

# Queen Elizabeth Scholarship - Advanced Scholars Program: R Workshop #2

## The tidyverse

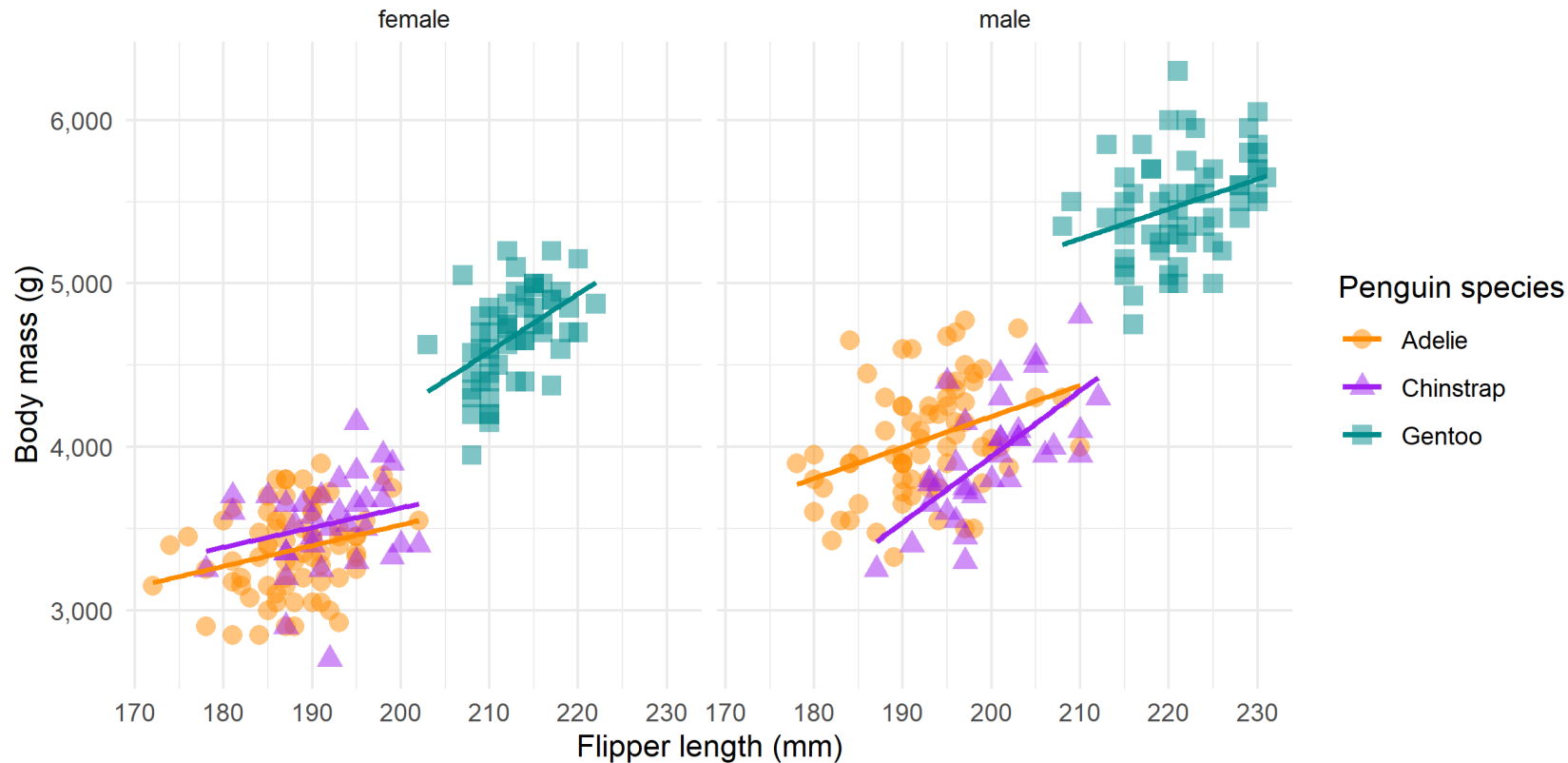
Melissa Van Bussel, Benjamin Burr, Amin Nabavi, Dr. Shirley Mills

Queen Elizabeth Scholarship - Advanced Scholars Program

July 29th, 2021

# Today's workshop

- Review of key concepts from Workshop #1
- An example of how to use R to perform data analysis on a dataset



# The code for today

The code for today's workshop is available at the following link: <https://github.com/melissavanbussel/QES-AS-R-Workshops>

Instructions:

1. Open RStudio
2. File > New File > R Script
3. Copy/paste the code from GitHub into the R Script
4. Save the script onto your computer (Ctrl+S)

# Review of key concepts from Workshop 1

# R Scripts vs. the R console

- Code that is run in the console only does not get saved when you close RStudio.
- If you type the code in your R Script and save the file, you will be able to run that code again in the future.
- The console is great for quick calculations, but anything that you'd like to run again should be saved in your R script.

# Comments

Any lines of code that start with hashtags are comments and do not get evaluated as code.

```
# This is a comment, it does not get evaluated as code  
# You can write anything you want in a comment!  
# The line below does not start with a hashtag, so it is evaluated  
print("Hello, world!")
```

```
## [1] "Hello, world!"
```

# Variable assignment

To assign an object to a variable in R, use the `←` assignment operator.

```
years ← c(2010, 2020, 2021)
```

Any line of code that has the assignment operator will have no output. Type the name of the variable to see the contents of the variable.

```
years
```

```
## [1] 2010 2020 2021
```

# Functions in R

Just like in Mathematics, a function in R is something that takes some input(s) and returns an output.

They take the form `function_name(argument_1 = val_1, argument_2 = val_2, ...)`

```
mean(x = years)
```

```
## [1] 2017
```

If you pass the arguments to the function in the correct order, you don't need to name the arguments.

```
mean(years)
```

```
## [1] 2017
```



# Viewing documentation

To view the documentation for a function, type `?functionName`. The documentation will appear in the bottom right pane of RStudio.

```
?mean
```

# Missing values

Missing values are represented as `NA`s in R. If there is even just one `NA` included in a calculation, the result will be `NA`.

```
mean(c(1, 2, 3)) # No problem
```

```
## [1] 2
```

```
mean(c(1, NA, 3)) # Result is NA
```

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

Similar results occur for other summary functions, such as:

- `max`
- `min`
- `median`

One option is to use the option `na.rm = TRUE` inside of these summary functions.

```
mean(c(1, NA, 3), na.rm = TRUE)
```

```
## [1] 2
```

# Packages

There are a lot of functions that are "built in" to R (they are automatically available when R is installed).

Sometimes, we want to use more advanced techniques. We could write this code ourselves, but this is time-consuming. Rather than re-inventing the wheel, we install a package that someone else has created.

Packages typically contain functions, but can also include other things like datasets (for example).

```
# Only needs to be done once  
install.packages("tidyverse")  
  
# Needs to be done each time RStudio is opened  
# (if you want to use the package)  
library(tidyverse)
```

# Using packageName::functionName

Sometimes, you may see R code on the internet that takes the form `packageName::functionName`.

This allows you to use the `functionName` function from the `packageName` package, without loading the `packageName` package.

