PGR Coding Workshop

An introduction to RStudio

What is RStudio?

RStudio is an 'Integrated Development Environment' for R, a software application that allows the user to perform statistical analysis, plot graphs, write functions and manage datasets using the R statistical programming language.

How to install RStudio?

Before you can install RStudio, you need to install R, the programming language RStudio uses. You can download R from the official website: https://cloud.r-project.org

For Windows users, select '**Download R for Windows**', followed by '**base**' then click on the download link.

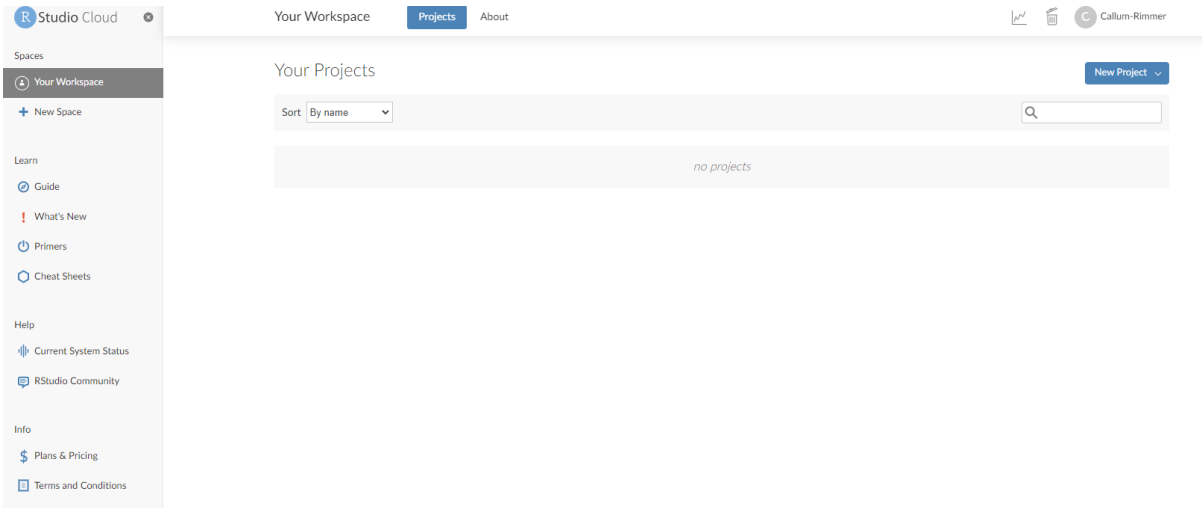
For macOS users, select '**Download R for macOS**', then click on the appropriate download link. At the time of writing, for M1 mac users it's '**R-4.1.2-arm64.pkg**' while for intel mac users it's '**R-4.1.2.pkg**'.

Now you can download RStudio: https://www.rstudio.com/products/rstudio/download/

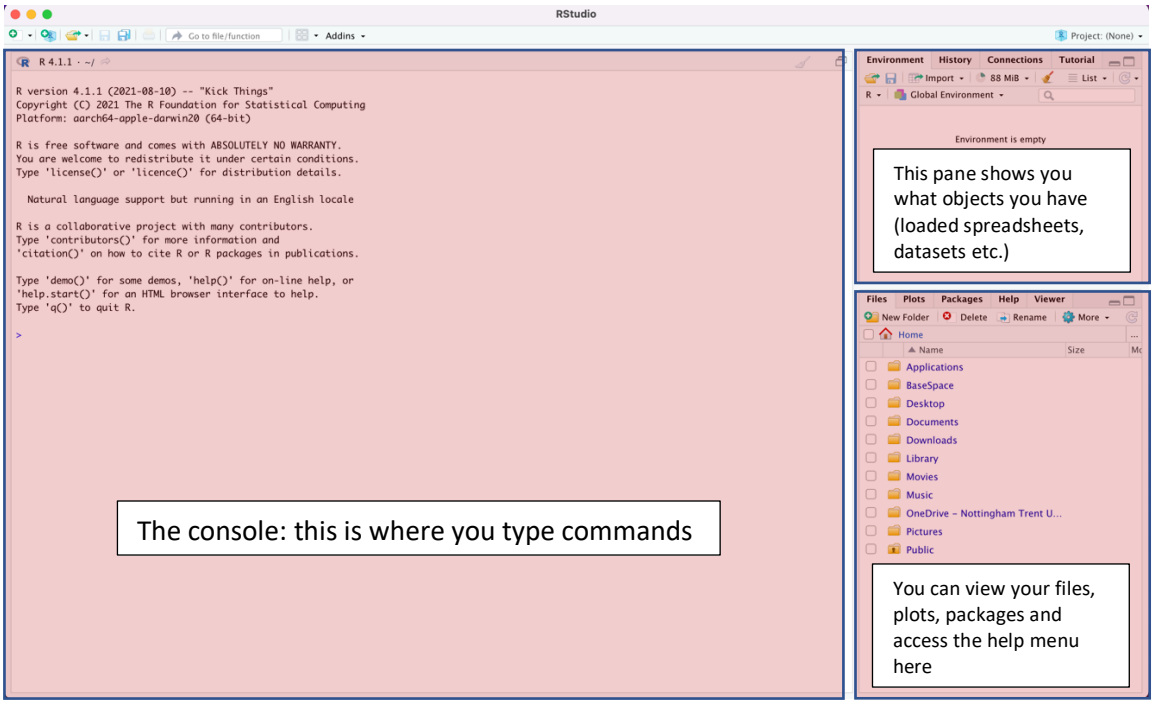
We recommend you install R and RStudio on your own computer, however there may be times where you don't have access to a local version of RStudio and instead need to use the cloud based version. This might be if an on-campus computer doesn't have RStudio installed.

