shows differing levels of the strength of association. If we drew a perfect straight line between two variables, how closely do the data points fit around this line.
- The second row shows a series of *perfect* linear relationships. We can accurately predict the value of one variable just by knowing the value of the other variable, but the steepness of the relationship in each example is very different. This is **important** because it means a perfect association can still have a small effect.
- The third row shows a series of associations where there is *clearly* a relationship between the two variables, but it is also not linear so would be inappropriate for a Pearson's correlation.

11.1.2 Non-linear correlations

So what should we do if the relationship between our variables is non-linear? Instead of using Pearson's correlation coefficient we can calculate something called a **rank correlation**.

Instead of working with the raw values of our two variables we can use rank ordering instead. The idea is pretty simple if we start with the lowest vaule in a variable and order it as '1', then assign labels '2', '3' etc. as we ascend in rank order. We can see a way that this could be applied manually with the function `dense_rank` from dplyr below:

```
penguins %>% select(bill_length_mm,
                    bill_depth_mm) %>%
  drop_na() %>%
  mutate(rank_length=dense_rank((bill_length_mm)),
         rank_depth=dense_rank((bill_depth_mm)))
```

```
## # A tibble: 342 x 4
##    bill_length_mm bill_depth_mm rank_length rank_depth
##             <dbl>         <dbl>       <int>      <int>
##  1           39.1          18.7          43         57
##  2           39.5          17.4          46         44
##  3           40.3          18            52         50
```

```
##  4           36.7          19.3          23          63
##  5           39.3          20.6          45          75
##  6           38.9          17.8          41          48
##  7           39.2          19.6          44          66
##  8           34.1          18.1           5          51
##  9           42            20.2          66          72
## 10           37.8          17.1          32          41
## # ... with 332 more rows
```

Measures of rank correlation then are just a comparison of the rank orders between two variables, with a value between -1 and 1 just like Pearsons's. We already know from our Pearson's correlation coefficient, that we expect this relationship to be negative. So it should come as no surprise that the highest rank order values for bill_length_mm appear to be associated with lower rank order values for bill_depth_mm.

To calculate Spearman's ρ 'rho' is pretty easy, you can use the cor functions again, but this time specify a hidden argument to `method="spearman"`.

```
penguins %>%
  cor_test(bill_length_mm, bill_depth_mm, method="spearman")
```

```
## # A tibble: 1 x 6
##   var1           var2            cor statistic         p method
##   <chr>          <chr>         <dbl>     <dbl>     <dbl> <chr>
## 1 bill_length_mm bill_depth_mm -0.22  8145268. 0.0000351 Spearman
```

What we can see in this example is that Pearson's r and Spearman's ρ are basically identical.

11.1.3 Graphical summaries between numeric variables

Correlation coefficients are a quick and simple way to attach a metric to the level of association between two variables. They are limited however in that a single number can never capture the every aspect of their relationship. This is why we visualise our data.

We have already covered scatter plots and `ggplot2()` extensively in previous chapters, so here we will just cover some of the different ways in which you could present the nature of a relationship

```
length_depth_scatterplot <- ggplot(penguins, aes(x= bill_length_mm,
                      y= bill_depth_mm)) +
    geom_point()

length_depth_scatterplot
```

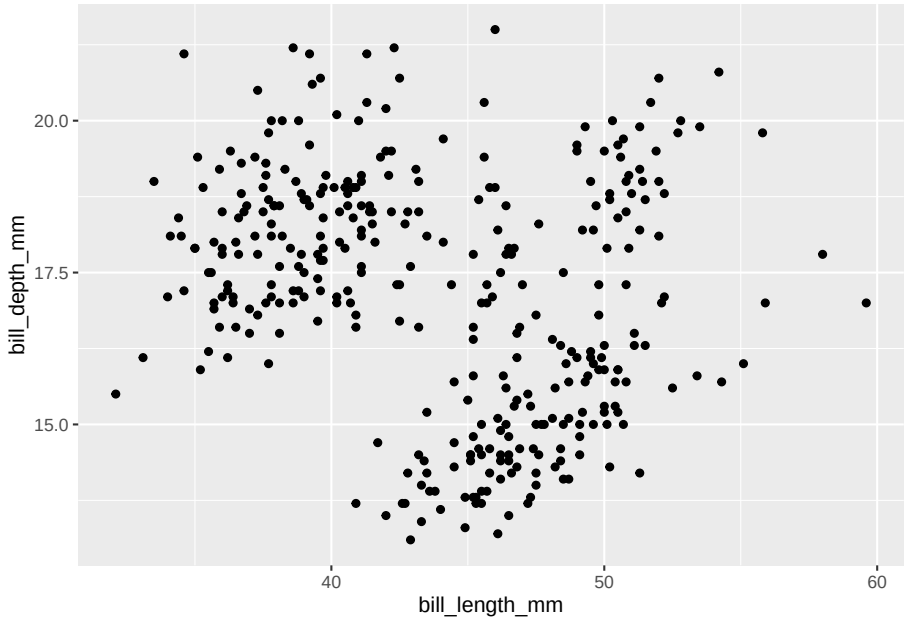


Figure 11.2: A scatter plot of bill depth against bill length in mm

**Note - Remember there are a number of different options available when constructing a plot including changing alpha to produce transparency if plots are lying on top of each other, colours (and shapes) to separate subgroups and ways to present third numerical variables such as setting aes(size=body_mass_g).

```
library(patchwork) # package calls should be placed at the TOP of your script

bill_depth_marginal <- penguins %>%
  ggplot()+
  geom_density(aes(x=bill_depth_mm), fill="darkgrey")+
  theme_void()+
  coord_flip() # this graph needs to be rotated

bill_length_marginal <- penguins %>%
  ggplot()+
  geom_density(aes(x=bill_length_mm), fill="darkgrey")+
  theme_void()

layout <- "
```

```
AA#
BBC
BBC"
# layout is easiest to organise using a text distribution, where ABC equal the three p

# layout <- "
# AAA#
# BBBC
# BBBC"
# BBBC

bill_length_marginal+length_depth_scatterplot+bill_depth_marginal+ # order of plots is
  plot_layout(design=layout) # uses the layout argument defined above to arrange the s
```

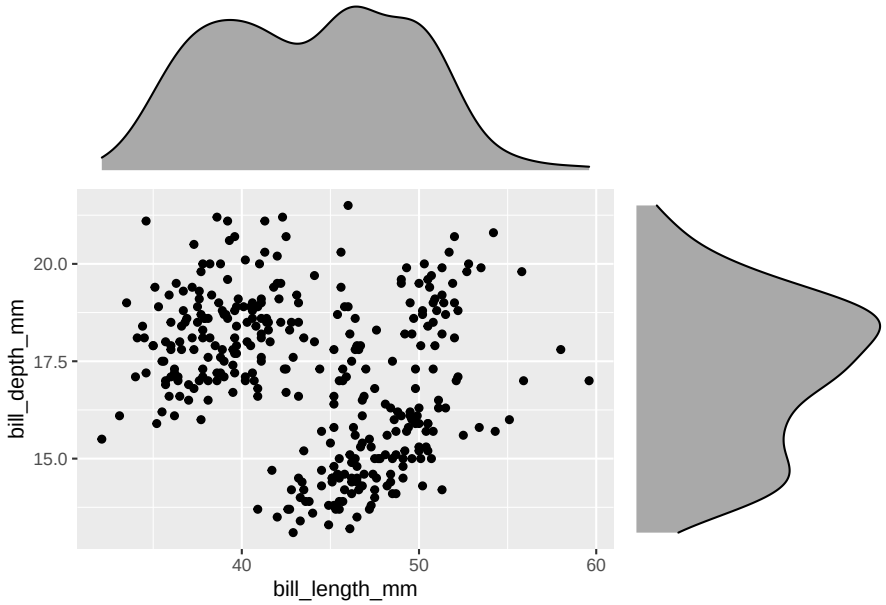


Figure 11.3: Using patchwork we can easily arrange extra plots to fit as marginals - these could be boxplots, histograms or density plots

These efforts allow us to capture details about the spread and distribution of both variables **and** how they relate to each other. This figure provides us with insights into

- The central tendency of each variable

- The spread of data in each variable
- The correlation between the two variables

11.2 Associations between categorical variables

Exploring associations between different categorical variables is not quite as simple as the previous numeric-numeric examples. Generally speaking we are interested in whether different combinations of categories are uniformally distributed or show evidence of clustering leading to *over- or under-represented* combinations. The simplest way to investigate this is to use `group_by` and `summarise` as we have used previously.

```
island_species_summary <- penguins %>%
  group_by(island, species) %>%
  summarise(n=n(),
            n_distinct=n_distinct(penguin_id)) %>%
  ungroup() %>% # needed to remove group calculations
  mutate(freq=n/sum(n)) # then calculates percentage of each group across WHOLE dataset

island_species_summary
```

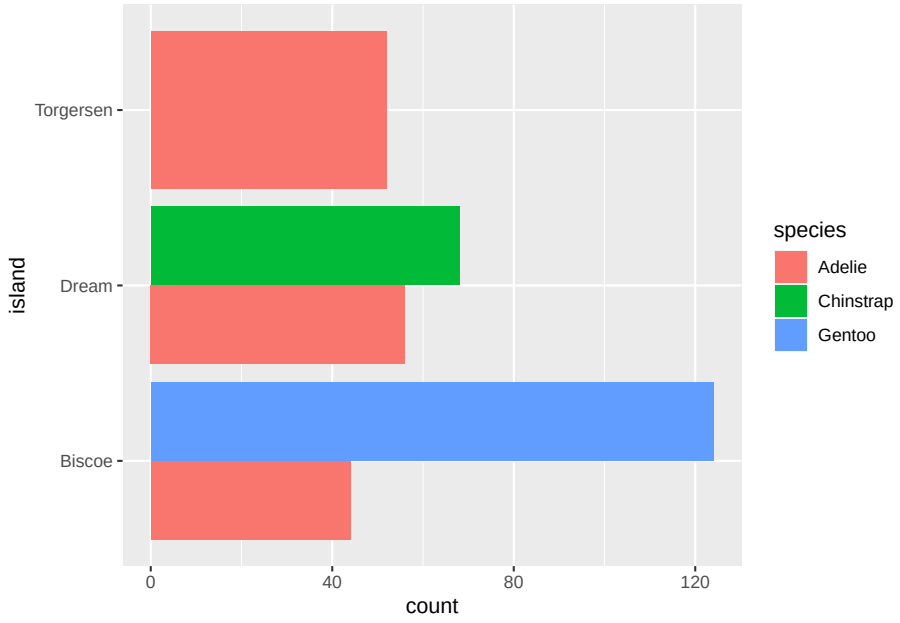
```
## # A tibble: 5 x 5
##   island    species       n n_distinct  freq
##   <chr>     <chr>     <int>      <int> <dbl>
## 1 Biscoe    Adelie       44         44 0.128
## 2 Biscoe    Gentoo      124         94 0.360
## 3 Dream     Adelie       56         56 0.163
## 4 Dream     Chinstrap    68         58 0.198
## 5 Torgersen Adelie       52         52 0.151
```

**Note - remember that group_by() applies functions which comes after it in a group-specific pattern.

What does the above tell us, that 168 observations were made on the Island of Biscoe, with three times as many Gentoo penguin observations made as Adelie penguins (remeber this is observations made, not individual penguins). When we account for penguin ID we see there are around twice as many Gentoo penguins recorded. We can see there are no Chinstrap penguins recorded on Biscoe. Conversely we can see that Gentoo penguins are **only** observed on Biscoe. The island of Dream has two populations of Adelie and Chinstrap penguins of roughly equal size, while the island of Torgensen appears to have a population comprised only of Adelie penguins.