This is also typically used if there is more than one package with the same function name. In this case, it is useful to explicitly state which package you're using.

```
scales::dollar(123456)
```

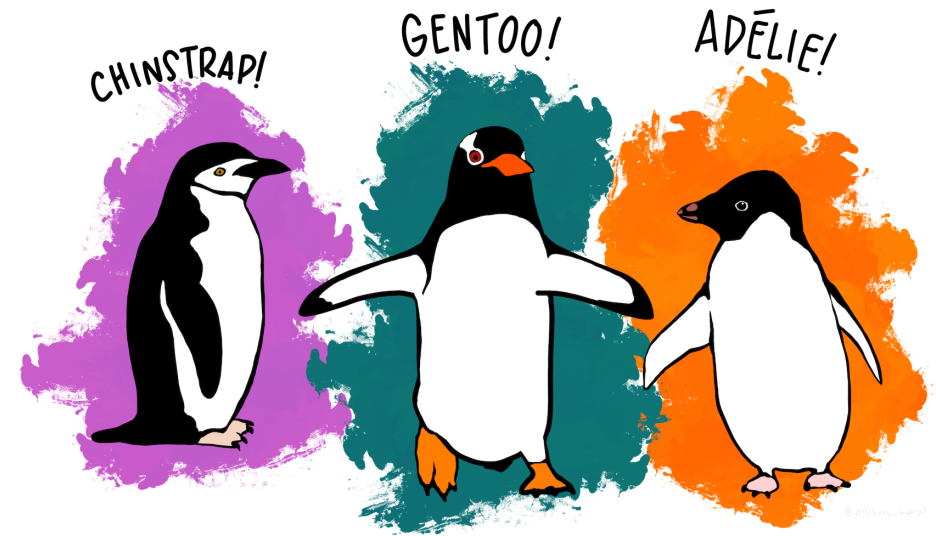
```
## [1] "$123,456"
```

# Data frames and the View function

A data frame is a table (two-dimensional) with columns and rows. We can examine the contents of a data frame by using the `View` function.

Recall the `penguins` dataset from the `palmerpenguins` package from Workshop 1:

- 344 observations of 7 variables
- Information about Adelie, Chinstrap, and Gentoo penguins
- Observed on islands near Palmer Station in Antarctica



```
library(palmerpenguins)
View(penguins)
```

# Tibbles

The `str` function shows us the **structure** of a data frame. It tells us:

- The number of rows and columns
- The data types of each of the variables
- A preview of the data (first few rows)

```
str(penguins)
```

```
## tibble [344 x 8] (S3: tbl_df/tbl/data.frame)
## $ species      : Factor w/ 3 levels "Adelie","Chinstrap",..: 1 1 1 1 1 1 1 1 1 1 1 ...
## $ island       : Factor w/ 3 levels "Biscoe","Dream",..: 3 3 3 3 3 3 3 3 3 3 3 ...
## $ bill_length_mm : num [1:344] 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
## $ bill_depth_mm  : num [1:344] 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
## $ flipper_length_mm: int [1:344] 181 186 195 NA 193 190 181 195 193 190 ...
## $ body_mass_g    : int [1:344] 3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
## $ sex           : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
## $ year          : int [1:344] 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 ...
```

We can see that the `penguins` dataset is both a "tibble" and a "data frame". Tibbles are a type of data frame.

- They are "simpler" (and therefore faster)
- They work great with the tidyverse, which we will learn about in a second!

# Introduction to the tidyverse

# What is the tidyverse?



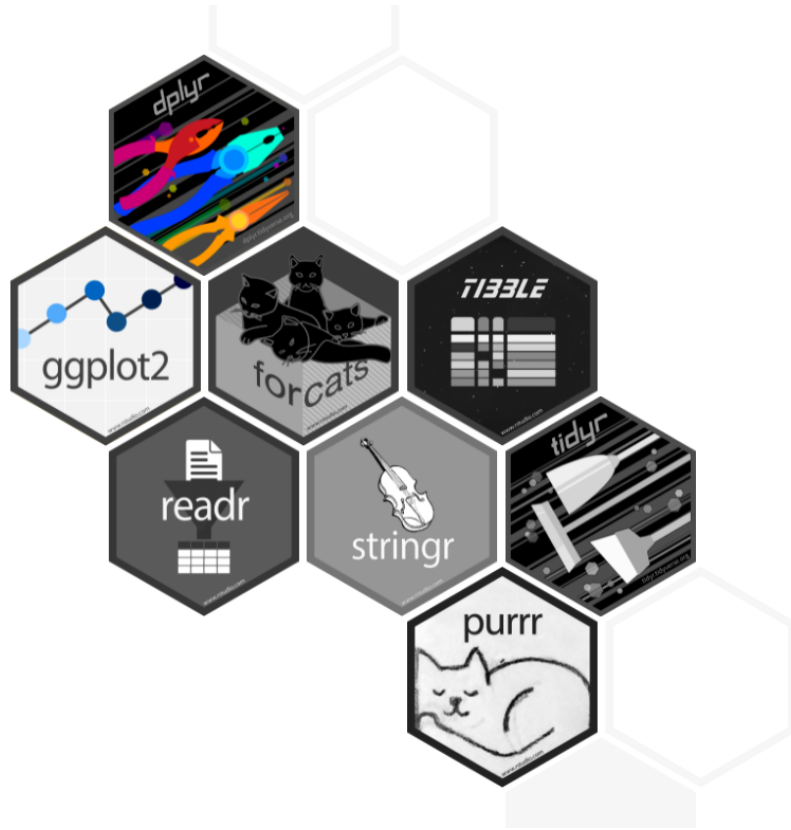
The tidyverse is a set of packages in R.

- dplyr
- tidyr
- ggplot2
- readr
- stringr
- etc...



# What is the tidyverse?

Today, we will focus on `dplyr` and `ggplot2`.



# Why use the tidyverse?

- Consistency (common underlying philosophy)
- Speed (many tasks implemented in C++)
- Easier to learn (learning one package = learning many)

# Loading the tidyverse

By installing (or loading) the tidyverse, we're actually installing (or loading) many packages at once, all with one line of code.

```
# If not already installed
install.packages("tidyverse")

# Equivalent to
install.packages("dplyr")
install.packages("ggplot2")
install.packages("stringr")
...

# Once installed
library(tidyverse)

# Equivalent to:
library(dplyr)
library(ggplot2)
library(stringr)
...
```

# Using dplyr for data exploration

# The select function

- select: select a subset of **columns** from a data frame

```
library(tidyverse)
select(.data = penguins, bill_length_mm, bill_depth_mm)
```

```
## # A tibble: 344 x 2
##   bill_length_mm bill_depth_mm
##           <dbl>         <dbl>
## 1           39.1           18.7
## 2           39.5           17.4
## 3           40.3            18
## 4            NA            NA
## 5           36.7           19.3
## 6           39.3           20.6
## 7           38.9           17.8
## 8           39.2           19.6
## 9           34.1           18.1
## 10          42            20.2
## # ... with 334 more rows
```

# The select function

To select all columns **except** for ones we specify, use the minus sign:

```
select(.data = penguins, -species)
```

```
## # A tibble: 344 x 7
##   island  bill_length_mm bill_depth_mm flipper_length_~ body_mass_g sex   year
##   <fct>         <dbl>         <dbl>         <int>         <int> <fct> <int>
## 1 Torger~         39.1           18.7           181           3750 male   2007
## 2 Torger~         39.5           17.4           186           3800 fema~  2007
## 3 Torger~         40.3            18           195           3250 fema~  2007
## 4 Torger~          NA            NA            NA            NA <NA>   2007
## 5 Torger~         36.7           19.3           193           3450 fema~  2007
## 6 Torger~         39.3           20.6           190           3650 male   2007
## 7 Torger~         38.9           17.8           181           3625 fema~  2007
## 8 Torger~         39.2           19.6           195           4675 male   2007
## 9 Torger~         34.1           18.1           193           3475 <NA>   2007
## 10 Torger~         42            20.2           190           4250 <NA>   2007
## # ... with 334 more rows
```

The result is all columns in the `penguins` data frame, other than the `species` column.

