

# Data Science and Data Analytics

The Data Science Workflow II – Visualize

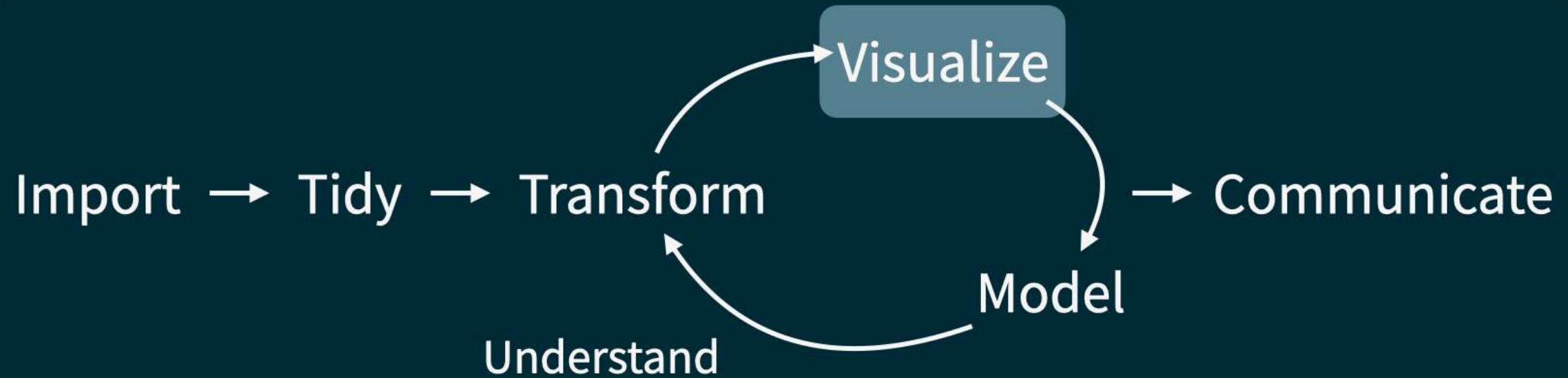
Julian Amon, PhD

Charlotte Fresenius Privatuniversität

April 4, 2025

# First steps in data visualization

# The Data Science workflow – Visualize



Program

# Why even look at data?

- Given the multitude of ways we can describe data numerically (using descriptive statistics), why do we even bother to present data visually?
- The answer is that descriptive statistics can never tell us the full story about our data. Consider the following artificial data set from Anscombe (1973):

It is a collection of four data sets, each with 11 observations for variables  $x$  and  $y$ . Here is the first 2 rows of each:

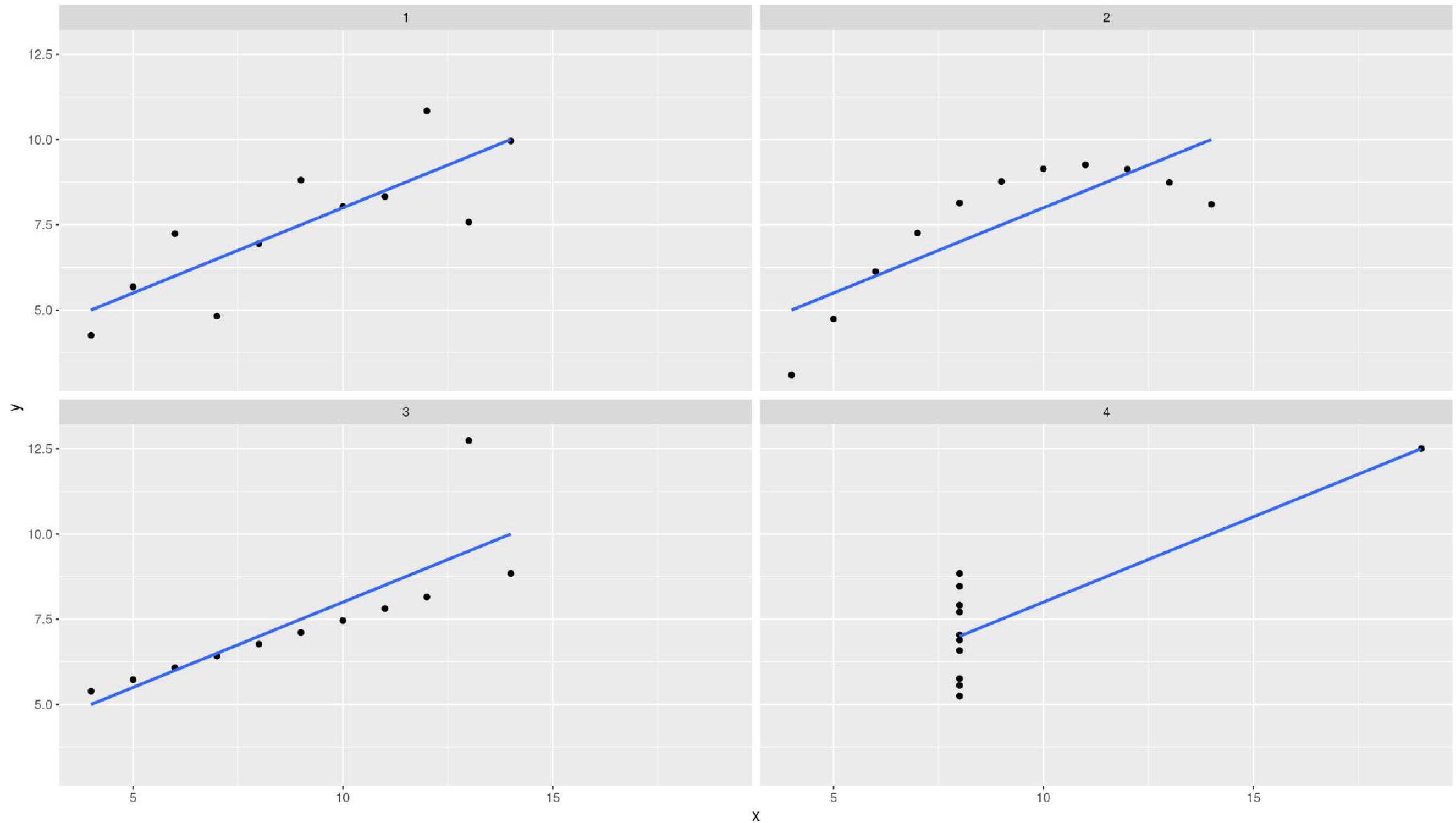
	data_id	x	y
1	1	10	8.04
2	1	8	6.95
12	2	10	9.14
13	2	8	8.14
24	3	8	6.77
25	3	13	12.74
36	4	8	7.71
37	4	8	8.84

All these four data sets have: same mean and SD in  $x$  and  $y$ , and same correlation between them:

	data_id	mean_x	sd_x	mean_y	sd_y	cor
1	1	9	3.317	7.501	2.032	0.816
2	2	9	3.317	7.501	2.032	0.816
3	3	9	3.317	7.500	2.030	0.816
4	4	9	3.317	7.501	2.031	0.817

So, seems like these data sets should look pretty similar, right?

# Why even look at data?



# Why even look at data?

- While this is an extreme, manufactured example, it does prove a point: visualizing data offers insight that the study of pure statistics does not.
- But that does not mean looking at data is all one needs to do:
  - Real data sets are often complicated and messy, displaying them graphically presents problems of its own.
  - The core problem of visualization is how to **map** relations between observations to visual representations in the plot, like shape, lines, colour, size, etc.
  - How to do this optimally for each given data set is subject to considerable debate. There is no simple recipe to follow.
- Fortunately, however, there is software that provides a lot of support in our data visualization endeavours.