To access RStudio Cloud, follow this link: https://rstudio.cloud/

RStudio Cloud is free to use, but you are limited to 25 hours per month. Once you have set up an account (or you can log in through a google or GitHub account), create a new project and it should load up RStudio the same as the local version.



The RStudio interface

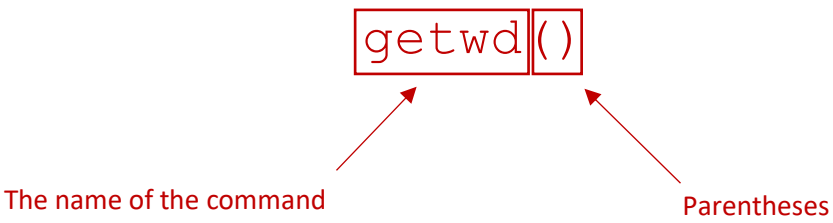


The basics

Functions

Unlike excel, SPSS or other similar software which operate using a 'point and click' system, R is an interpreted language. In order to do anything in RStudio, you need to tell it what to do using the R language, typically through functions.

A function in R usually consists of two parts…

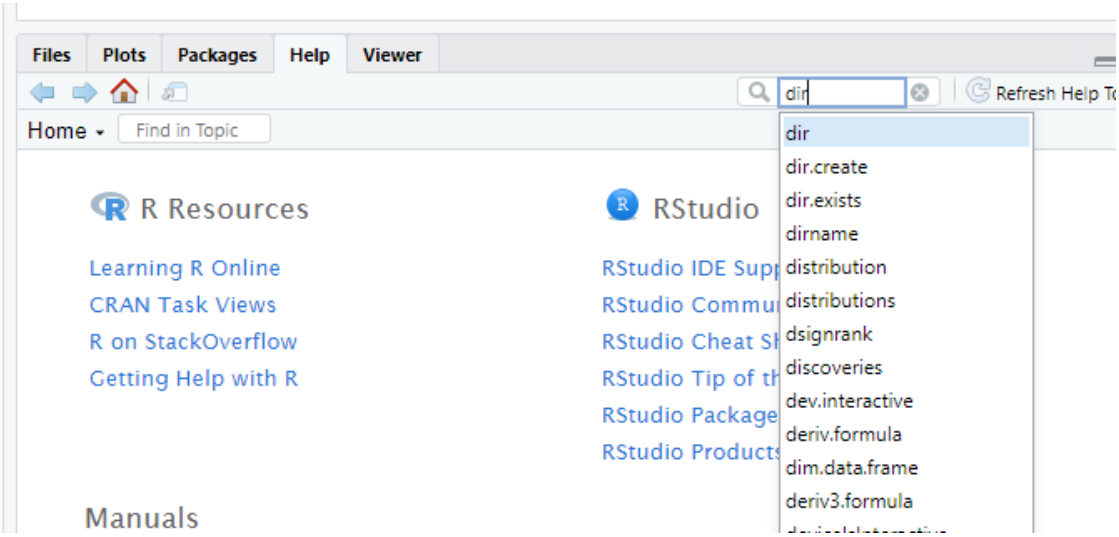


The function `getwd()` tells you the current folder location RStudio is looking at. You can see the brackets are empty, there is nothing inside them. This is because you don't need to supply any additional information, also called 'arguments', to the function. Most functions however do require one or more arguments.

Another function you can try is `dir()` – this lists all the files in the current folder location. This time however, lets try supplying an argument to this function. If you run `dir(recursive = TRUE)` not only will the function tell you what files you have in your current folder location, but all the sub-

folders within that as well. This might have filled your console with a list of files, so this is a good time to learn how to clear your console. Press CTRL + L on your keyboard, and this should wipe your console.

If you want to learn more about a function, you can either run the help command `?(name_of_function)`, search for it on the internet or use the help menu within RStudio. Select the 'Help' tab in the bottom right pane, then type dir.