# The filter function

- `filter`: select a subset of **rows** from a data frame

A logical condition is specified, and R checks if the condition is true for each row in the data frame. Only rows where the condition is `TRUE` are returned:

```
filter(.data = penguins, bill_length_mm ≥ 55)
```

```
## # A tibble: 5 x 8
##   species island bill_length_mm bill_depth_mm flipper_length_~ body_mass_g sex
##   <fct>   <fct>         <dbl>         <dbl>         <int>         <int> <fct>
## 1 Gentoo  Biscoe           59.6           17            230          6050 male
## 2 Gentoo  Biscoe           55.9           17            228          5600 male
## 3 Gentoo  Biscoe           55.1           16            230          5850 male
## 4 Chinst~ Dream           58            17.8           181          3700 fema~
## 5 Chinst~ Dream           55.8           19.8           207          4000 male
## # ... with 1 more variable: year <int>
```

# Relational and logical operators in R

There are many other relational operators in R:

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Multiple conditions can be combined by using logical operators such as & ("AND") or | ("OR").

For example: `bill_length_mm ≤ 180 | bill_length_mm ≥ 200`



# The pipe operator (%>%)

The pipe operator takes the object on the left of the %>% and feeds it as input to the first argument of the function on the right of the %>%.

The code on the left and the code on the right will produce the **exact** same output, but using the pipe operator (left) is much faster to write and easier to read.

With the pipe operator (one step)

```
penguins %>%  
  select(bill_length_mm, bill_depth_mm) %>%  
  filter(bill_length_mm ≥ 55)
```

Without the pipe operator (two steps)

```
step1 ← select(.data = penguins,  
               bill_length_mm,  
               bill_depth_mm)  
filter(step1, bill_length_mm ≥ 55)
```

# The pipe operator (%>%)

```
penguins %>%  
  select(bill_length_mm, bill_depth_mm) %>%  
  filter(bill_length_mm ≥ 55)
```

```
## # A tibble: 5 x 2  
##   bill_length_mm bill_depth_mm  
##           <dbl>         <dbl>  
## 1           59.6           17  
## 2           55.9           17  
## 3           55.1           16  
## 4           58            17.8  
## 5           55.8           19.8
```

TIP: When reading the code in your head, replace the %>% with the word "THEN".

# Challenge problem

Which of the following will result in a data frame that only contains observations about male penguins?

A:

```
penguins %>%  
  select(bill_length_mm, bill_depth_mm) %>%  
  filter(sex = "male")
```

B:

```
penguins %>%  
  filter(sex = "male") %>%  
  select(bill_length_mm, bill_depth_mm)
```

C: All of the above

# The mutate function

- mutate: create a new variable based on existing variables

Create a new variable called `bill_length_m` that expresses the bill lengths in metres rather than in millimetres:

```
penguins %>%  
  mutate(bill_length_m = bill_length_mm / 1000) %>%  
  select(bill_length_m, bill_length_mm)
```

```
## # A tibble: 344 x 2  
##   bill_length_m bill_length_mm  
##           <dbl>         <dbl>  
## 1         0.0391          39.1  
## 2         0.0395          39.5  
## 3         0.0403          40.3  
## 4         NA             NA  
## 5         0.0367          36.7  
## 6         0.0393          39.3  
## 7         0.0389          38.9  
## 8         0.0392          39.2  
## 9         0.0341          34.1  
## 10        0.042           42  
## # ... with 334 more rows
```

# The group\_by and summarise functions

- The group\_by and summarise functions are often used together
  - group\_by splits the data into groups (user-specified categorical variable)
  - summarise returns a summary value for each group
  - When used together, the resulting data frame has one row per group

```
penguins %>%  
  group_by(species) %>%  
  summarise(num_penguins = n())
```

```
## # A tibble: 3 x 2  
##   species    num_penguins  
## * <fct>         <int>  
## 1 Adelie           152  
## 2 Chinstrap         68  
## 3 Gentoo          124
```

Some things to note:

- You can group by more than one variable as well
- You can use the summarise function to compute more than just one summary value
- While using the group\_by and summarise functions together is quite common, it is possible to use them separately.

# The arrange function

- `arrange`: sorts the data frame by a specified variable
- By default, it sorts in ascending order (smallest to largest)

```
penguins %>%  
  group_by(species) %>%  
  summarise(num_penguins = n()) %>%  
  arrange(num_penguins)
```

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

# The arrange function

- To sort in descending order (largest to smallest), use the `desc` function

```
penguins %>%  
  group_by(species) %>%  
  summarise(num_penguins = n()) %>%  
  arrange(desc(num_penguins))
```

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

# The count function

- The `count` function is a shortcut for `summarise(n = n())`
- Therefore, the `count` function names the variable "n"

```
penguins %>%  
  group_by(species) %>%  
  count() %>%  
  arrange(desc(n))
```

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



# Missing values

Recall that missing values are represented as `NA` in R.

```
penguins %>%  
  group_by(species) %>%  
  summarise(mean_body_mass = mean(body_mass_g))
```

```
## # A tibble: 3 x 2  
##   species    mean_body_mass  
## * <fct>      <dbl>  
## 1 Adelie           NA  
## 2 Chinstrap    3733.  
## 3 Gentoo         NA
```

The `mean` function (as well as many other summary functions) returns `NA` if even one value in the calculation is `NA`.

# Missing values

There are many different ways to deal with missing values (this is an entire topic in itself!). Today, we will just exclude the missing values from our calculations.

**Option 1:** The `na.rm = TRUE` option (only removes for the `body_mass_g` variable)

```
penguins %>%  
  group_by(species) %>%  
  summarise(mean_body_mass = mean(body_mass_g, na.rm = TRUE))
```

```
## # A tibble: 3 x 2  
##   species    mean_body_mass  
## * <fct>          <dbl>  
## 1 Adelie         3701.  
## 2 Chinstrap      3733.  
## 3 Gentoo         5076.
```

# Missing values

**Option 2:** Removing **any** observations with missing values via the `drop_na` function (what we will do for the remainder of today's workshop)

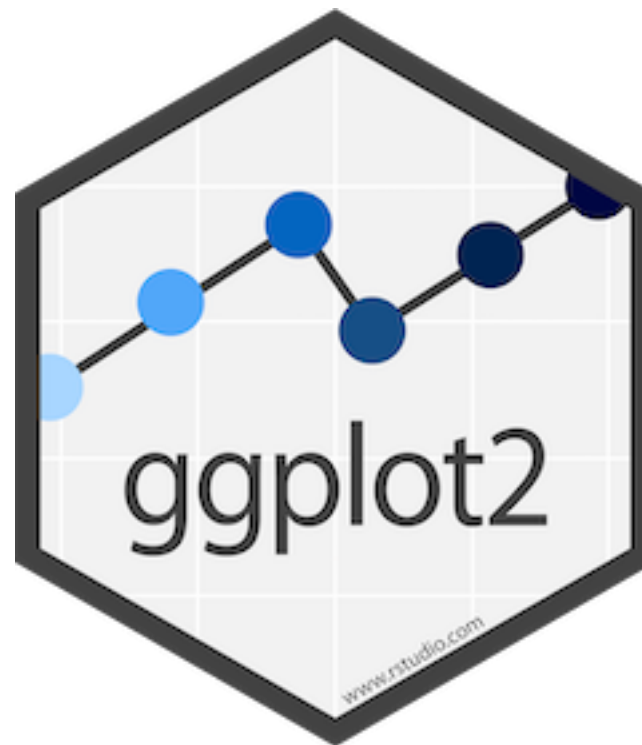
```
penguins2 <- penguins %>%  
  drop_na()  
penguins2 %>%  
  group_by(species) %>%  
  summarise(mean_body_mass = mean(body_mass_g))
```

```
## # A tibble: 3 x 2  
##   species    mean_body_mass  
## * <fct>      <dbl>  
## 1 Adelie      3706.  
## 2 Chinstrap  3733.  
## 3 Gentoo     5092.
```