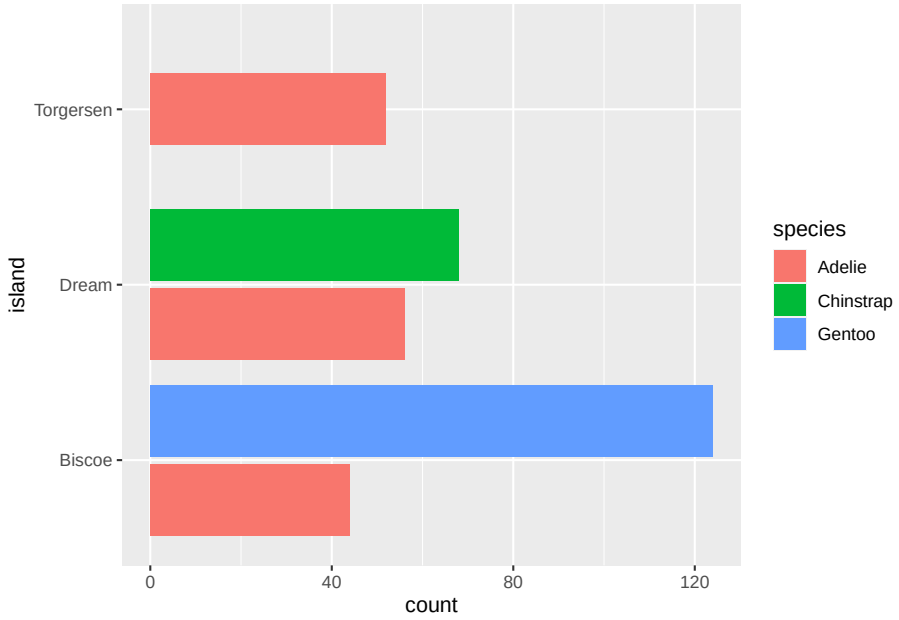
We could also use a bar chart in ggplot to represent this count data.

```
penguins%>%
  ggplot(aes(x=island, fill=species))+
  geom_bar(position=position_dodge())+
  coord_flip()
```



This is fine, but it looks a bit odd, because the bars expand to fill the available space on the category axis. Luckily there is an advanced version of the postion_dodge argument.

```
penguins%>%
  ggplot(aes(x=island, fill=species))+
  geom_bar(position=position_dodge2(preserve="single"))+
  #keeps bars to appropriate widths
  coord_flip()
```



> **Note the default for bar charts would have been a stacked option, but we have already seen how that can produce graphs that are difficult to read.

An alternative approach would be to look at the 'relative proportions' of each population in our overall dataset. Using the same methods as we used previously when looking at single variables. Let's add in a few aesthetic tweaks to improve the look.

```
penguins %>%
  ggplot(aes(x=island, fill=species))+
  geom_bar(position=position_dodge2(preserve="single"))+
  #keeps bars to appropriate widths
    labs(x="Island",
       y = "Number of observations")+
  geom_text(data=island_species_summary, # use the data from the summarise object
            aes(x=island,
                y= n+10, # offset text to be slightly to the right of bar
                group=species, # need species group to separate text
                label=scales::percent(freq) # automatically add %
                ),
            position=position_dodge2(width=0.8))+ # set width of dodge
  scale_fill_manual(values=c("cyan",
                             "darkorange",
                             "purple"
                             ))+
```

```
  coord_flip()+
  theme_minimal()+
  theme(legend.position="bottom") # put legend at the bottom of the graph
```

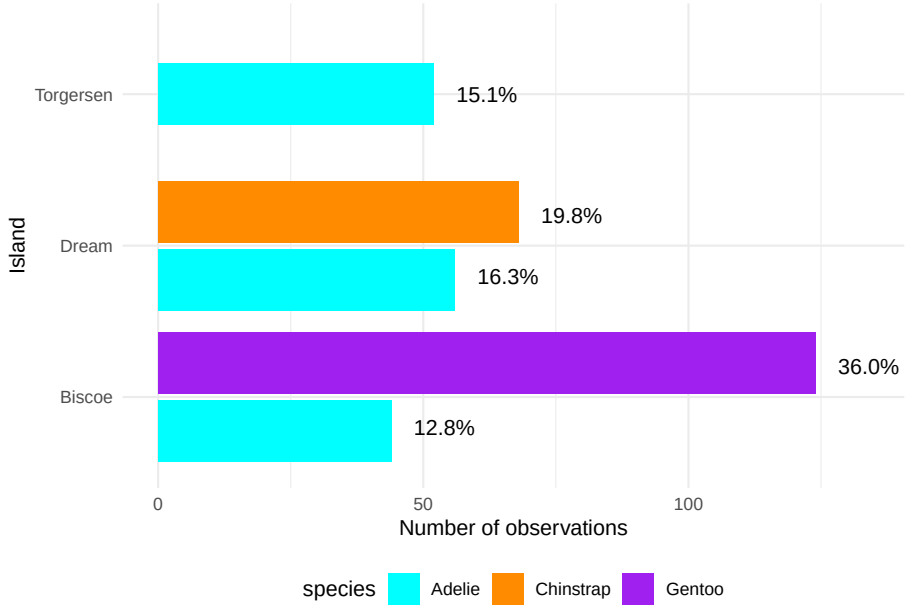
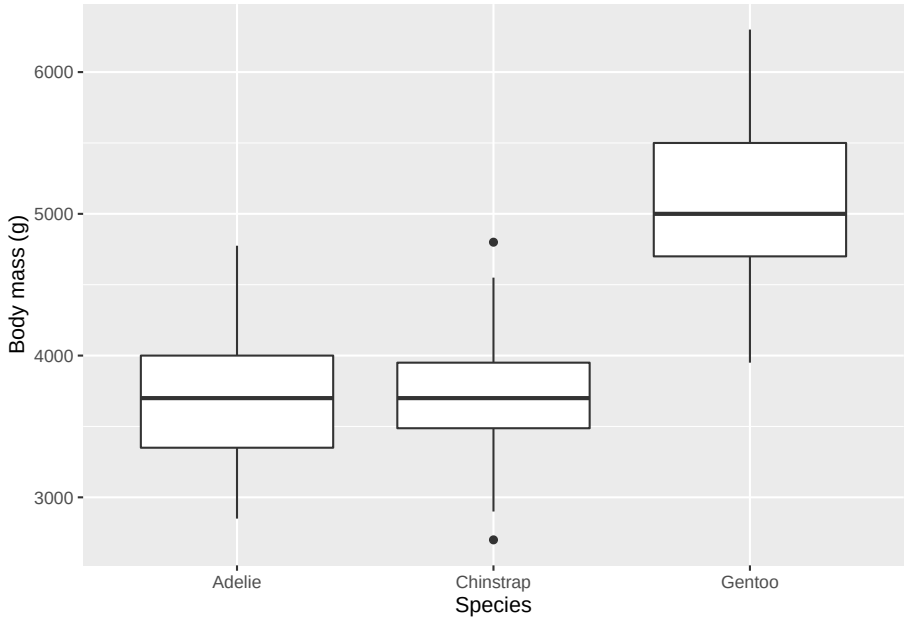


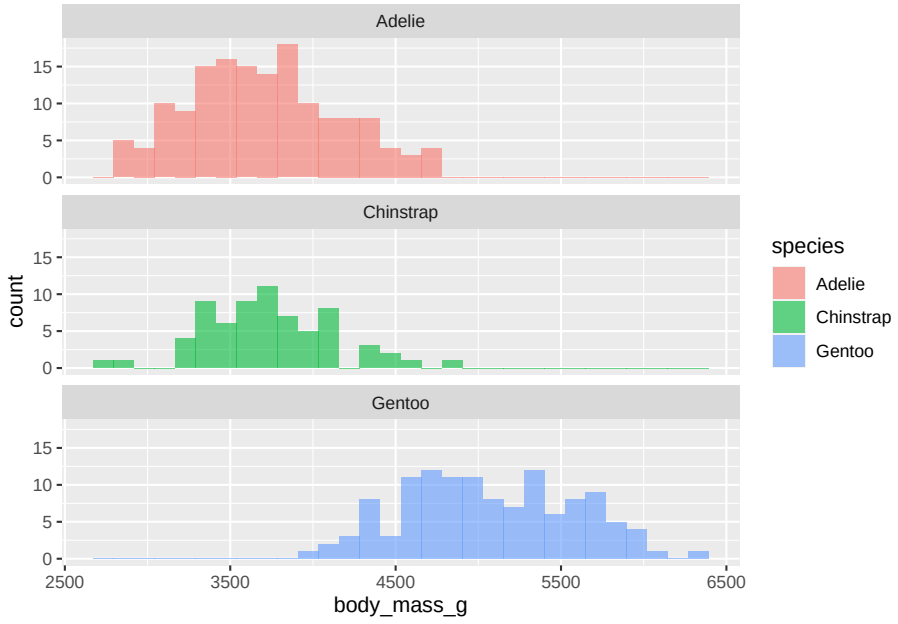
Figure 11.4: A dodged barplot showing the numbers and relative proportions of data observations recorded by penguin species and location

11.3 Associations between Categorical-numerical variables

```
penguins %>%
  ggplot(aes(x=species,
             y=body_mass_g))+
  geom_boxplot()+
  labs(y="Body mass (g)",
         x= "Species")
```