# Data visualization in R

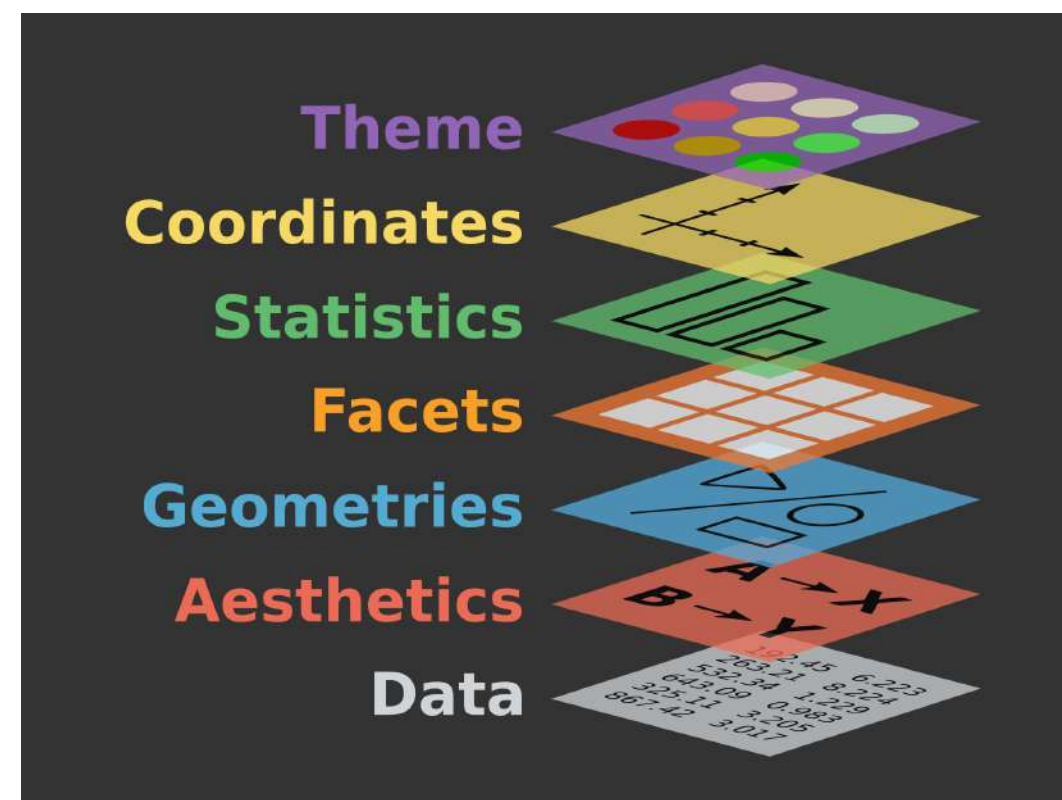
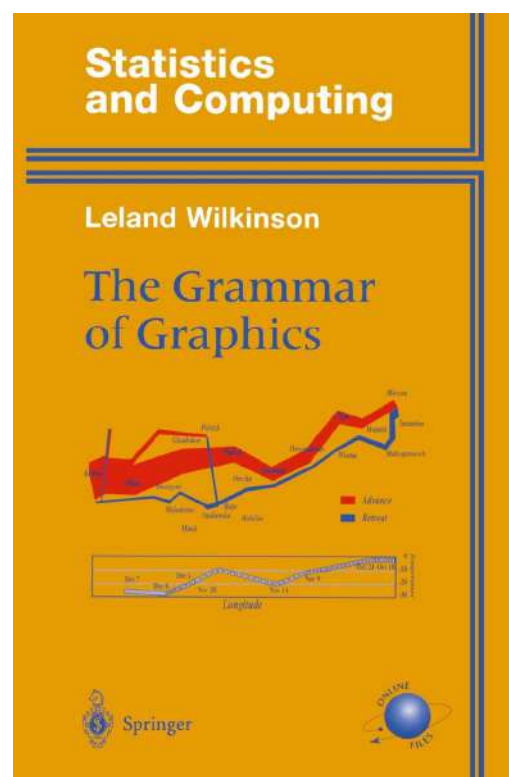
“The simple graph has brought more information to the data analyst’s mind than any other device.”