# Data Visualization with `ggplot2`

# The ggplot2 package

- Part of the tidyverse
- For producing data visualizations
- Understanding ggplot = understanding the "grammar of graphics"



# The Grammar of Graphics

- Language has grammatical elements; combining them together creates a meaningful sentence



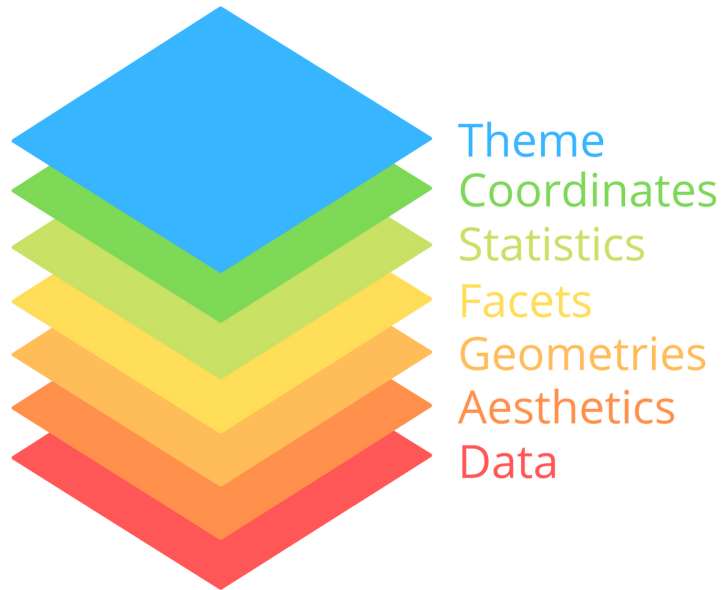
- Data visualizations have their own grammatical elements; combining them together creates a meaningful plot

## All Grammatical Elements

Element	Description
Data	The dataset being plotted.
Aesthetics	The scales onto which we <i>map</i> our data.
Geometries	The visual elements used for our data.
Facets	Plotting small multiples.
Statistics	Representations of our data to aid understanding.
Coordinates	The space on which the data will be plotted.
Themes	All non-data ink.

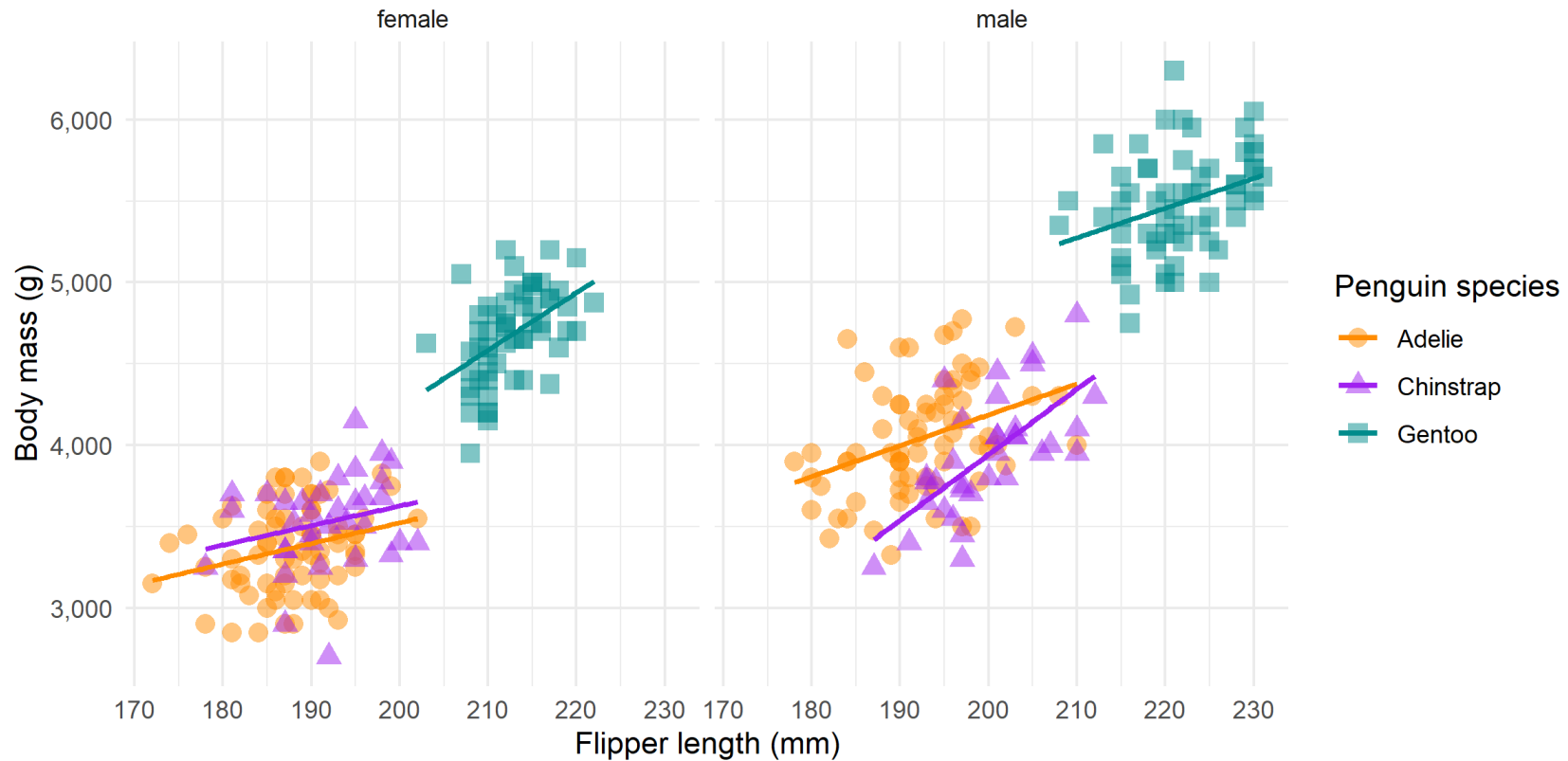
# Grammatical elements

- There are a total of 7 grammatical elements for graphics
- We combine these grammatical elements to communicate meaningfully through data visualizations
- We layer these grammatical elements on top of one another, typically with one layer per grammatical element



# Example

In the next few slides, we'll build the plot below, one layer at a time.