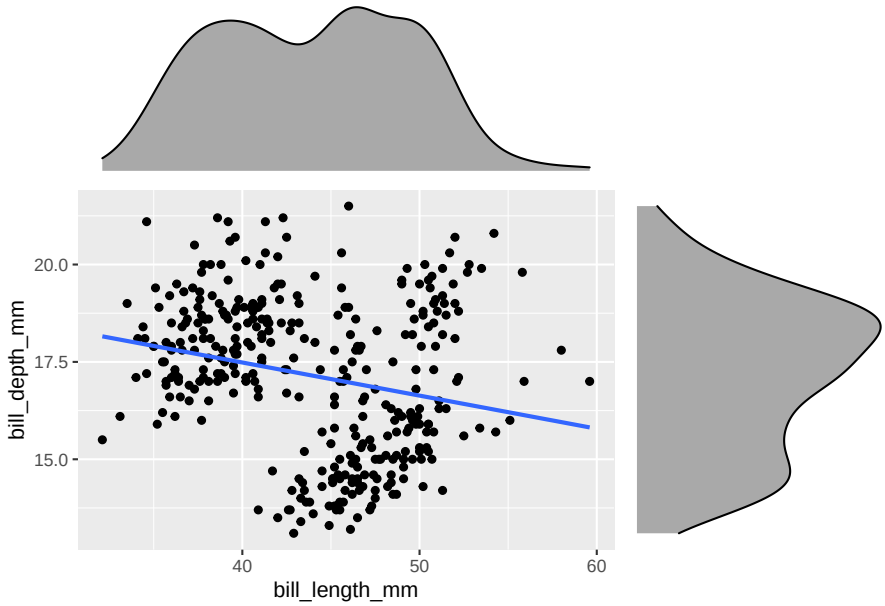

```
penguins %>%
  ggplot(aes(x=body_mass_g,
             fill=species))+
  geom_histogram(alpha=0.6,
         bins=30,
         position="identity")+
  facet_wrap(~species,
             ncol=1)
```



11.4 Complexity

11.4.1 Simpson's Paradox

Remember when we first correlated bill length and bill depth against each other we found an overall negative correlation of -0.22. However, this is because of a confounding variable we had not accounted for - species.

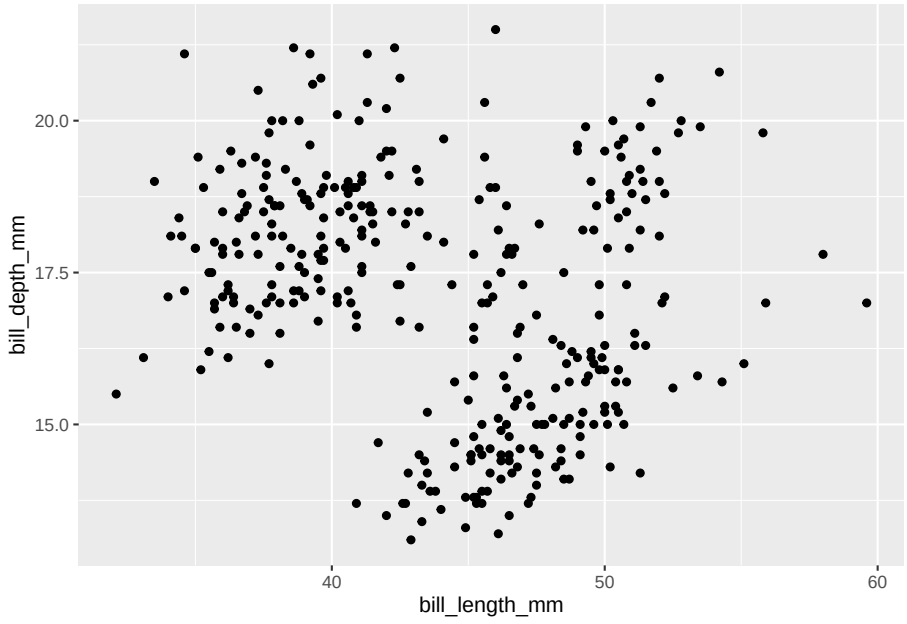


This is another example of why carefully studying your data - and carefully considering those variables which are likely to affect each other are studied or controlled for. It is an entirely reasonable hypothesis that different penguin species might have different bill shapes that might make an overall trend misleading. We can easily check the effect of a categoricial variable on our two numeric variables by assigning the aesthetic colour.

```
colours <- c("cyan",
             "darkorange",
             "purple")

length_depth_scatterplot_2 <- ggplot(penguins, aes(x= bill_length_mm,
                     y= bill_depth_mm,
                     colour=species)) +
    geom_point()+
  geom_smooth(method="lm",
              se=FALSE)+
  scale_colour_manual(values=colours)+
  theme_classic()+
  theme(legend.position="none")+
    labs(x="Bill length (mm)",
         y="Bill depth (mm)")

length_depth_scatterplot
```

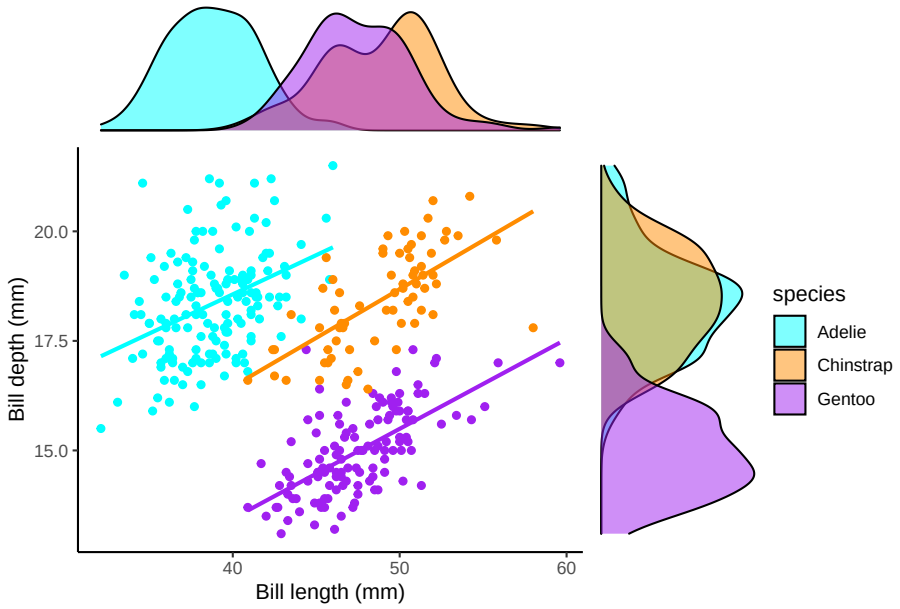



```
bill_depth_marginal_2 <- penguins %>%
  ggplot()+
  geom_density(aes(x=bill_depth_mm,
                   fill=species),
               alpha=0.5)+
  scale_fill_manual(values=colours)+
  theme_void()+
  coord_flip() # this graph needs to be rotated

bill_length_marginal_2 <- penguins %>%
  ggplot()+
  geom_density(aes(x=bill_length_mm,
                   fill=species),
               alpha=0.5)+
  scale_fill_manual(values=colours)+
  theme_void()+
  theme(legend.position="none")

layout2 <- "
AAA#
BBBC
BBBC
BBBC"
```

```
bill_length_marginal_2+length_depth_scatterplot_2+bill_depth_marginal_2+ # order of plots is impo
  plot_layout(design=layout2) # uses the layout argument defined above to arrange the size and po
```



We now clearly see a striking reversal of our previous trend, that in fact *within* each species of penguin there is an overall positive association between bill length and depth.

This should prompt us to re-evaluate our correlation metrics:

```
penguins %>%
  group_by(species) %>%
  cor_test(bill_length_mm, bill_depth_mm)
```

```
## # A tibble: 3 x 9
##   species   var1     var2       cor statistic        p conf.low conf.high method
##   <chr>     <chr>    <chr>    <dbl>     <dbl>    <dbl>    <dbl>     <dbl> <chr>
## 1 Adelie    bill_le~ bill_de~  0.39      5.19 6.67e- 7    0.247     0.519 Pears~
## 2 Chinstrap bill_le~ bill_de~  0.65      7.01 1.53e- 9    0.492     0.772 Pears~
## 3 Gentoo    bill_le~ bill_de~  0.64      9.24 1.02e-15    0.526     0.737 Pears~
```

We now see that the correlation values for all three species is >0.22 - indicating these associations are much closer than previously estimated.

11.4.2 Three or more variables

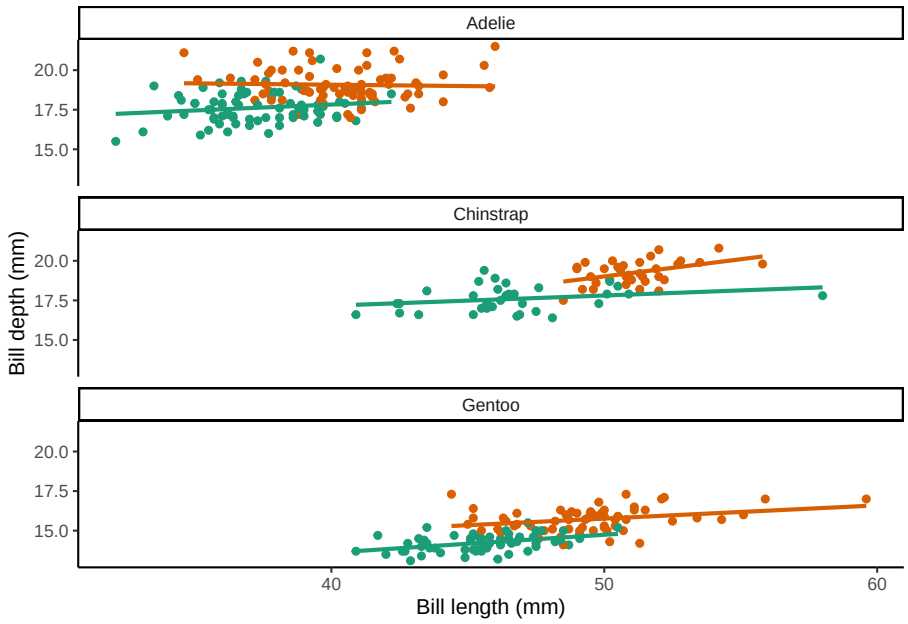
In the above example therefore, we saw the importance of exploring relationships among more than two variables at once. Broadly speaking there are two ways top do this

1. Layer an extra aesthetic mapping onto ggplot - such as size, colour, or shape

2. Use facets to construct multipanel plots according to the values of a categorical variable

If we want we can also adopt both of these approaches at the same time:

```
penguins %>%
  drop_na(sex) %>%
ggplot(aes(x= bill_length_mm,
                      y= bill_depth_mm,
                      colour=sex)) + # colour aesthetic set to sex
    geom_point()+
  geom_smooth(method="lm",
              se=FALSE)+
  scale_colour_manual(values=c("#1B9E77", "#D95F02"))+ # pick two colour scheme
  theme_classic()+
  theme(legend.position="none")+
    labs(x="Bill length (mm)",
         y="Bill depth (mm)")+
  facet_wrap(~species, ncol=1) # specify plots are stacked split by species
```



Here we can see that the trends are the same across the different penguin sexes. Although by comparing the slopes of the lines, lengths of the lines and amounts of overlap we can make insights into how "sexually dimorphic" these different species are e.g. in terms of beak morphology do some species show greater differences between males and females than others?