Data types

Vectors:

A series of values. These are created using the `c()` function, where `c()` stands for "combine" or 'concatenate'. For example, `c(6, 11, 13, 31, 90, 92)` creates a six-element series of positive integer values .

Factors:

Categorical data are commonly represented in R as factors. Categorical data can also be represented as strings.

Data frames:

Rectangular spreadsheets. They are representations of datasets in R where the rows correspond to observations and the columns correspond to variables.

Objects

What if you need the output from one function in order to perform something else? R doesn't save the output from anything you run in the console unless you assign it to an object.

If we go back to the vector above, if you type it in the console it just writes it for you but doesn't save it. We can instead assign this vector to an object, using either an `=` or `<-`.

`My_vector <- c(6, 11, 13, 31, 90, 92)` You should now see this in your environment pane in the top right corner.

You can now just type `My_vector` and RStudio will print out your vector, or you could use it in another function.

Packages

Alongside all the functions that come with R built-in, you can also install packages containing lots of other functions. The main package we will be working with later is called 'Tidyverse', and contains functions relating to graph plotting, managing datasets and much more. Tidyverse is actually just a collection of multiple packages all collated together, you can find out more on their official website: https://www.tidyverse.org/

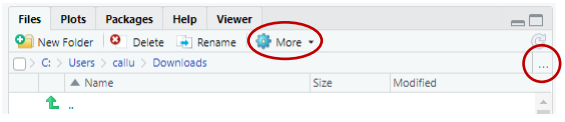
To install this package, type `install.packages('tidyverse')`.

Another package we will be using today is called 'palmerpenguins'. This is a dataset designed to be used as a learning tool for data exploration and visualisation. It contains information about penguin species, their geographical distribution and their bill dimensions.

To install this package, type `install.packages('palmerpenguins')`

Setting your working directory and importing files

Your working directory (the current folder location RStudio is set to work in) is important in RStudio. This is the location you can import files from, and where RStudio will place any files you export. To see what the current working directory is set to type `getwd()`. To set your working directory, you can use the function `setwd('C:/insert/folder/path/here')`. There is an easier way to do this however:



Click on the three small dots then browse to the desired folder location and select it. Then select the 'More' option and click 'Set as working directory'.

The function and arguments you use to import a file and assign it to an object depend on the type of file it is e.g. .tsv or .csv. One option using base R (no packages needed) is the `read.delim()` function. You need to assign the function to an object and include whether the file has column headers and the type of separator the file uses (comma `','` for .csv files and tab `'\t'` for .tsv files).

```
My_tsv_file <- read.delim('name_of_file.tsv', header = TRUE, sep = '\t')
My_csv_file <- read.delim('name_of_file.csv', header = TRUE, sep = ',')
```

Exercise

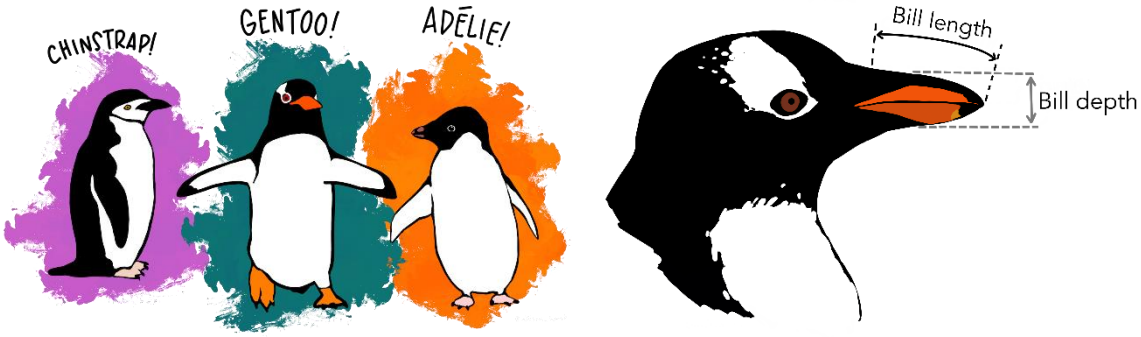
Learning to plot with ggplot2

Loading libraries and preparing our dataset

Earlier we installed both packages we need, Tidyverse and palmerpenguins. RStudio cannot yet access these until we load them (referred to as libraries). Think of it like installing a program on your computer, you can't use it until you load it up despite it being installed. To load a library you use the library() function.

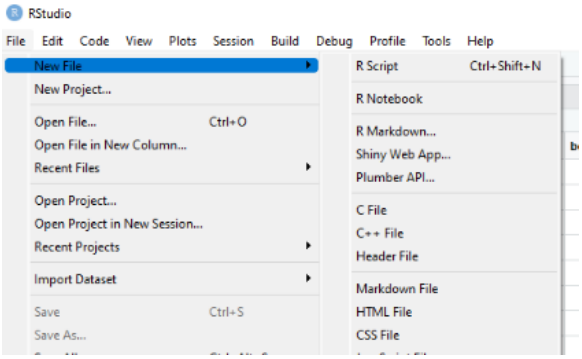
Type in both `library(tidyverse)` and `library(palmerpenguins)`.