# Layer 1: Data

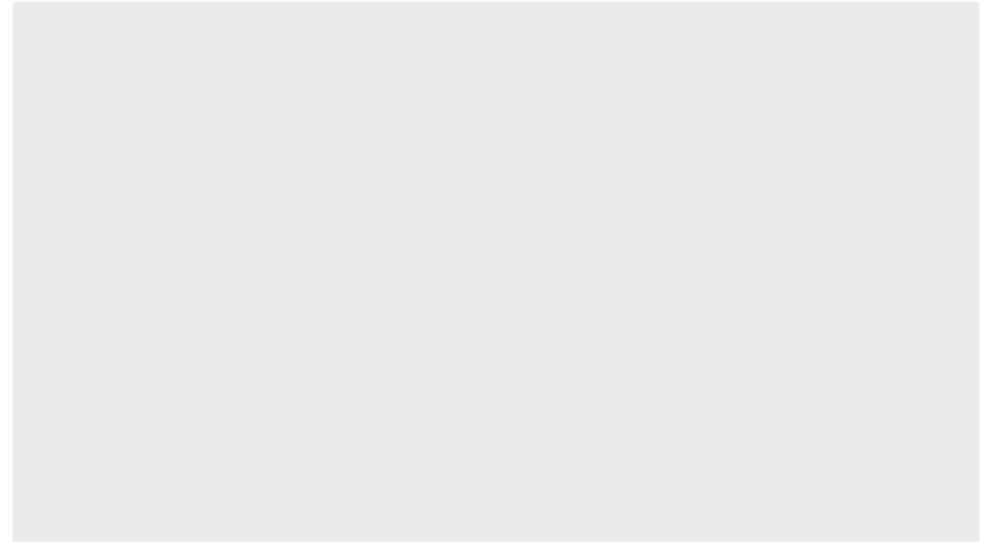
- The data layer is where we define which dataset we'll use for our visualization
- For `ggplot`, this must be a data frame
- This differs from creating plots using base R

```
# The penguins dataset is a tibble, which is a type of data frame  
head(penguins)
```

```
## # A tibble: 6 x 8  
##   species island bill_length_mm bill_depth_mm flipper_length_~ body_mass_g sex  
##   <fct>   <fct>         <dbl>         <dbl>         <int>         <int> <fct>  
## 1 Adelie  Torge~           39.1           18.7           181           3750 male  
## 2 Adelie  Torge~           39.5           17.4           186           3800 fema~  
## 3 Adelie  Torge~           40.3            18           195           3250 fema~  
## 4 Adelie  Torge~           NA            NA            NA            NA <NA>  
## 5 Adelie  Torge~           36.7           19.3           193           3450 fema~  
## 6 Adelie  Torge~           39.3           20.6           190           3650 male  
## # ... with 1 more variable: year <int>
```

# Layer 1: Data

```
# Equivalent to ggplot(data = penguins2)  
penguins2 %>%  
  ggplot()
```



# Layer 2: Aesthetics

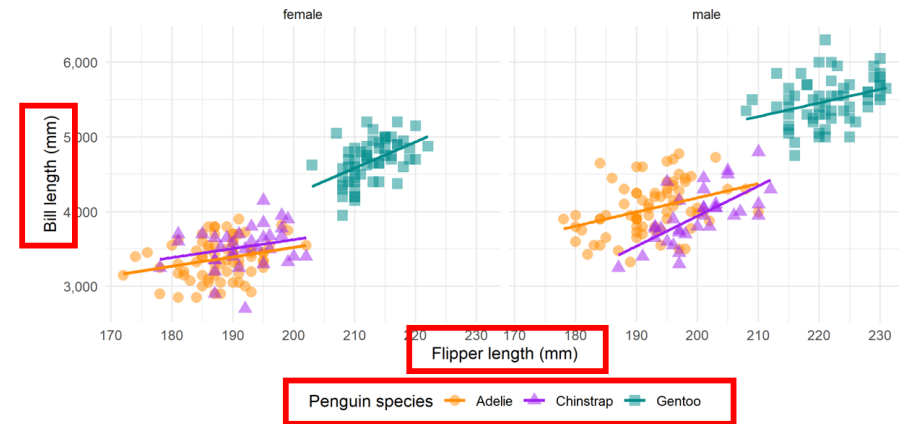
We "map" variables from our dataset onto "aesthetics" on our plot.

flipper\_length\_mm  $\Rightarrow$  x-axis

body\_mass\_g  $\Rightarrow$  y-axis

species  $\Rightarrow$  colour

species  $\Rightarrow$  shape

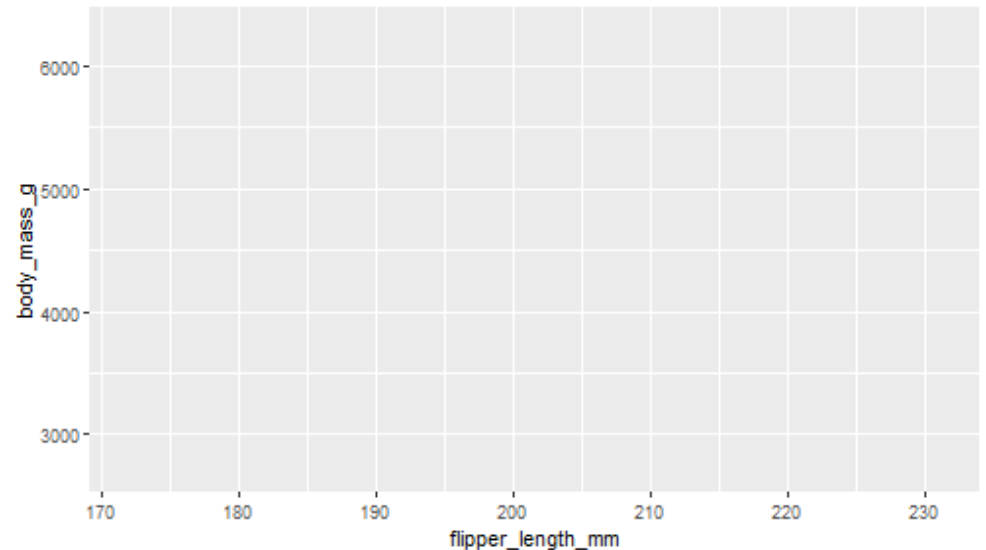


Notice how a variable can be mapped to more than one aesthetic.

# Layer 2: Aesthetics

- We use the `aes` function inside of the `ggplot` function
- On the left hand side of the equals sign: aesthetic we're mapping to
- On the right hand side of the equals sign: variable name from our dataset

```
penguins2 %>%  
  ggplot(aes(x = flipper_length_mm,  
             y = body_mass_g,  
             colour = species,  
             shape = species))
```



The resulting plot is still "empty", but we can see that the variables have now been mapped to the correct axes.

# Layer 3: Geometries

The geometries layer controls which type of data visualization we are creating.