11.5 Summing up

This is our last data handling workshop. We have built up towards being able to discover and examine relationships and differences among variables in our data. You now have the skills to handle many different types of data, tidy it, and produce visuals to generate insight and communicate this to others.

A note of caution is required - it is very easy to spot and identify patterns.

When you do spot a trend, difference or relationship, it is important to recognise that you may not have enough evidence to assign a reason behind this observation. As scientists it is important to develope hypotheses based on knowledge and understanding, this can help (sometimes) with avoiding spurious associations.

Sometimes we may see a pattern in our data, but it has likely occurred due to random chance, rather than as a result of an underlying process. This is where formal statistical analysis, to quantitatively assess the evidence, assess probability and study effect sizes can be incredibly powerful. We will delve into these exciting topics next term.

That's it! Thank you for taking the time to get this far. Be kind to yourself if you found it difficult. You have done incredibly well.

Have some more praise!!!!

```
praise::praise()
```

```
[1] "You are spectaculaR!"
```

Chapter 12

Introduction to Statistics - Spring Week One

12.1 Introduction to statistics

Welcome back! Last term we worked on developing our data manipulation, organisation, reproducibility and visualisation skills.

This term we will be focusing more on *statistics*. We actually did quite a lot of *descriptive statistics* work last term. Every time we summarised or described our data, by calculating a **mean**, **median**, **standard deviation**, **frequency/count**, or **distribution** we were carrying out *descriptive statistics* that helped us understand our data better.

We are building on this to develop our skills in *inferential statistics*. Inferential statistics allow us to make generalisations - taking a descriptive statistics from our data such as the **sample mean**, and using it to say something about a population parameter (i.e. the **population mean**).

For example we might measure the measure the heights of some plants that have been outcrossed and inbred and make some summaries and figures to construct an average difference in height (this is **descriptive**). Or we could use this to produce some estimates of the general effect of outcrossing vs inbreeding on plant heights (this is **inferential**).

In fact that gives me an idea…

Head to Blackboard and check out today's assignment from GitHub Classrooms

12.2 Darwin's maize data

Loss of genetic diversity is an important issue in the conservation of species. Declines in population size due to over exploitation, habitat fragmentation lead to loss of genetic diversity. Even populations restored to viable numbers through conservation efforts may suffer from continued loss of population fitness because of inbreeding depression Katherine Ralls and Frankham (2020).

Charles Darwin even wrote a book on the subject *"The Effects of Cross and Self-Fertilisation in the Vegetable Kingdom"* Darwin (1876). In this he describes how he produced seeds of maize (*Zea mays*) that were fertilised with pollen from the same individual or from a different plant. The height of the seedlings that were produced from these were then measured as a proxy for their evolutionary fitness.

Darwin wanted to know whether inbreeding reduced the fitness of the selfed plants - this was his **hypothesis**. The data we are going to use today is from Darwin's original dataset.

```
library(tidyverse)

darwin <- read_csv("data/darwin.csv")
```

```
# A tibble: 6 x 5
   ...1 pot    pair type  height
  <dbl> <chr> <dbl> <chr>  <dbl>
1     1 I         1 Cross   23.5
2     2 I         1 Self    17.4
3     3 I         2 Cross   12
4     4 I         2 Self    20.4
5     5 I         3 Cross   21
6     6 I         3 Self    20
```

Before you go any further - make sure your basic R project set up is up to scratch. Do you know where the data file is saved? Have you got separate subfolders set up within your project?

You should have a script set up to put your work from today into - use this to write instructions and store comments. Use the File > New Script menu item and select an R Script.

12.2.1 Description

The first thing we should know by now is to always start by exploring our data. If you want to stretch yourself, see if you can perform a basic data check **without** prompts.

If you need a little help with remembering what to check, but would like to try and adapt scripts go to 3.5.1 from workflow 1.

Click-me for answer - If you got completely stuck, or had a go but want to check your answers then click here:

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

# check data is in a tidy format
head(darwin)

# check variable names
colnames(darwin)

# check for duplication
darwin %>%
  duplicated() %>%
  sum()

# check for typos - by looking at impossible values
darwin %>%
  summarise(min=min(height, na.rm=TRUE),
            max=max(height, na.rm=TRUE))

# check for typos by looking at distinct characters/values
darwin %>%
  distinct(pot)

darwin %>%
  distinct(pair)

darwin %>%
  distinct(type)

# missing values
darwin %>%
  is.na() %>%
  sum()

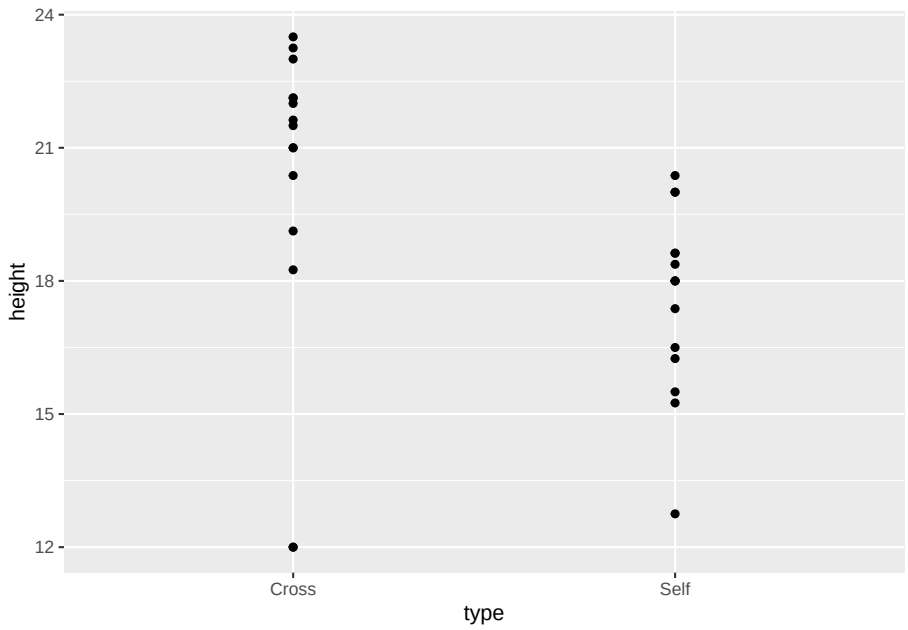
# quick summary

summary(darwin)
```

12.2.2 Visualisation

Now seems like a good time for our first data visualisation.

```
darwin %>%
  ggplot(aes(x=type,
         y=height))+
  geom_point()
```




```
# you could also substitute (or combine) other geoms including
# geom_boxplot()
# geom_violin()
# Why not have a go and see what you can make?
```

The graph clearly shows that the average height of the 'crossed' plants is greater than that of the 'selfed' plants. But we need to investigate further in order to determine whether the signal (any apparent differences in mean values) is greater than the level of noise (variance within the different groups).

The variance appears to be roughly similar between the two groups - though by making a graph we can now clearly see that in the crossed group, there is a *potential* outlier with a value of 12.

12.2.3 Comparing groups

As we have seen previously we can use various tidy functions to determine the mean and standard deviations of our groups.

```
darwin %>%
  group_by(type) %>%
  summarise(mean=mean(height),
            sd=sd(height))
```

```
# A tibble: 2 x 3
  type   mean    sd
  <chr> <dbl> <dbl>
1 Cross  20.2  3.62
2 Self   17.6  2.05
```

**Note - You should (re)familiarise yourself with how (and why) we calculate standard deviation.

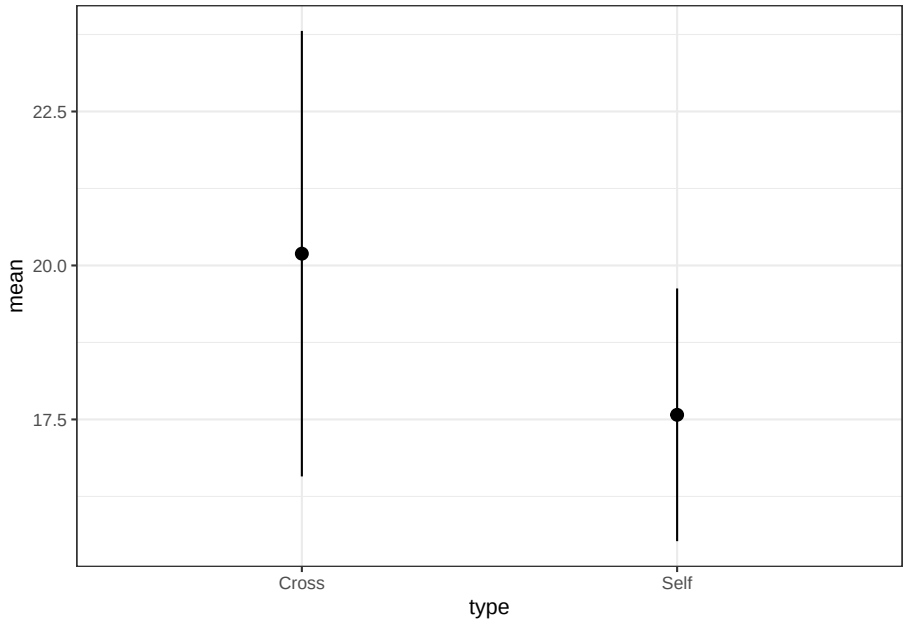
Summary statistics like these could be presented as figures or tables. We normally reserve tables for **very** simple sets of numbers, and this instance we could present both.