John Tukey (Statistician, 1915 - 2000)

- There are many tools for visualizing data. Of course, we use R 😊
- Even with R, there are many approaches / systems / packages for creating data visualizations. By far the most common one is the package `ggplot2`.
- The `gg` in `ggplot2` stands for the **grammar of graphics**, which is a coherent system for describing and building graphs developed by statistician Leland Wilkinson in his aptly named 2005 book.
- As we will see in great detail, this system constructs plots in layers that are built on top of one another in an additive fashion.

# The grammar of graphics

The grammar is a set of rules for producing graphics from data, taking pieces of data and mapping them to **geometric objects** (like points and lines) that have **aesthetic attributes** (like position, colour and size), together with further rules for transforming the data if needed, adjusting scales, adapting coordinate system and themes.



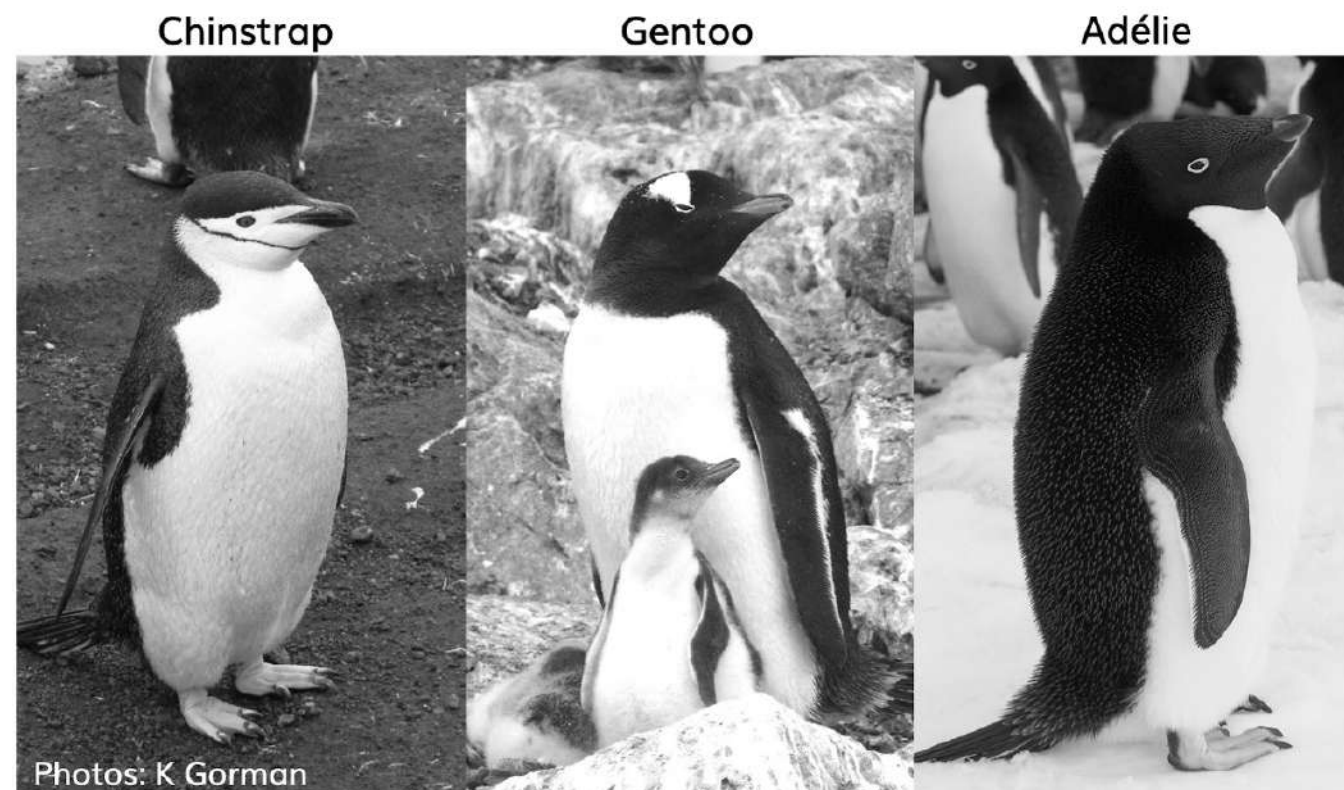
**Note:** a grammar limits the structure of what you can say, but it does not automatically make what you say meaningful, i.e. your code just being “grammatically” correct does not make the resulting plot sensible.