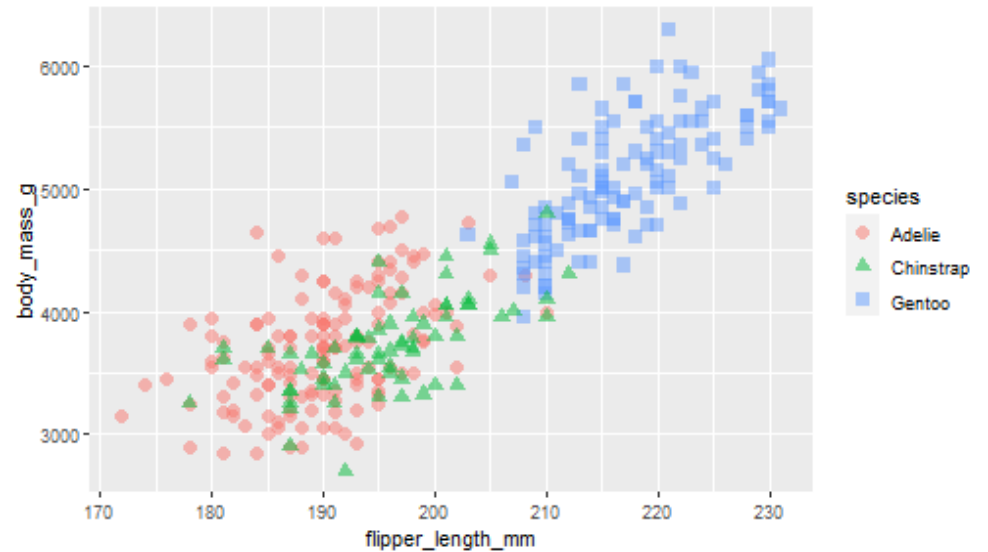
Some examples:

- `geom_boxplot()`: For creating boxplots
- `geom_point()`: For creating scatterplots
- `geom_line()`: For creating line plots
- `geom_histogram()`: For creating histograms

# Layer 3: Geometries

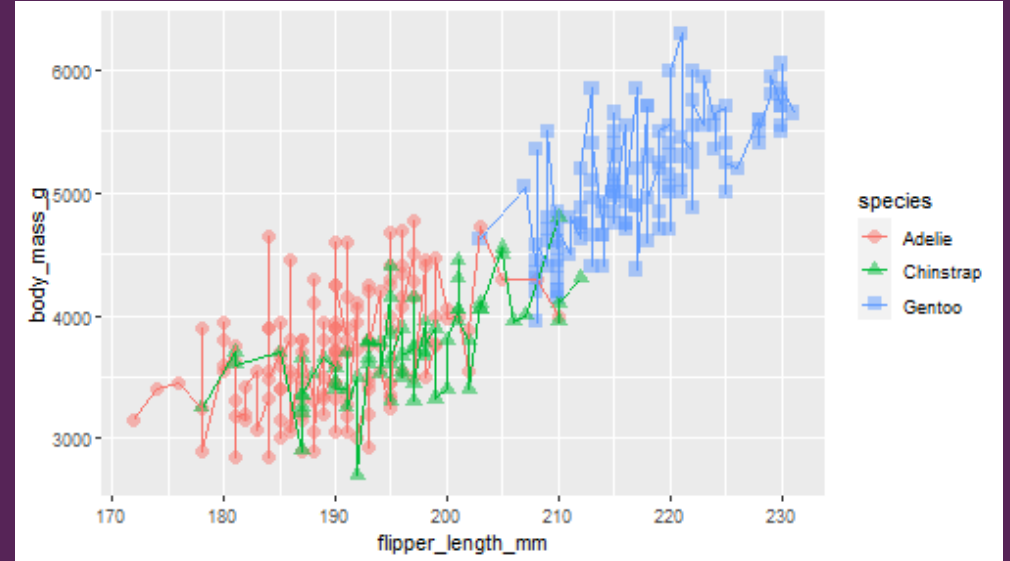
Notice that we use the addition sign after the `ggplot` function is finished.

```
penguins2 %>%  
  ggplot(aes(x = flipper_length_mm,  
             y = body_mass_g,  
             colour = species,  
             shape = species)) +  
  geom_point(size = 3, alpha = 0.5)
```



# Challenge problem

```
penguins2 %>%  
  ggplot(aes(x = flipper_length_mm,  
             y = body_mass_g,  
             colour = species,  
             shape = species)) +  
  geom_point(size = 3, alpha = 0.5) +  
  geom_line()
```



We've added one more geometry compared to the previous slide, by using the `geom_line` function. Which of the following statements is true?

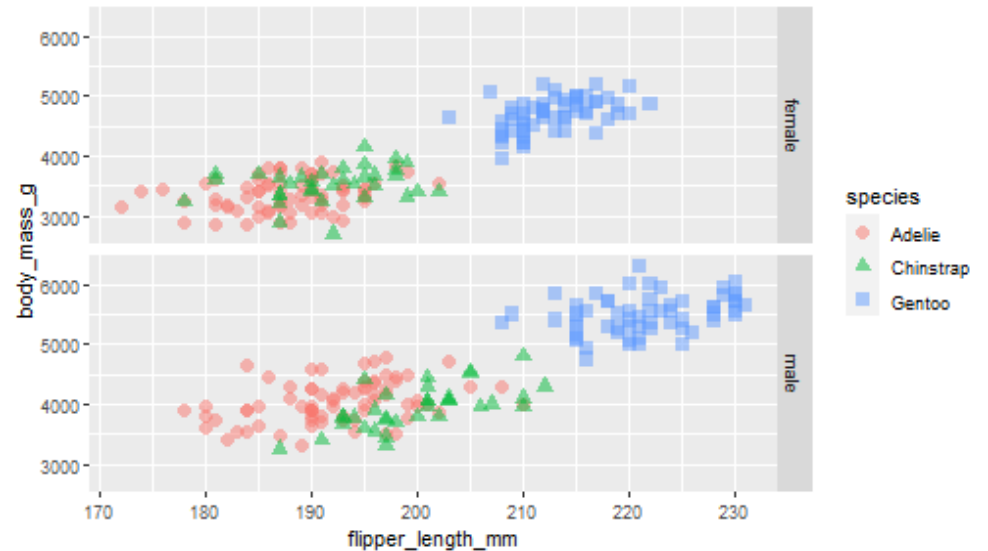
- a) The plot is obviously not a meaningful plot, but the code is correct.
- b) The plot is obviously not a meaningful plot, **and** the code is problematic as well -- we are on "Layer 3: Geometries" but we have 4 layers

# Layer 4: Facets

Facets allow us to create a grid of plots, one for each level of a categorical variable in our dataset.

For vertical facets, we use `variableName ~ .`.

```
penguins2 %>%  
  ggplot(aes(x = flipper_length_mm,  
             y = body_mass_g,  
             colour = species,  
             shape = species)) +  
  geom_point(size = 3, alpha = 0.5) +  
  facet_grid(sex ~ .) # Vertical
```





# Layer 4: Facets

For horizontal facets, we use `. ~ variableName`

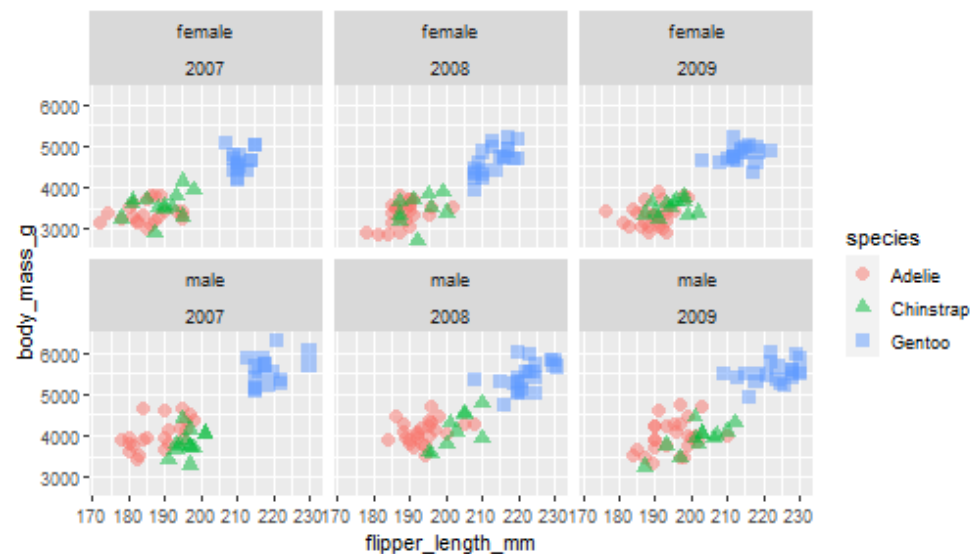
```
penguins2 %>%  
  ggplot(aes(x = flipper_length_mm,  
             y = body_mass_g,  
             colour = species,  
             shape = species)) +  
  geom_point(size = 3, alpha = 0.5) +  
  facet_grid(. ~ sex) # Horizontal
```