```
# make a new object
darwin_summary <-darwin %>%
  group_by(type) %>%
  summarise(mean=mean(height),
            sd=sd(height))

# make a summary plot
darwin_summary %>%
  ggplot(aes(x=type,
             y=mean))+
  geom_pointrange(aes(ymin=mean-sd, ymax=mean+sd))+
  theme_bw()
```

Table 12.1: Summary statistics of crossed and selfed maize plants

type	mean	sd
Cross	20.19167	3.616945
Self	17.57500	2.051676




```
# put this at top of script
library(kableExtra)

# use kable extra functions to make a nice table (could be replaced with kable() if ne
darwin_summary %>%
    kbl(caption="Summary statistics of crossed and selfed maize plants") %>%
  kable_styling(bootstrap_options = "striped", full_width = T, position = "left")
```

Descriptive statistics and careful data checking are often skipped steps in the rush to answer the **big** questions. However, description is an essential part of early phase analysis.

12.3 Estimation

In the section above we concentrated on description. But the hypothesis Darwin aimed to test was whether 'inbreeding reduced the fitness of the selfed plants'. To do this we will use the height of the plants as a proxy for fitness and explicitly

address whether there is a **difference** in the mean heights of the plants between these two groups.

Our goal is to:

- Estimate the mean heights of the plants in these two groups
- Estimate the mean *difference* in heights between these two groups
- Quantify our *confidence* in these differences

12.3.1 Differences between groups

Darwin's data used match pairs - each pair shared one parent. So that in pair 1 the same parent plant was either 'selfed'or 'crossed' to produce offspring. This is a powerful approach to experimental design, as it means that we can look at the differences in heights across each of the 15 pairs of plants - rather than having to infer differences from two randomly derived groups.

In order to calculate the differences in height between each pair we need to do some data wrangling - **can you complete the below to produce the required output?**

```
# pivot data to wide format then subtract Selfed plant heights from Crossed plant heights

darwin_wide <- darwin %>%
  pivot_wider(names_from=type, values_from=height, id_cols=pair) %>%
  mutate(difference=___________)
```

```
# A tibble: 15 x 4
    pair Cross  Self difference
   <dbl> <dbl> <dbl>      <dbl>
 1     1  23.5  17.4       6.12
 2     2  12    20.4      -8.38
 3     3  21    20         1
 4     4  22    20         2
```

We now have the difference in height for each pair, we can use this to calculate the **mean difference** in heights between paired plants, and the amount of variance (as standard deviation)

```
difference_summary <- darwin_wide %>%
  summarise(mean=mean(difference),
            sd=sd(difference),
            n=n())

difference_summary
```

```
# A tibble: 1 x 2
   mean    sd
  <dbl> <dbl>
1  2.62  4.72
```

What we have just calculated is the average difference in height between these groups of plants and the standard deviation of the difference Moving forward we will be working a lot with estimating our confidence in differences between groups

12.3.2 Standard error of the difference

Remember standard deviation is a *descriptive* statistic - it measures the variance within our dataset - e.g. how closely do datapoints fit to the mean. However for *inferential* statistics we are more interested in our confidence in our estimation of the mean. This is where standard error comes in.

We can think of error as a standard deviation of the mean. The mean we have calculated is an estimate based on one sample of data. We would expect that if we sampled another 30 plants these would have a different sample mean. Standard error describes the variability we would expect among sample means if we repeated this experiment many times. So we can think of it as a measure of the confidence we have in our estimate of a **true population mean**.

$$SE = \frac{s}{\sqrt{(n)}}$$

As sample size increases the standard error should reduce - reflecting an increasing confidence in our estimate.

We can calculate the standard error for our sample by applying this equation to our `difference_summary` object, can you complete this?

```
difference_summary %>%
  mutate(se=_______)
```

```
# A tibble: 1 x 4
   mean    sd     n    se
  <dbl> <dbl> <int> <dbl>
1  2.62  4.72    15  1.22
```

Our estimate of the mean is not really very useful without an accompanying measuring of *uncertainty* like the standard error, in fact estimates of averages or differences should **always** be accompanied by their measure of uncertainty.

Click me - With the information above, how would you present a short sentence describing the average different in height?**

... the average difference in height was 2.62 ± 1.22 inches (mean ± SE).

With a mean and standard error of the difference in heights between inbred and crossed plants - how do we work out how much confidence we have in their being a difference between our **population means**?

Standard error is a measure of uncertainty, the larger the standard error the more *noise* around our data and the more uncertainty we have. The smaller the standard error the more confidence we can have that our difference in means is *real*.

- Null hypothesis - there is no difference in the mean height of self vs crossed plants
- Alternate hypothesis - inbreeding reduces the fitness of the selfed plants, observed as selfed plants on average being smaller than crossed plants

A statistical way of thinking about our inferences is in terms of confidence around keeping or rejecting the null hypothesis. The (alternate) hypothesis is simply one that contradicts the null.

When we decide whether we have enough confidence that a difference is real (e.g. we could reject the null hypothesis), we cannot ever be 100% certain that this isn't a false positive (also known as a Type I error). More on this later

12.3.3 Normal distribution

As we should remember from last term (and before), the normal distribution is the bell-shaped curve. It is defined by two parameters:

- the mean
- the standard deviation

The mean determines where the centre/peak of the bell curve is, the standard deviation determines the width of the bell (how long the tails are).

Large standard deviations produce wide bell curves, with short peaks. Small standard deviations produce narrow bell curves with tall peaks.

The bell curve occurs frequently in nature, most circumstances where we can think of a continuous measure coming from a population e.g. human mass, penguin flipper lengths or plant heights.

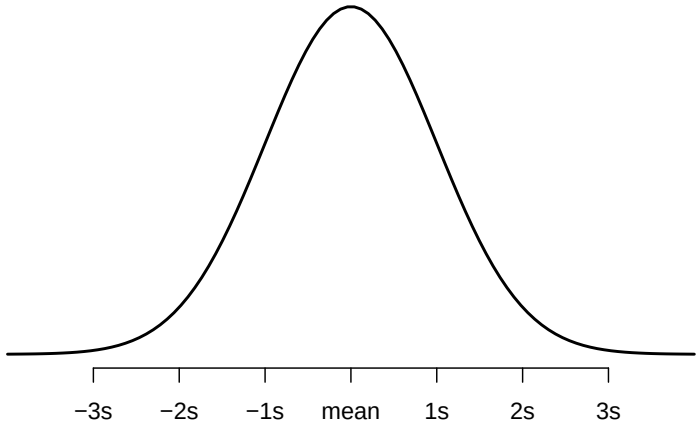
As a *probability* distribution, the area within the curve sums to the whole population (e.g. the probability that the curve contains every possible measurement is 1). Known proportions of the curve lie within certain distances from the

centre e.g. 67.8% of observations should have values within one standard deviation of the mean. 95% of observations should have values within two standard deviations of the mean. This *idealised* normal distribution is presented below:

```
#Create a sequence of 100 equally spaced numbers between -4 and 4
x <- seq(-4, 4, length=100)

#create a vector of values that shows the height of the probability distribution
#for each value in x
y <- dnorm(x)

#plot x and y as a scatterplot with connected lines (type = "l") and add
#an x-axis with custom labels
plot(x,y, type = "l", lwd = 2, axes = FALSE, xlab = "", ylab = "")
axis(1, at = -3:3, labels = c("-3s", "-2s", "-1s", "mean", "1s", "2s", "3s"))
```

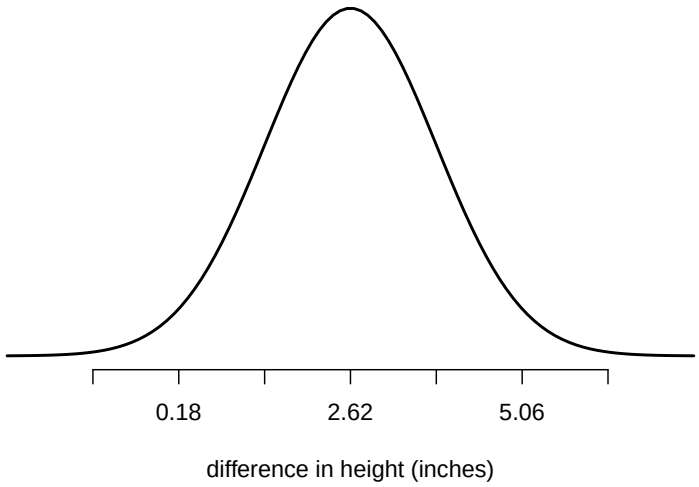


How do we convert this information into how likely we are to observe a difference of 2.62 inches in plant heights if the 'true' difference between crosses and selfed plants is *zero*?

The **central limit theorem** states that if you have a population with mean and standard deviation, and take sufficiently large random samples from the population, then the distribution of the sample means will be approximately normally distributed. Standard error then is our measure of variability around our sample mean, and we will assume that we can apply a normal distribution

to our ability to estimate the mean (*we will revisit this assumption later*).

So if we now center our bell curve on the estimate of the mean (2.62), then just over two thirds of the area under the curve is ± 1.22 inches. 95% of it will be within ± 2 standard errors, and 99.8% will be within ± 3 standard errors.



Taking a look at this figure we can ask ourselves where is zero on our normal distribution? One way to think about this is, if the true difference between our plant groups is zero, how surprising is it that we estimated a difference of 2.62 inches?

If zero was close to the center of the bell curve, then our observed mean would not be surprising at all (**if the null hypothesis is true**). However in this case it is not in the middle of the bell. It falls in the left-hand tail, and it is > than two standard deviations from our estimated mean.

We can describe this in two ways:

- We estimate that if we ran this experiment 100 times then >95 of our experiments would estimate a mean difference between our plant groups that is > 0.
- This is also usually taken as the minimum threshold needed to *reject* a null hypothesis. We can think of the probability of estimating this mean difference, if our true mean difference was zero, as $p < 0.05$.

You will probably be very used to a threshold for null hypothesis rejection of $\alpha = 0.05$, but this only the very lowest level of confidence at which we can pass a

statistical test. If we increase the severity of our test so that the minimum we require is $\alpha = 0.001$ or 99% confidence, we can see that we no longer believe we have enough confidence to reject the null hypothesis (0 is within 3 s.d. of our estimated mean).

12.3.4 Confidence Intervals

Because ± 2 standard errors covers the central 95% of the area under the normal curve, we refer to this as our 95% confidence interval. We can calculate confidence intervals for any level, but commonly we refer to standard error (68% CI), 95% and 99%.

We can work out the 95% confidence interval range of our estimated mean as follows:

```
lowerCI <- 2.62-(2*1.22)

upperCI <- 2.62+(2*1.22)

lowerCI
upperCI
```

A common mistake it to state that we are 95% confident that the 'true' mean lies within our interval. But *technically* it refers to the fact that if we kept running this experiment again and again the intervals we calculate would capture the true mean in 95% of experiments. So really we are saying that we are confident we would capture the true mean in 95% of our experiments.

How might we write this up?

The maize plants that have been cross pollinated were taller on average than the self-pollinated plants, with a mean difference in height of 2.62 [0.18, 5.06] inches (mean [95% CI]).

Note that because it is possible to generate multiple types of average and confidence interval, we clearly state them in our write up. The same would be true if you were presenting the standard error (± S.E.) or the standard deviation (± S.D.) or a median and interquartile range (median [± IQR]).

The above is a good example of a simple but clear write-up because it clearly describes the *direction* of the difference, the *amount* and the *error* in our estimate.

12.4 Summary

Statistics is all about trying to interpret whether the signal (of a difference or trend) is stronger than the amount of noise (variability). In a sample the standard deviation is a strong choice for estimating this *within* a dataset. The standard deviation of the sampling distribution of the mean is known as the standard error. The standard error (of the mean) is a measure of the precision we have in our estimate of the mean. Thanks to the central limit theorem and normal distribution we can use the variability in our estimate to calculate confidence intervals, and decide whether our signal of an effect is strong enough to reject a null hypothesis (of no effect or no difference).

Next time we will start working with linear models - this approach allows us to estimate means and intervals in a more sophisticated (and automated) way.

Chapter 13

Keeping code DRY - Spring Week One

Keep working in the same project workspace as Estimates because there's a file you need, but I would advise you start a new script to work on iteration.

So what the flip is **DRY** code anyway? It stands for **D**on't **R**epeat **Y**ourself. It's aimed at reducing the repetition and number of lines of code in your scripts.

So what the flip is **DRY** code anyway? It stands for **D**on't **R**epeat **Y**ourself. It's aimed at reducing the repetition and number of lines of code in your scripts.

Why is this useful?

- If you keep repeating the same/similar lines of code over and over then your script will become very large
- Changes or updates have to be manually applied to each and every time you have used a line of code in your script
- Mistakes can easily start to creep in

There are two broad ways we can DRY out our code

1) Use Functions - functions contain code instructions, and we can re-use them for similar processes simply by changing the arguments. In R you can use pre-built functions and start building your own!
2) Use Iteration - Apply functions to loop/repeat over different groups or lists.

13.1 Functions

Most of the time when we work in R, we will use functions; either pre-written functions or ones we write ourselves. **Functions** make it easy to use sets of code instructions repeatedly (without filling our scripts with the code underlying the function) and help us carry out multiple tasks in a single step without having to go through the details of how the steps are executed.

13.1.1 Common functions

You have been using functions in R from week one, we type the name of the function followed by parentheses e.g. `read_csv()`. Within the parentheses is where we will specify the function input and options, separated by commas `,`. One function you will use a lot is the **combine function** `c()`, which as the name suggests lets you concatenate different types of values into a **vector**.

If you are struggling to remember what a vector is, you might want to go back to our intro to R pages for a brief refresher on data types.

- Complete the script below to combine any set of numerical values using `c()`.
- Then use the `sum()` function to add them together

```
my_combined_values <- c(__,__)
sum(my_combined_values)
```

13.1.2 Write your own function

R makes it easy to create user defined functions by using `function()`. Here is how it works:

- Give your function an object name and assign the function to it, e.g. `my_function_name <- function()`.
- Within the parentheses you specify inputs and options just like how pre-written functions work, e.g. `function(input_data)`.
- Next, put all the code you want your function to execute inside curly brackets like this: `function(input_data){code to  run}`
- Use `return()` to specify what you want to your function to output once it is done running the code.

Use the following instructions to complete the function in the window below:

- I've started writing a function for you that will sum up values and take the square root of the sum. To take the square root, we use the `sqrt()` function.
- Complete this function by filling in input_data for the `sum()`, and then filling in the remaining empty parentheses with the appropriate object names.

  *Now create an object containing a set of numerical values and call it `my_combined_values`. Then try out your new function on this object, which will compute the square root of the object's sum.

```
# Use the instructions above to complete the function below
my_function_name <- function(input_data){
  s <- sum( )
  ss <- sqrt( )
  return( )
}

# Create a new object and try out your new function
my_combined_values <- c(,)

my_function_name(my_combined_values)
```

Solution

```
# Use the instructions above to complete the function below
my_function_name <- function(input_data){
  s <- sum(input_data)
  ss <- sqrt(s)
  return(ss)
}

# Create a new object and try out your new function
my_combined_values <- c(__,__)

my_function_name(my_combined_values)
```

A general rule of thumb. If you end up repeating a line of code more than three times in a script - you should write a function to do the work instead. And write clear comments on its use! Why?

It reduces the numbers of lines of code in your script, it reduces the amount of repetition in the code, if you need to make changes you can change the function without having to hunt through all of your code.

A really good way to organise your functions is to organise them into a separate

script to the rest of your analysis. Write functions in a separate script and use source("scripts/functions.R")

Figure 13.1: An example of a typical R project set-up

It might surprise you to know that there is no prebuilt function for standard error in base R, but we can build our own!

```
se <- function(x) {
  se <- sd(x)/(sqrt(length(x)))
  return(se)
  }
```

```
# The seq R function generates a sequence of numeric values
x <- seq(8, 20, length=100)
```

```
# use the mean function to calculate average for x
mean(x)

# use our custom function to calculate se for x
se(x)

# could be used in the group_by and summarise pipes we are used to using
darwin_wide %>%
  summarise(mean=mean(difference),
            sd=sd(difference),
            n=n(),
            se=se(difference))
```

Functions can basically be given any name you want. However, you may accidentally overwrite existing functions if you give them the same name, so try and make them unique.

13.1.3 More functions

This is an example of a very simple function that just prints the string "Hello World" whenever you type the function `say_hello()`

```
say_hello <- function(){
  paste("Hello World")
}

say_hello()

# Notice the argument section is blank, what happens when you try to put something in the bracket

# say_hello("x")
```

Now lets try a similar function, but we include an argument:

```
say_morning <- function(x){
  paste("Good morning", x)
}

# what happens when you try to run this function
say_morning()
```

```
#  what about this one?
say_morning("Phil")
```

So that was an example where we included an argument for our function. However, you are probably used to the idea that many functions have "default" values for arguments, and we can easily set these.

```
say_morning_default <- function(name="you"){
  paste("Good morning", name)
}
# what happens when you put a "string" in the brackets? What happens when you leave it
say_morning_default()
```

13.1.4 Exercise

We are going to try and write a custom function called `find_largest_male()`, it will be used to identify the largest male *Drosophila* from a small dataset.

```
# Make some fake data into a tibble

vial <- 1:10
sex <- (rep(c("male","female"),5))
weight_mg <- rnorm(10, mean=0.2, sd=0.01)

dros_weight <- tibble(vial, sex, weight_mg)
```

13.1.4.1 Step 1.

What functions would you use to extract the heaviest male from this dataset? Try and think that through first, then see if you can rework it into a function.

Click-me for A solution

```
find_largest_male <- function(df){
  dros_male <- filter(df, df[,2]=="male")
  head(arrange(dros_male, desc(dros_male[,3])),n=1)
}

# Note that this uses dplyr functions, so this would need to be loaded for this to wor
```

13.1.4.2 Step 2.

Now we have a basic function we can work to refine and extend it. What if we made a function that could pick the largest Drosophila overall, or male or female depending on what we need?

Click-me for A solution

```
find_largest_fly <- function(df, n=1, s=c("male", "female") ){
  drossex <- filter(df, sex %in% s )
  head(arrange(drossex, desc(drossex[,3])),n=n)
}
```

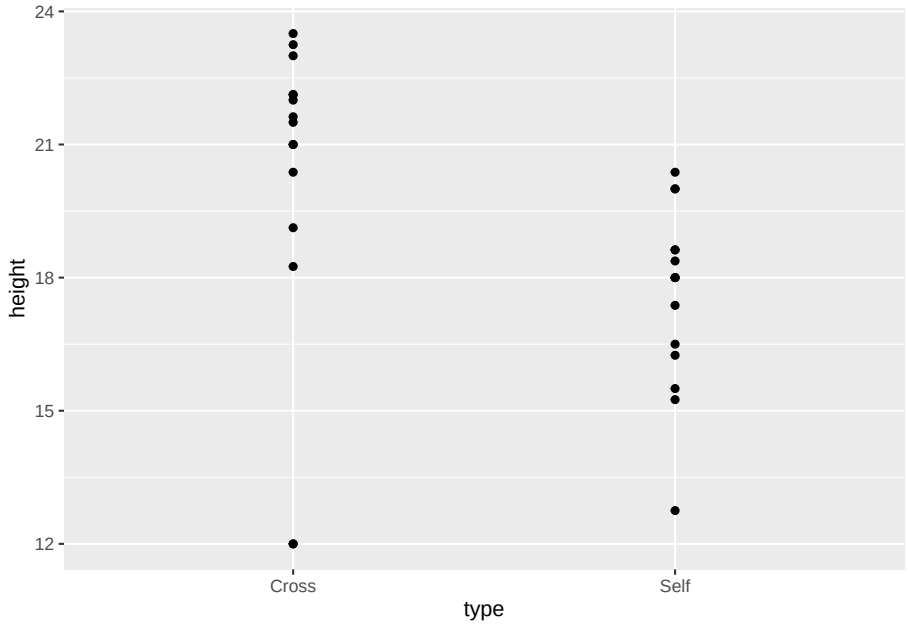
13.1.5 Custom ggplot themes

It is often the case that we start to default to a particular 'style' for our figures, or you may be making several similar figures within a research paper. Creating custom functions can extend to making our own custom ggplot themes. You have probably already used theme variants such as `theme_bw()`, `theme_void()`, `theme_minimal()` - these are incredibly useful, but you might find you still wish to make consistent changes.

Here is a plot we made in the previous chapter

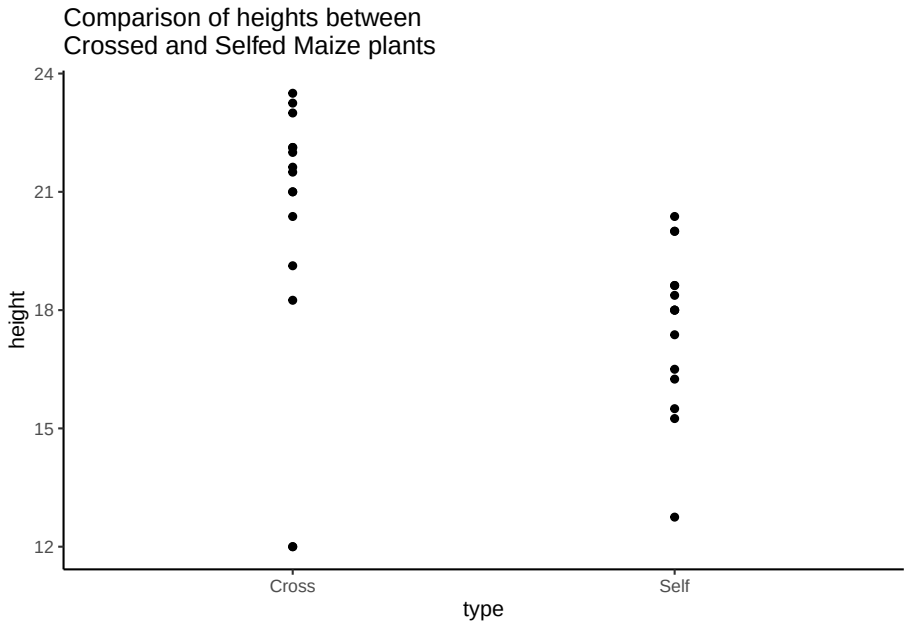
```
plot <- darwin %>%
  ggplot(aes(x=type,
         y=height))+
  geom_point()

plot
```



With the addition of a title and `theme_classic()` we can improve the style quickly