We can now use all the functions from the Tidyverse package and we have our dataset loaded and ready to use. It's useful though if we could actually view our dataset, so you can see what you are plotting. There are many ways to do this. One option is to use the `head()` function. Type `head(penguins)` and see what you get. This just views some of the data in your console, but you can view the entire dataset in its own pane by typing `data(penguins)`. This should add a 'penguins' entry under values in your environment pane. Then type `penguins` in the console and click on its data entry in the top right to view the table. As you can see this dataset contains information about three species of penguin, metrics associated with their bills, their sex and island location.



Creating an R script file

The best way to keep track of all the code you are going to use is to note it down in an R script file. By doing this you can keep all the relevant code in the same place, and you can even run it directly from the script file as opposed to continuously typing it into the console. To create a new R script file click on File → New File → R Script. If you want include non-code text start the line with a #.



How does plotting with ggplot2 work?

The function we are going to use is called `ggplot()` which is from the ggplot2 package within Tidyverse.

With ggplot2, you begin a plot with the function `ggplot()`. `ggplot()` creates a coordinate system that you can add layers to. The first argument of `ggplot()` is the dataset to use in the graph. So `ggplot(data = penguins)` creates an empty graph, but it's not very interesting to look at.

You complete your graph by adding one or more layers to `ggplot()`. The function `geom_point()` adds a layer of points to your plot, which creates a scatterplot. ggplot2 comes with many geom functions that each add a different type of layer to a plot.

Each geom function in ggplot2 takes a `mapping` argument. This defines how variables in your dataset are mapped to visual properties. The `mapping` argument is always paired with `aes()`, and the `x` and `y` arguments of `aes()` specify which variables to map to the x and y axes. ggplot2 looks for the mapped variables in the `data` argument, in this case penguins. The basic template for a graph with ggplot2 looks like this:

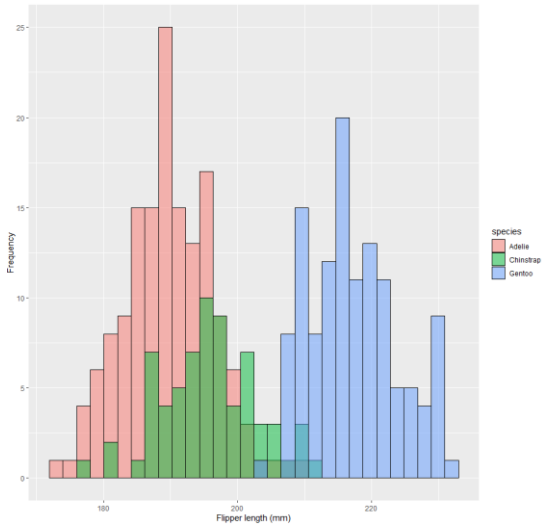
```
ggplot(data = your_dataset, aes(x = variable, y = variable)) + geom_function()
```

Plotting one variable

The first thing we are going to do is plot a single variable from our penguin dataset, flipper length. We are first going to plot a histogram, from the function `geom_histogram()`.

```
flipper_histogram <- ggplot(data = penguins, aes(x = flipper_length_mm)) + geom_histogram(aes(fill = species), alpha = 0.5, colour = 'black', position = 'identity) + labs(x = 'Flipper length (mm)', y = 'Frequency')
```

You can see that a few extra functions and arguments were added in addition to the template. The `fill = species` argument within the `geom_histogram` function colours the bars by species. `Alpha` makes overlapping bars transparent. The `labs()` function allows you to add your own labels for axis, legends and titles. You can search for the different geom functions in the help menu or on the internet, where there is much more information available on how to customise these plots exactly how you want.

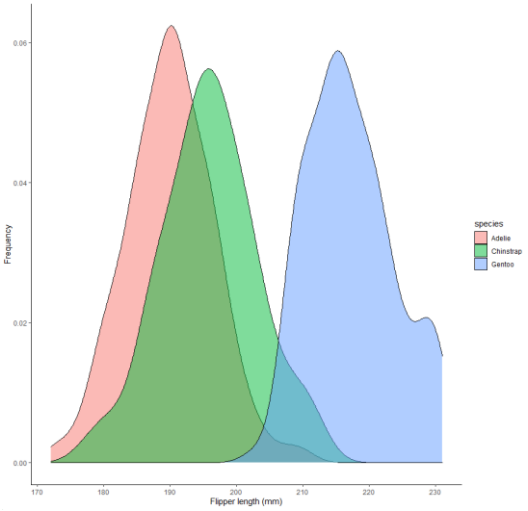


Because we assigned this plot to the object `flipper_histogram` RStudio won't actually show you the graph. Type `flipper_histogram` into the console and the plot will appear in the pane in the bottom right.