# An introductory example – Palmer penguins

Let's see an example of how this works with `ggplot2` in practice.

- The data set we will use for our first `ggplot` is the `palmerpenguins` data set that can be obtained from CRAN in a package that bears exactly that name.
- It 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.





# An introductory example – Palmer penguins

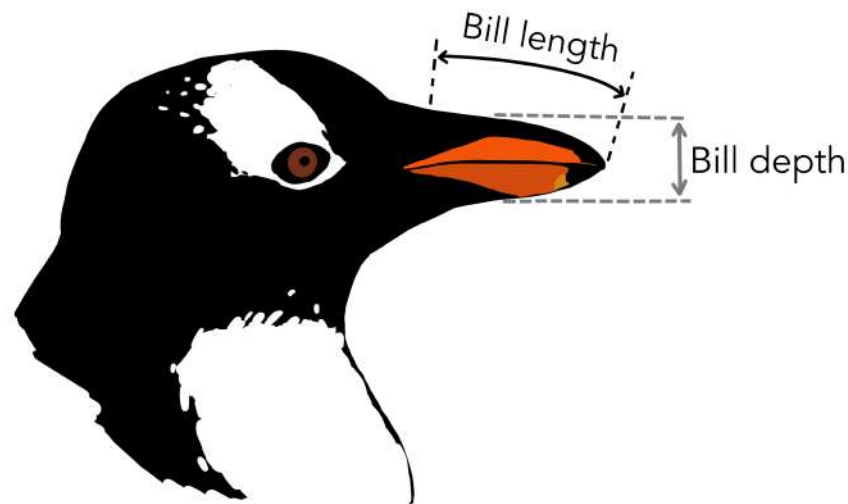
Make sure the required packages `palmerpenguins` and `ggplot2` are installed. Then we can load them and start by inspecting the data set:

```
1 library(palmerpenguins)
2 library(ggplot2)
3
4 penguins <- as.data.frame(penguins)
5 head(penguins)
```

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g
1	Adelie	Torgersen	39.1	18.7	181	3750
2	Adelie	Torgersen	39.5	17.4	186	3800
3	Adelie	Torgersen	40.3	18.0	195	3250
4	Adelie	Torgersen	NA	NA	NA	NA
5	Adelie	Torgersen	36.7	19.3	193	3450
6	Adelie	Torgersen	39.3	20.6	190	3650

	sex	year
1	male	2007
2	female	2007
3	female	2007
4	<NA>	2007
5	female	2007
6	male	2007



Artwork by  
[@allison\\_horst](#)