# Layer 4: Facets

We can also group by two variables by using the `facet_wrap` function.

```
penguins2 %>%  
  ggplot(aes(x = flipper_length_mm,  
             y = body_mass_g,  
             colour = species,  
             shape = species)) +  
  geom_point(size = 3, alpha = 0.5) +  
  facet_wrap(. ~ sex + year)  
# Row variable: sex; Column variable: year
```



# Layer 5: Statistics

The "statistics" grammatical elements allow us to perform statistical calculations on our existing ggplot.

To add a least squares regression line, use `stat_smooth`:

- "lm" stands for "Linear Model"
- "se" stands for "Standard Error"

```
penguins2 %>%  
  ggplot(aes(x = flipper_length_mm,  
             y = body_mass_g,  
             colour = species,  
             shape = species)) +  
  geom_point(size = 3, alpha = 0.5) +  
  facet_grid(. ~ sex) +  
  stat_smooth(method = "lm", se = FALSE)
```



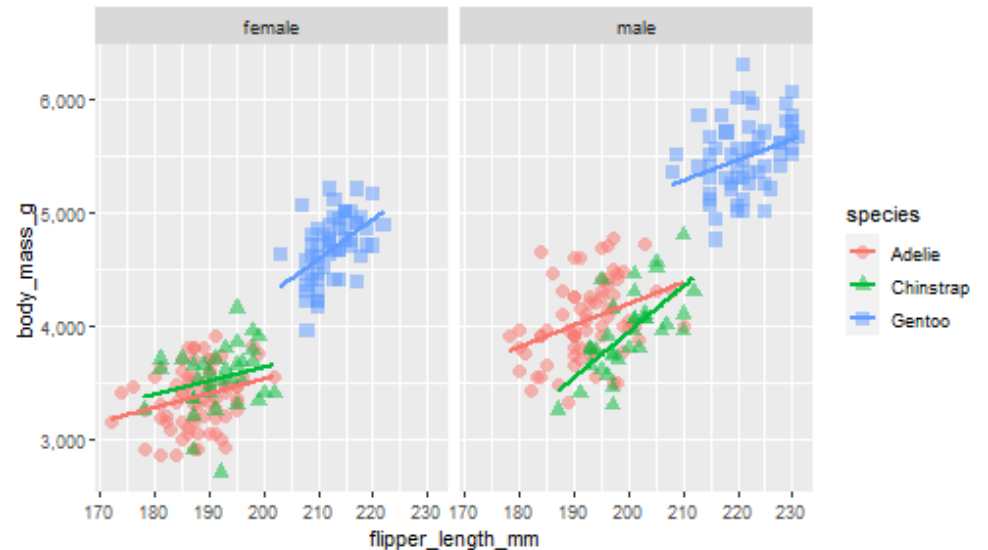
# Layer 6: Coordinates

The coordinates grammatical element allows us to control the way each coordinate (aesthetic) looks on our plot.

They take the form `scale_aestheticName_transformationType`.

To add commas to the values along the y-axis:

```
penguins2 %>%  
  ggplot(aes(x = flipper_length_mm,  
             y = body_mass_g,  
             colour = species,  
             shape = species)) +  
  geom_point(size = 3, alpha = 0.5) +  
  facet_grid(. ~ sex) +  
  stat_smooth(method = "lm", se = FALSE) +  
  scale_y_continuous(labels = scales::comma)
```

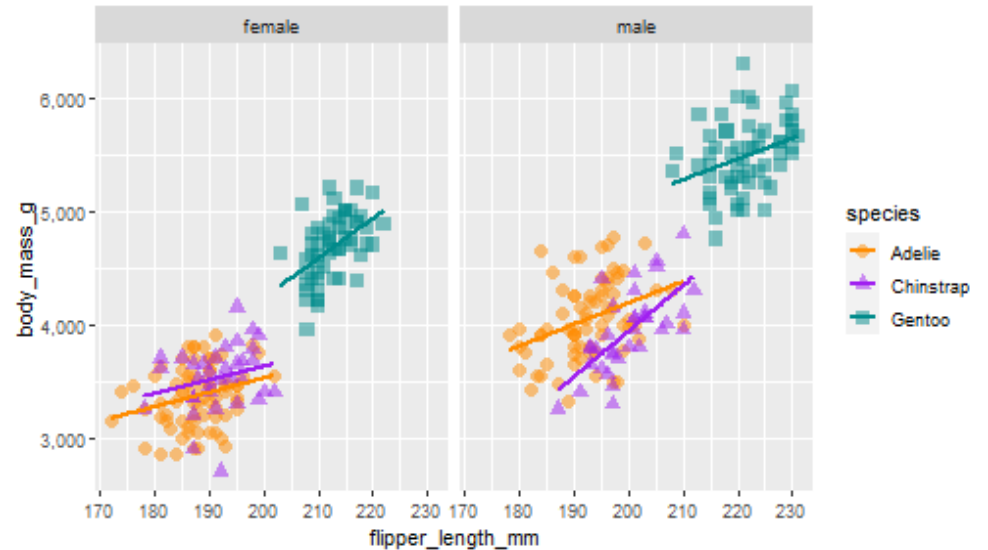


Since we previously mapped the `bill_length_mm` variable to the `y` aesthetic, `scale_y_continuous` changes the y-axis (which represents the bill length).

# Layer 6: Coordinates

To (manually) change the colours used for the `species` variable:

```
penguins2 %>%  
  ggplot(aes(x = flipper_length_mm,  
             y = body_mass_g,  
             colour = species,  
             shape = species)) +  
  geom_point(size = 3, alpha = 0.5) +  
  facet_grid(. ~ sex) +  
  stat_smooth(method = "lm", se = FALSE) +  
  scale_y_continuous(labels = scales::comma) +  
  scale_colour_manual(values = c("darkorange",  
                                 "purple", "cyan4"))
```