Your plot should look like this.

Let's have a go at making a nicer looking plot using the same data.

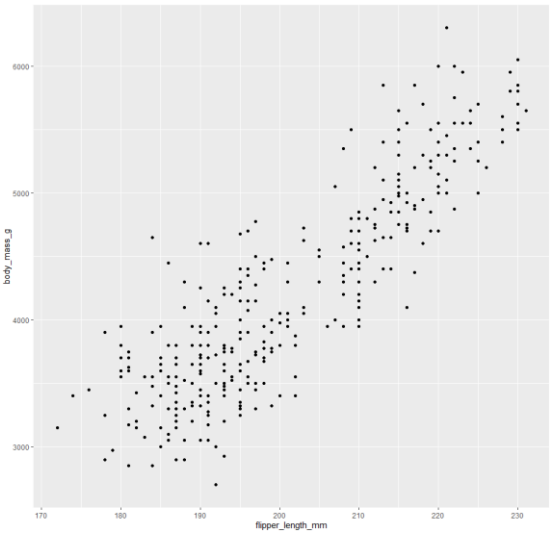
```
flipper_density <- ggplot(data = penguins, aes(x = flipper_length_mm)) +
geom_density(aes(fill = species), alpha = 0.5) + labs(x = 'Flipper length
(mm)', y = 'Frequency') + theme_classic()
```

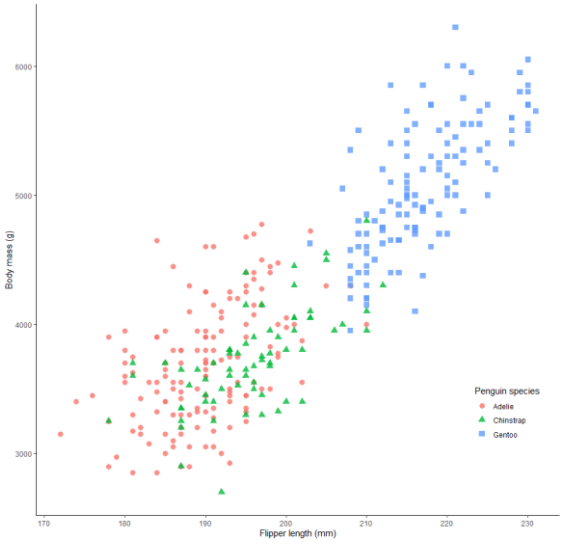


You can see the code is mostly the same, except this time we changed the geom function to a density plot, and added a theme. There are lots of different themes you can try, such as `theme_minimal()`, `theme_bw()`, `theme_gray()`, and `theme_dark()`. You graph should look something like this:

Plotting two variables – Flipper length against Body mass

This time we are going to plot the flipper length (x axis) against body mass (y axis). In addition, we are going to iteratively build up the code. Your graph will start like the plot on the left and finish like the one on the right.





Iteration One

```
ggplot(data = penguins, aes(x = flipper_length_mm, y = body_mass_g)) + geom_point()
```

As you can see this plot is limited in how much information you can infer from it. Can you tell how the different species mix? This code is a starting point which you can build on to make a better plot.

Iteration Two

```
ggplot(data = penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
geom_point(aes(color = species))
```

This time we have added some extra detail to the `geom_point` function. This has now coloured the data points according to what species that penguin belongs to.

Iteration Three

```
ggplot(data = penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
geom_point(aes(color = species, shape = species), alpha = 0.8, size = 3)
```

Now we have not only coloured the datapoints by species but given each species a unique shape as well. Using the `alpha` argument, we have set the transparency to 80% and also set the size of each datapoint using the `size` argument.

Iteration Four

This time we are going to add some of our own labelling, however some of the code is missing. Replace the question marks with the appropriate code.

```
ggplot(data = ?, ?(x = ?, y = ?)) + geom_point(aes(color = species, shape =
species), alpha = 0.8, size = 3) + labs(x = "Flipper length (mm)", y = ?, color =
"Penguin species", shape = 'Penguin species')
```

Iteration Five

This time, using the `theme()` function, we are going to customise the legend position and the legend background.

```
ggplot(data = ?, aes(?)) + geom_point(aes(color = species, shape = species), alpha
= 0.8, size = 3) + labs(?) + theme(legend.position = c(0.9, 0.2), legend.background
= element_rect(fill = "white", color = NA))
```

Iteration Six

Finally, let's overlay an appearance theme in addition to our previous code.

```
ggplot(?) + geom_point(?) + labs(?) + theme_classic() + theme(?)
```

Saving your graph

You can use the function `ggsave()` to export your plot as a pdf, png etc.

```
ggsave('my_scatterplot.pdf')
```

There are many more settings within `ggsave()`. You can look it up in the help menu to see the different arguments available to you. These include setting the height and width of your image, alongside its dpi quality. By default `ggsave()` will export your most recent plot.