# An introductory example – Palmer penguins

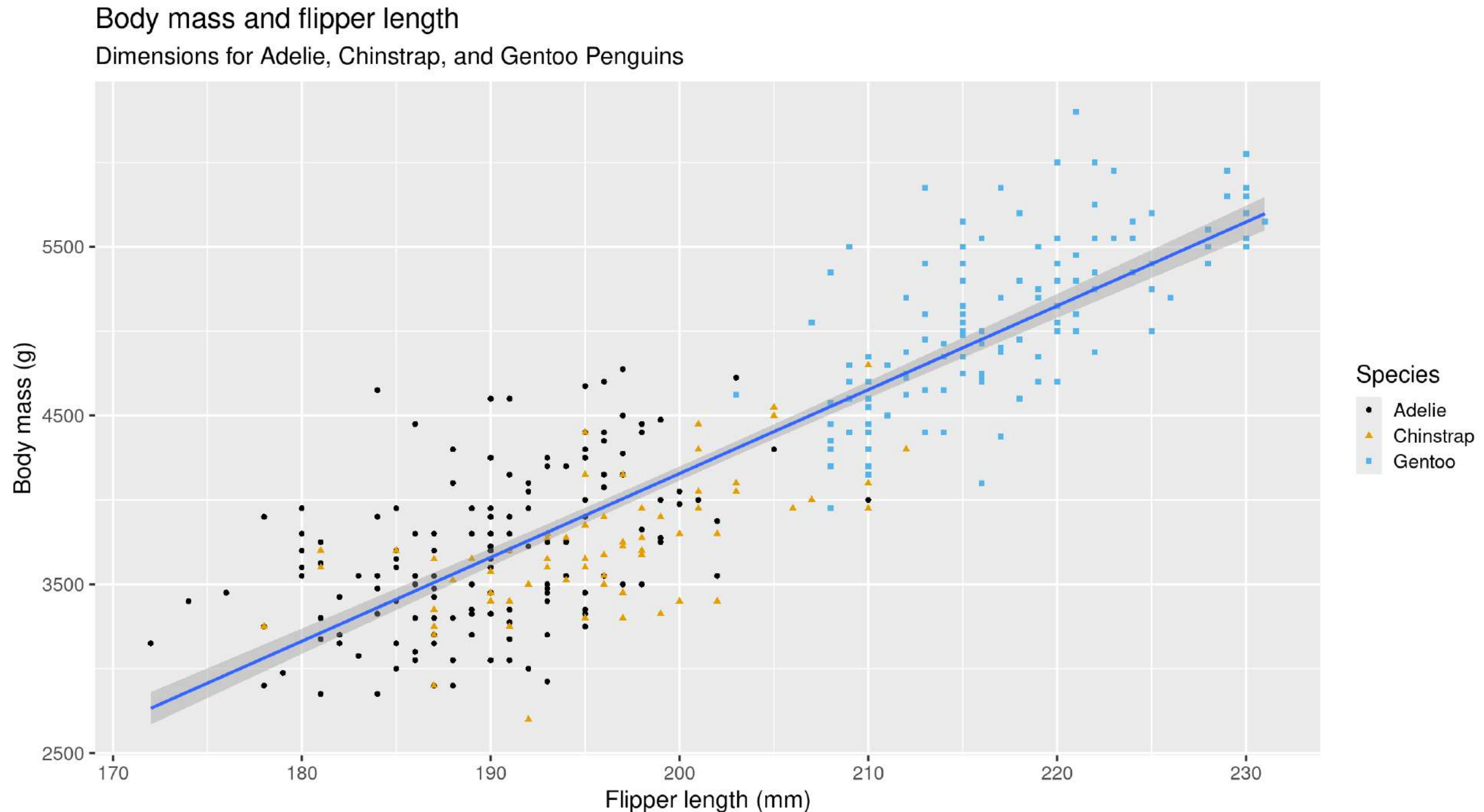
As zoologists, we might be interested in the following questions about the different types of Palmer penguins:

- Do penguins with longer flippers weigh more or less than penguins with shorter flippers?
- What does the relationship between **flipper length** and **body mass** look like? Is it positive? Negative? Linear? Non-linear?
- Does the relationship vary by the species of the penguin? How about by the island where the penguin lives?

Let's answer all of these questions using a single visualization.