```
plot+
  ggtitle("Comparison of heights between \nCrossed and Selfed Maize plants")+
  theme_classic()
```



But I **still** want to make some more changes, rather than do this work for one figure, and potentially have to repeat this several times for subsequent figures, I can decide to make a new function instead.

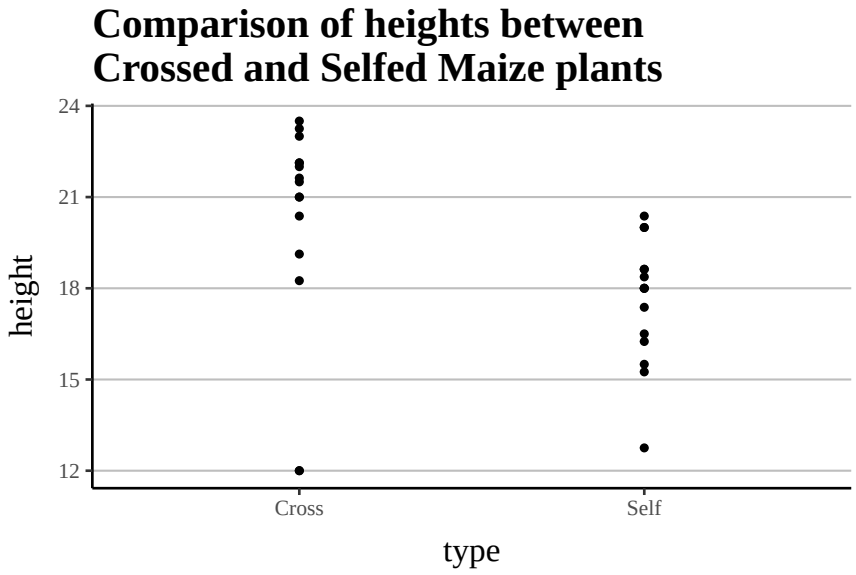
```
# custom theme sets defaults for font and size, but these can be changed without changing the fu
theme_custom <- function(base_size=14, base_family="serif"){
  theme_classic(base_size = base_size,
                base_family = base_family,
                ) +
# update theme minimal
theme(
  # specify default settings for plot titles - use rel to set titles relative to base size
  plot.title=element_text(size=rel(1.5),
      face="bold",
      family=base_family),
  #specify defaults for axis titles
  axis.title=element_text(
    size=rel(1.2),
    family=base_family),
  # specify position for y axis title
  axis.title.y=element_text(margin = margin(r = 10, l= 10)),
  # specify position for x axis title
  axis.title.x = element_text(margin = margin( t = 10, b = 10)),
  # set major y grid lines
```

```
  panel.grid.major.y = element_line(colour="gray", size=0.5),
  # add axis lines
  axis.line=element_line(),
   # Adding a 0.5cm margin around the plot
  plot.margin = unit(c(0.5, 0.5, 0.5, 0.5), units = , "cm"),
   # Setting the font for the legend text
  legend.text = element_text(face = "italic"),
    # Removing the legend title
          legend.title = element_blank(),
   # Setting the position for the legend - 0 is left/bottom, 1 is top/right
          legend.position = c(0.9, 0.8)
)

}
```

With this function set, I can now use it for as many figures as I wish. To use it in the future I should probably save it in a unique script, with a clear title and comments for future use.

```
plot+
ggtitle("Comparison of heights between \nCrossed and Selfed Maize plants")+
theme_custom()
```



13.1.6 Writing Packages

You should be familiar by now with the idea that R packages add data and functions to your workspace. They are bundles of code that anyone can write, most commonly you will be downloading packages from CRAN. However development stage packages can also be downloaded directly from GitHub.

In the previous section we walked through some very basic writing of new functions. We can save these as R scripts and move them from project to project. Or we could choose to write them into a documented R package. This tutorial shows you just how to do that. Writing packages is a useful thing to do, even if you think the only person that will ever use them is yourself, as you then more easily access these functions across different workspaces and projects.

13.2 Iteration

We've seen how to write a function and how they can be used to create concise reusable operations that can be applied multiple times in a script without having to copy and paste, but where functions really come into their own is when combined with iteration. Iteration is the process of running the same operation on a group of objects, further minimising code replication.

13.2.1 iteration in tidyverse

You have already used functions within tidyverse that allow you to perform iteration. For example we have the used the function `group_by()` many times. It allows us to subset our data, then command R to perform subsequent `dplyr` functions on each group separately. Sometimes however, you might want to use iteration more explicitly.

13.2.2 For Loops

A for loop has three core parts:

1) The sequence of items to iterate through

2) The operations to conduct per item in the sequence

3) The container for the results (optional)

The basic syntax is: for (item in sequence) {do operations using item}. Note the parentheses and the curly brackets. The results could be printed to console, or stored in a container R object.

```
for(i in list){
    # PERFORM SOME ACTION
}
```

A simple for loop **example** is below. For every number in the vector add 2. There is no *container object* here, the results of the function are printed directly into the console.

```
for (num in c(1,2,3,4,5)) {  # the SEQUENCE is defined (numbers 1 to 5) and loop is op
  print(num + 2)             # The OPERATIONS (add two to each sequence number and pri
}                            # The loop is closed with "}"
```

```
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
```

So let's make a slightly more complicated function - first we are making a new tibble, first we have four vectors - made of 10 numbers each randomly generated to be roughly close to a 0 mean with a s.d. of 1. Then we combine them to make a tibble

```
# a simple tibble
df <- tibble(
  a =  rnorm(10),
  b =  rnorm(10),
  c =  rnorm(10),
  d = rnorm(10)
)
```

Each vector randomly generated so the actual averages will be slightly different

```
median(df$a)
# [1] 0.3085154
median(df$b)
# [1] 0.5429483
median(df$c)
# [1] -0.5137691
median(df$d)
# [1] 0.04415608
```

So the above code works, but it is repetitive, applying the same function again and again. Below we have a simple for loop

```
output <- vector("double", ncol(df))  # 1. output having a predefined empty list of th

for (i in seq_along(df)) {            # 2. sequence - determines what to loop over - h
```

```
  output[[i]] <- median(df[[i]])      # 3. body - each time the loop runs it allocates a value to
}
output
# [1]  0.30851540  0.54294832 -0.51376911  0.0441560
```

13.2.3 Exercise for For Loops

This part of the exercise is a real world example of using simple `for()` loops to create graphs. This data comes from the Living Planet Index, which holds data on various vertebrate species collected from 1974 to 2014.

First we should import the data:

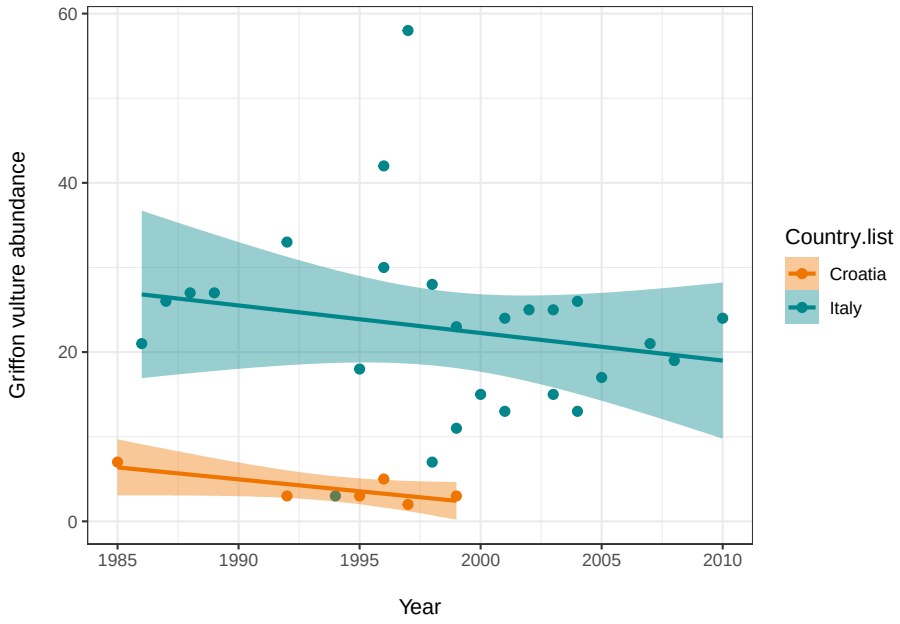
```
LPI <- read_csv("data/LPI_data_loops.csv")
```

Here is an example of a graph we can make from the data looking at Griffon Vulture changes in numbers over time.

```
vultureITCR <- LPI %>%
  filter(Common.Name == "Griffon vulture / Eurasian griffon") %>%
  filter(Country.list==c("Croatia", "Italy"))

vulture_scatter <- ggplot(vultureITCR, aes(x = year, y = abundance, colour = Country.list)) +
    geom_point(size = 2) +
  # Changing point size
    geom_smooth(method = lm, aes(fill = Country.list)) +
  # Adding a linear model fit and colour-coding by country
    scale_fill_manual(values = c("#EE7600", "#00868B")) +
  # Adding custom colours
    scale_colour_manual(values = c("#EE7600", "#00868B"),
                        # Adding custom colours
                        labels = c("Croatia", "Italy")) +
  # Adding labels for the legend
    ylab("Griffon vulture abundance\n") +
    xlab("\nYear")  +
        theme_bw()

vulture_scatter
```



We can use custom themes (like the one you made earlier) to quickly update figures

```
vulture_scatter+theme_custom()
```

Now let's take a look at using functions and loops to help us build figures.

```
LPI_UK <- filter(LPI, Country.list == "United Kingdom")

# Pick 4 species and make scatterplots with a simple regression model fits that show h

# Careful with the spelling of the names, it needs to match the names of the species i

house_sparrow <- filter(LPI_UK, Common.Name == "House sparrow")
great_tit <- filter(LPI_UK, Common.Name == "Great tit")
corn_bunting <- filter(LPI_UK, Common.Name == "Corn bunting")
meadow_pipit <- filter(LPI_UK, Common.Name == "Meadow pipit")
```

So now we have four separate R objects holding data from four bird species, our standard approach might then be to make four figures looking at abundance over time.