Plotting two variables – adding a regression line and performing an ANOVA

For this example we are going to plot the length of the penguin bills on the x axis against the depth of the bills on the y axis.

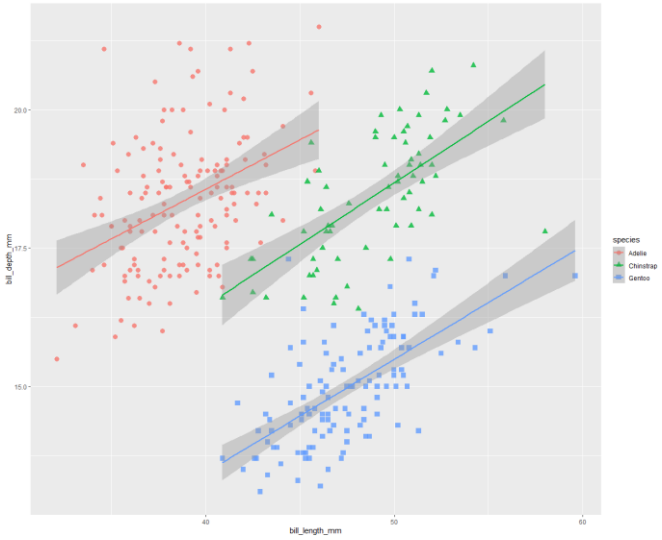
Generating the basic plot

```
bill_anatomy <- ggplot(data = ?, aes(x = ?, y = ?, group = species)) +
  geom_point(aes(color = species, shape = species), size = 3, alpha = 0.8)
```

Try and fill in the missing code. Your plot should look like this:

Adding regression lines

Regression lines can easily be added to a scatterplot by adding another geom function to it, `geom_smooth()`. Here we are going to add one regression line per species, so three regression lines in total. Your plot should look like this:



You can remove the confidence intervals overlayed on your regression lines by including the argument `se = FALSE` within the `geom_smooth()` function.

```
bill_anatomy <- bill_anatomy + geom_smooth(aes(color = ?), method = 'lm')
```

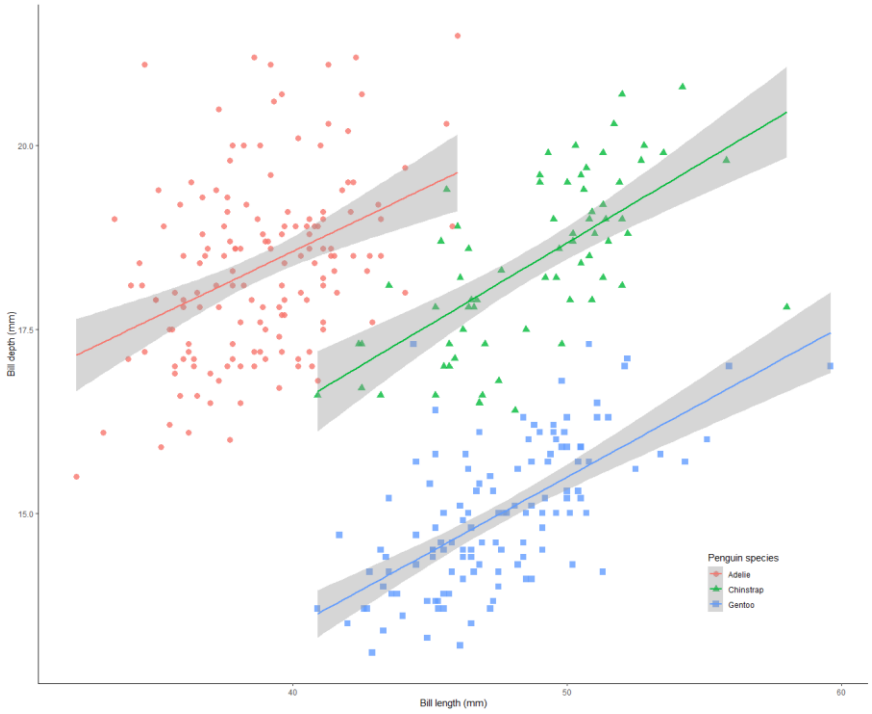
The `method = 'lm'` argument above specifies the way the regression line is drawn, in this case `'lm'` means linear model.