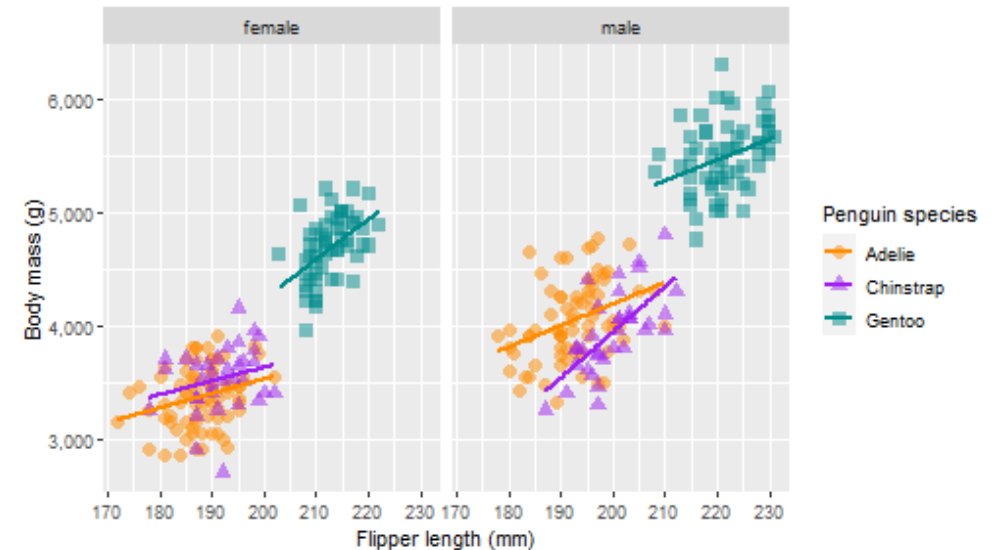
Since we previously mapped the `species` variable to the `colour` aesthetic, `scale_colour_manual` changes the colours for the `species` variable.

# Layer 7: Theme

- The "theme" grammatical elements control the non-data related aspects of the plot
- Changing the theme is useful for creating publication-ready plots

Let's start by changing the plot labels (aesthetic name goes on the left hand side of the equals sign, text to display goes on the right hand side of the equals sign):

```
penguins2 %>%  
  ggplot(aes(x = flipper_length_mm,  
             y = body_mass_g,  
             colour = species,  
             shape = species)) +  
  geom_point(size = 3, alpha = 0.5) +  
  facet_grid(. ~ sex) +  
  stat_smooth(method = "lm", se = FALSE) +  
  scale_y_continuous(labels = scales::comma) +  
  scale_colour_manual(values = c("darkorange",  
                                 "purple", "cyan4"))  
  
labs(x = "Flipper length (mm)",  
     y = "Body mass (g)",  
     colour = "Penguin species",  
     shape = "Penguin species")
```



# Layer 7: Theme

We can change almost anything imaginable by using the `theme` function. Let's change the position of the legend:

```
penguins2 %>%  
  ggplot(aes(x = flipper_length_mm,  
             y = body_mass_g,  
             colour = species,  
             shape = species)) +  
  geom_point(size = 3, alpha = 0.5) +  
  facet_grid(. ~ sex) +  
  stat_smooth(method = "lm", se = FALSE) +  
  scale_y_continuous(labels = scales::comma) +  
  scale_colour_manual(values = c("darkorange",  
                                 "purple", "cyan4"))  
  
  labs(x = "Flipper length (mm)",  
       y = "Body mass (g)",  
       colour = "Penguin species",  
       shape = "Penguin species") +  
  theme(legend.position = "bottom")
```



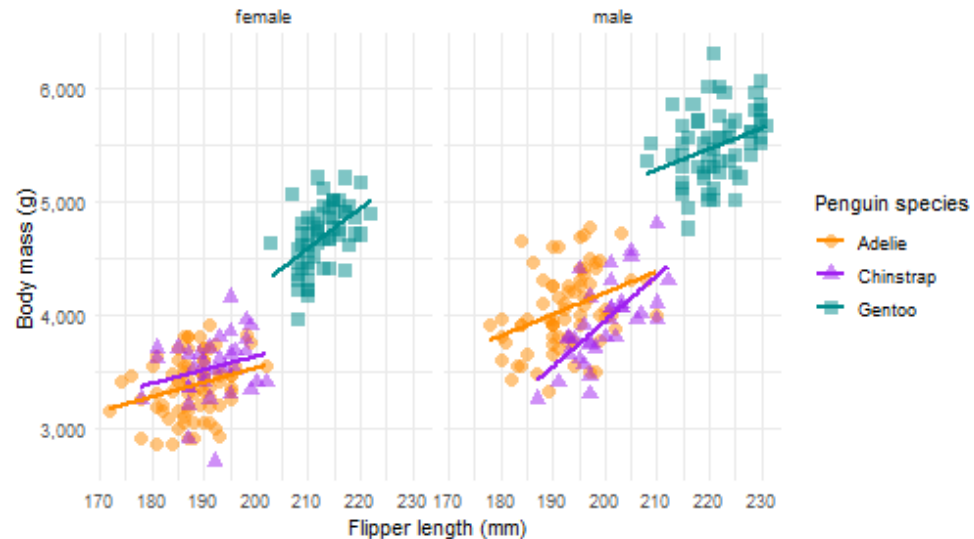
To see a full list of options that can be changed using the `theme` function, see [ggplot2.tidyverse.org/reference/theme.html](https://ggplot2.tidyverse.org/reference/theme.html).

# Layer 7: Theme

There are also some pre-existing themes that you can use in order to save time. For example, the "minimal" theme removes the grey background:

```
penguins2 %>%
  ggplot(aes(x = flipper_length_mm,
             y = body_mass_g,
             colour = species,
             shape = species)) +
  geom_point(size = 3, alpha = 0.5) +
  facet_grid(. ~ sex) +
  stat_smooth(method = "lm", se = FALSE) +
  scale_y_continuous(labels = scales::comma) +
  scale_colour_manual(values = c("darkorange",
                                "purple", "cyan4"))

labs(x = "Flipper length (mm)",
     y = "Body mass (g)",
     colour = "Penguin species",
     shape = "Penguin species") +
theme_minimal()
```





# Challenge problem

**If we want to use the minimal theme AND change the legend position to "top", which of the following will do this?**

- a) `theme_minimal() + theme(legend.position = "top")`
- b) `theme(legend.position = "top") + theme_minimal()`
- c) Both will give the desired result

# Saving a ggplot

To save a plot created using ggplot, use the `ggsave` function.

```
# Assign plot object to a variable
p <- penguins2 %>%
  ggplot(aes(x = flipper_length_mm,
             y = body_mass_g,
             colour = species,
             shape = species)) +
  geom_point(size = 3, alpha = 0.5) +
  facet_grid(. ~ sex) +
  stat_smooth(method = "lm", se = FALSE) +
  scale_y_continuous(labels = scales::comma) +
  scale_colour_manual(values = c("darkorange",
                                "purple", "cyan4")) +

  labs(x = "Flipper length (mm)",
       y = "Body mass (g)",
       colour = "Penguin species",
       shape = "Penguin species") +
  theme_minimal() +
  theme(legend.position = "bottom")

# Use the ggsave function to save the plot
ggsave(plot = p,
       filename = "C:/Users/Melissa/Documents/ggplot.png",
       width = 8, height = 4,
       units = "in")
```

# Working directories

- On the last slide, we specified the entire file path
- This is fine if we're saving one plot, but becomes frustrating if we're saving many
- It's a good practice to set your working directory (the location on your computer where R will read in files and save files to)

```
# Determine current working directory  
getwd()
```

```
# Set working directory  
setwd("C:/Users/Melissa/Documents")
```

```
# Now we can save our plot without specifying the entire file path  
ggsave(plot = p,  
        filename = "ggplot.png",  
        width = 8, height = 4,  
        units = "in")
```

# Challenge problem

Can you re-create the following plot?

**Hint:** you will need to set the argument `position = "identity"` inside of the `geom_histogram` function.