```
house_sparrow_scatter <- ggplot(house_sparrow, aes (x = year, y = abundance)) +
    geom_point(size = 2, colour = "#00868B") +
    geom_smooth(method = lm, colour = "#00868B", fill = "#00868B") +
    theme_custom() +
    labs(y = "Abundance\n", x = "", title = "House sparrow")

great_tit_scatter <- ggplot(great_tit, aes (x = year, y = abundance)) +
    geom_point(size = 2, colour = "#00868B") +
    geom_smooth(method = lm, colour = "#00868B", fill = "#00868B") +
    theme_custom() +
    labs(y = "Abundance\n", x = "", title = "Great tit")

corn_bunting_scatter <- ggplot(corn_bunting, aes (x = year, y = abundance)) +
    geom_point(size = 2, colour = "#00868B") +
    geom_smooth(method = lm, colour = "#00868B", fill = "#00868B") +
    theme_custom() +
    labs(y = "Abundance\n", x = "", title = "Corn bunting")

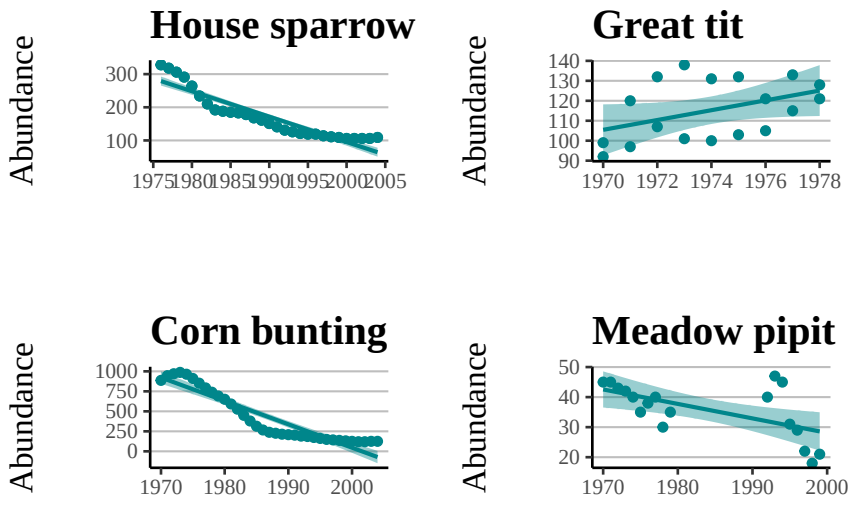
meadow_pipit_scatter <- ggplot(meadow_pipit, aes (x = year, y = abundance)) +
    geom_point(size = 2, colour = "#00868B") +
    geom_smooth(method = lm, colour = "#00868B", fill = "#00868B") +
    theme_custom() +
    labs(y = "Abundance\n", x = "", title = "Meadow pipit")
```

If we want to look at all four plots at once we can use the layout functions from the package `patchwork`.

```
# put at the top of your script
library(patchwork)

layout <- "AABB
           CCDD"

house_sparrow_scatter+
  great_tit_scatter+
  corn_bunting_scatter+
  meadow_pipit_scatter+
  plot_layout(design=layout)
```



This is ok, but arguably still requires a lot of code repetition. We have used the same lines of code four times to recreate four plots that are functionally the same. If we want to make any changes to the look of our plots we have to make four separate edits & mistakes can easily creep in.

If we want to apply a loop, then the easiest thing is to first make our objects into an R list:

```
Sp_list <- list(house_sparrow, great_tit, corn_bunting, meadow_pipit)
```

Then loop down the length of our list:

```
my_plots <- list(length(Sp_list))

for (i in 1:length(Sp_list)) {
  # For every item along the length of Sp_list we want R to perform the following func
  data <- as.data.frame(Sp_list[i])
  # Create a dataframe for each species
  sp.name <- unique(data$Common.Name)
  # Create an object that holds the species name, so that we can title each graph
  plot <- ggplot(data, aes (x = year, y = abundance)) +
    # Make the plots and add our customised theme
    geom_point(size = 2, colour = "#00868B") +
    geom_smooth(method = lm, colour = "#00868B", fill = "#00868B") +
```

```
    theme_custom() +
    labs(y = "Abundance\n", x = "", title = sp.name)

   # makes a list of all the plots generates
  my_plots[[i]] <- plot

  # prints each plot out as it is made
  print(plot)
}
```

So now we have a new object `my_plots` which is a list containing the four plots. This loop allowed us to code the details of our figures once, then iterate across four different groups.

```
wrap_plots(my_plots)+
  plot_layout(design=layout)
#wrap_plots function from patchwork can take a list of ggplots
```

What if you want to write a loop to save all four plots at once - can you modify the script to do this?

```
for (i in 1:length(Sp_list)) {
  # For every item along the length of Sp_list we want R to perform the following functions
  data <- as.data.frame(Sp_list[i])
  # Create a dataframe for each species
  sp.name <- unique(data$Common.Name)
  # Create an object that holds the species name, so that we can title each graph
  plot <- ggplot(data, aes (x = year, y = abundance)) +
    # Make the plots and add our customised theme
    geom_point(size = 2, colour = "#00868B") +
    geom_smooth(method = lm, colour = "#00868B", fill = "#00868B") +
    theme_custom() +
    labs(y = "Abundance\n", x = "", title = sp.name)

    if(i %% 1==0){    # The %% operator is the remainder, this handy if line prints a number ever
    print(i)
    }
# use paste to automatically add filename
  ggsave(plot, file=paste("figure/", sp.name, ".png", sep=''), dpi=900)
}
```

13.2.4 Learning to purr

Another approach to iterative operations is the purrr package - it is the tidyverse approach to iteration.

If you are faced with performing the same task several times, it is probably worth creating a generalised solution that you can use across many inputs. For example, producing plots for multiple jurisdictions, or importing and combining many files.

There are also a few other advantages to purrr - you can use it with pipes %>%, it handles errors better than normal for loops, and the syntax is quite clean and simple! If you are using a for loop, you can probably do it more clearly and succinctly with purrr!

Remember purrr is a very tidyverse focused method of iteration, so understanding for loops can be useful if you end up learning other programming languages in the future.

purrr is more focused on the function, and less obviously focused on the looping aspect, as you might be able to see below.

This is a really good intro to learning how to purrr - try some of the simple exercises here before trying the below

Now you have a simple grasp of `purrr` you should see that we have a list containing species objects, and an anonymous function ~ggplot on the right. We can use this to quickly make four plots, just like we did with our for loops.

```
my_plots_2 <- map(Sp_list, ~ggplot(data, aes (x = year, y = abundance)) +
                      geom_point(size = 2, colour = "#00868B") +
                      geom_smooth(method = lm, colour = "#00868B", fill = "#00868B") +
                      theme_custom() +
                      labs(y = "Abundance\n", x = "", title = sp.name))


wrap_plots(my_plots_2)+plot_layout(design=layout)
```

13.3 More Practice

If you want to play with basic coding more, try some of these other tutorials, if you try them, let me know how you get on!

Importing Hospital Records

Bioinformatics and Lists

13.4 Summary

Making functions and iterations are advanced R skills, and can often seem daunting. I do not expect you to HAVE to implement these for this course, but I do want to give you an insight into some core programming skills that you might be interested in and want to develop.

Below are some links you may find useful

- RStudio education cheat sheet for ourr
- R4DS - intro to programming
- Coding club
- EpiR handbook

*R Cheat Sheets

*R Cheat Sheets

Bibliography

Bryan, J. (2012). *Happy Git and GitHub for the useR*. Chapman and Hall/CRC, Boca Raton, Florida.

Csárdi, G. (2020). *gitcreds: Query git Credentials from R*. R package version 0.1.1.

Darwin, C. (1876). *The Effects of Cross and Self Fertilisation in the Vegetable Kingdom*. Public Domain.

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, D. N. and Pedersen, T. L. (2020). *ggplot2: Elegant Graphics for Data Analysis*. Chapman and Hall/CRC, Boca Raton, Florida.

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.

Kassambara, A. (2020). *rstatix: Pipe-Friendly Framework for Basic Statistical Tests*. R package version 0.6.0.

Katherine Ralls, Paul Sunnucks, R. L. and Frankham, R. (2020). Genetic rescue: A critique of the evidence supports maximizing genetic diversity rather than minimizing the introduction of putatively harmful genetic variation. *Biological Conservation*, 251:108784.

Müller, K. (2020). *here: A Simpler Way to Find Your Files*. R package version 1.0.1.

Ou, J. (2021). *colorBlindness: Safe Color Set for Color Blindness*. R package version 0.1.9.

Pedersen, T. L. (2020). *patchwork: The Composer of Plots*. R package version 1.1.1.

Ram, K. and Wickham, H. (2018). *wesanderson: A Wes Anderson Palette Generator*. R package version 0.3.6.

Sievert, C., Parmer, C., Hocking, T., Chamberlain, S., Ram, K., Corvellec, M., and Despouy, P. (2021). *plotly: Create Interactive Web Graphics via plotly.js*. R package version 4.9.3.

Spinu, V., Grolemund, G., and Wickham, H. (2021). *lubridate: Make Dealing with Dates a Little Easier*. R package version 1.7.10.

Wickham, H. (2019). *stringr: Simple, Consistent Wrappers for Common String Operations*. R package version 1.4.0.

Wickham, H. (2021). *tidyr: Tidy Messy Data*. R package version 1.1.3.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, 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., Bryan, J., and Barrett, M. (2021a). *usethis: Automate Package and Project Setup*. R package version 2.1.3.

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., Yutani, H., and Dunnington, D. (2021b). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.3.5.

Wickham, H., François, R., Henry, L., and Müller, K. (2021c). *dplyr: A Grammar of Data Manipulation*. R package version 1.0.6.

Wickham, H., Hester, J., and Bryan, J. (2021d). *readr: Read Rectangular Text Data*. R package version 2.1.1.

Wickham, H. and Seidel, D. (2020). *scales: Scale Functions for Visualization*. R package version 1.1.1.

Wilke, C. O. (2020). *Fundamentals of Data Visualization*. Chapman and Hall/CRC, Boca Raton, Florida.

Wilke, C. O. (2021). *ggridges: Ridgeline Plots in ggplot2*. R package version 0.5.3.

Xiao, N. (2018). *ggsci: Scientific Journal and Sci-Fi Themed Color Palettes for ggplot2*. R package version 2.9.

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Zhu, H. (2020). *kableExtra: Construct Complex Table with kable and Pipe Syntax*. R package version 1.3.1.