Final tidying

Now we can add some final tidying and customisation.

```
bill_anatomy <- bill_anatomy + theme_classic() + labs(?) + theme(legend.position = c(0.85, 0.15), legend.background = element_rect(fill = 'white', color = NA))
```

Your graph should look something like this:



Performing an ANOVA

Let's examine the effect of bill length, penguin species and their interaction on bill depth using an ANOVA. Run the following code below:

```
model <- aov(bill_depth_mm ~ bill_length_mm * species, data = penguins)
```

The `A ~ B * C` notation means that the anova looks at bill length vs bill depth, species vs bill depth and the interaction of bill length and species on bill depth. Now you can run the following code:

```
drop1(model, scope = ~., test = "F")
```

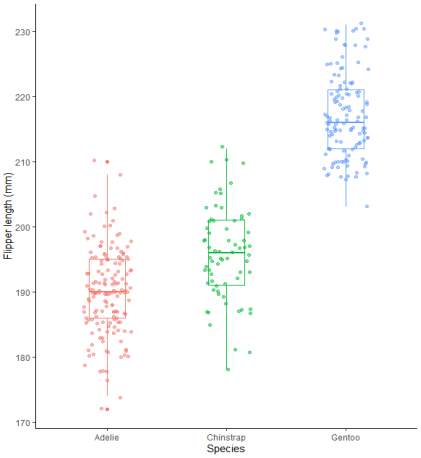
The above code iteratively removes each term from the ANOVA to explore its influence on the R^2 value. The `scope` argument determines what is to be removed, in this case all terms will be removed. The `test = 'F'` argument carries out an F test on this, an integral part of an ANOVA.

Drawing Boxplots

This time we are going to create a series of boxplots, plotting flipper length on the y axis separated by species on the x axis. For boxplots you use the `geom_boxplot` function.

```
flipper_boxplots <- ggplot(data = ?, aes(?)) + geom_boxplot(aes(color = species), width = 0.5, show.legend = FALSE) + geom_jitter(aes(color = species), alpha = 0.5, show.legend = FALSE, position = position_jitter(width = 0.2, seed = 0)) + theme_classic() + labs(?)
```

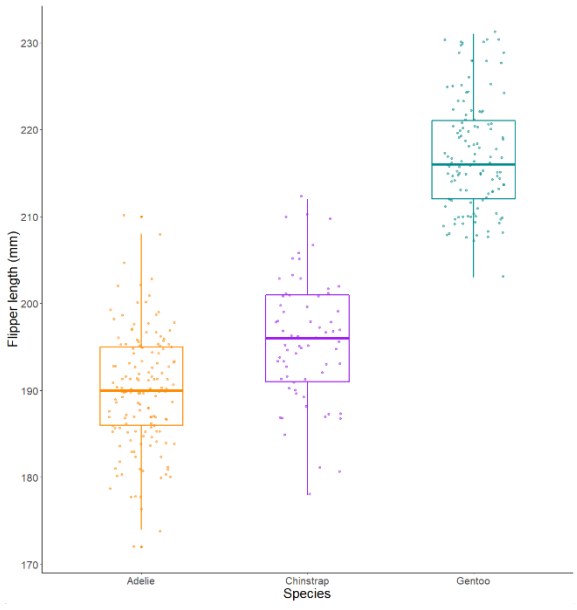
You will notice the function `geom_jitter()` was also included. This function adds a small amount of random variation to the location of each datapoint and is a useful way of handling overplotting (where there is a lot of overlap of your datapoints). Your boxplots should look something like this:



Some other useful things to edit with graphs is their line thickness, text size and selecting your own colours. The below code shows you how to do that with these boxplots:

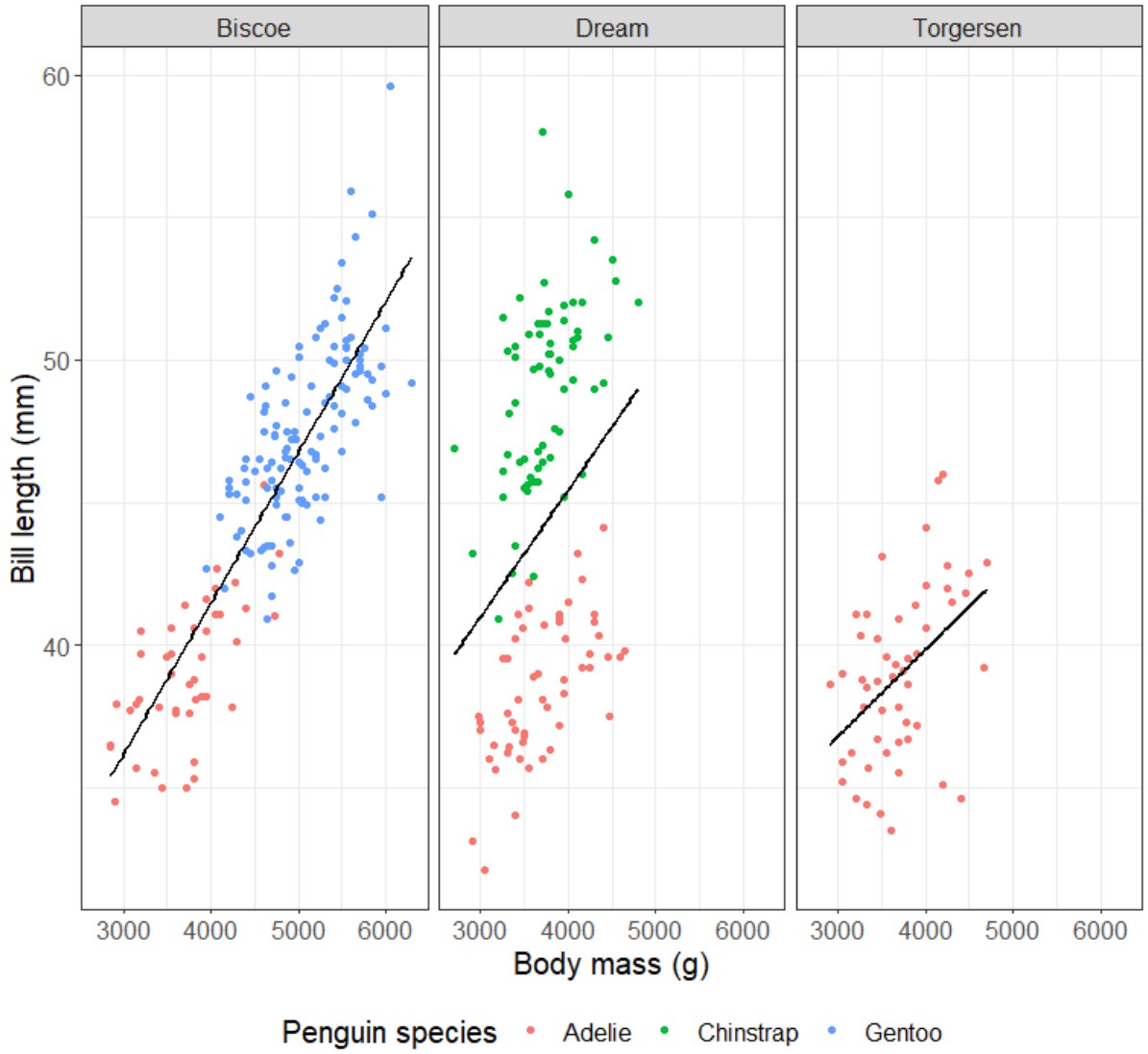
```
flipper_boxplots <- ggplot(data = penguins, aes(y = flipper_length_mm, x = species)) + geom_boxplot(aes(color = species), width = 0.5, show.legend = FALSE, size = 0.9) + geom_jitter(aes(color = species), alpha = 0.5, show.legend = FALSE, position = position_jitter(width = 0.2, seed = 0)) + theme_classic() + labs(x = 'Species', y = 'Flipper length (mm)') + theme(text = element_text(size = 20)) + scale_colour_manual(values = c('darkorange', 'purple', 'cyan4'))
```

Your boxplots should now look like this:



Final task

Your final task is to prepare a scatter plot of body mass on the x axis against bill length on the y axis. Separate the three species by colour. Add a regression line to your plot. Add the function `facet_wrap(~island)` on the end of your code. The goal is for your plot to look like this:



HINT- the framework for your code should be something close to:

```
ggplot(?) + geom_point(?) + geom_smooth(?) + labs(?) + theme_bw() + theme(?) +
facet_wrap(~island)
```

Remember to save your plot with `ggsave()`, specifying whether you want it as a png or pdf by ending the filename with .png or .pdf. You can also specify the width and height of your file with `ggsave()`.

Extra Resources

The R Graph Gallery: https://www.r-graph-gallery.com/index.html

This website teaches you how to plot over 39 different plots, starting off with basic code and building up to more complex graphs.

R for Data Science: https://r4ds.had.co.nz/

This is a very easy to use webpage-based book which teaches you the very basics of plotting and dataset management (filtering, sub-setting, ordering etc.) through to more complex functions.

Quick R: https://www.statmethods.net/

This is a great website for learning how to perform statistical analyses using R.

R Graphics Cookbook: https://r-graphics.org/

This is another very easy to use webpage-based book focused solely on generating high-quality plots, with more than 150 different sets of code for generating a variety of different graphs.

swirl: https://swirlstats.com/

This website shows you how to download the R package swirl. swirl contains numerous interactive courses such as exploratory data analysis or a just a simple introduction to R.

Acknowledgements

Palmerpenguins artwork by @allison_horst
Palmerpenguins Developers: Allison Horst, Alison Hill, Kristen Gorman

Palmerpenguins: https://allisonhorst.github.io/palmerpenguins/index.html

Tidyverse: https://www.tidyverse.org/