# An introductory example – Ultimate goal

Ultimately, we want to create the following plot:

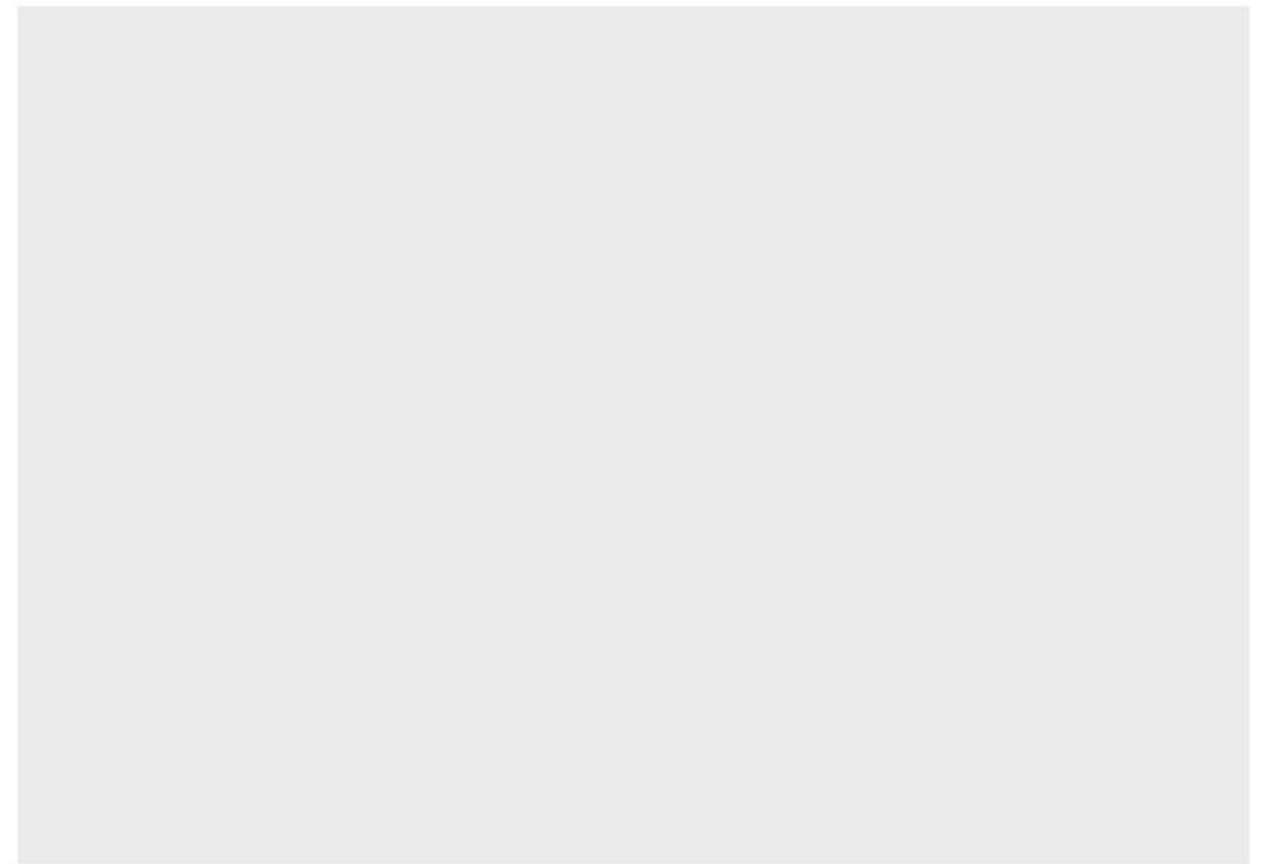


# First step – Telling ggplot about our data set

Let's recreate this plot step-by-step.

With `ggplot2`, we begin a plot with the function `ggplot`. Its first argument is the dataset to use in the graph.

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

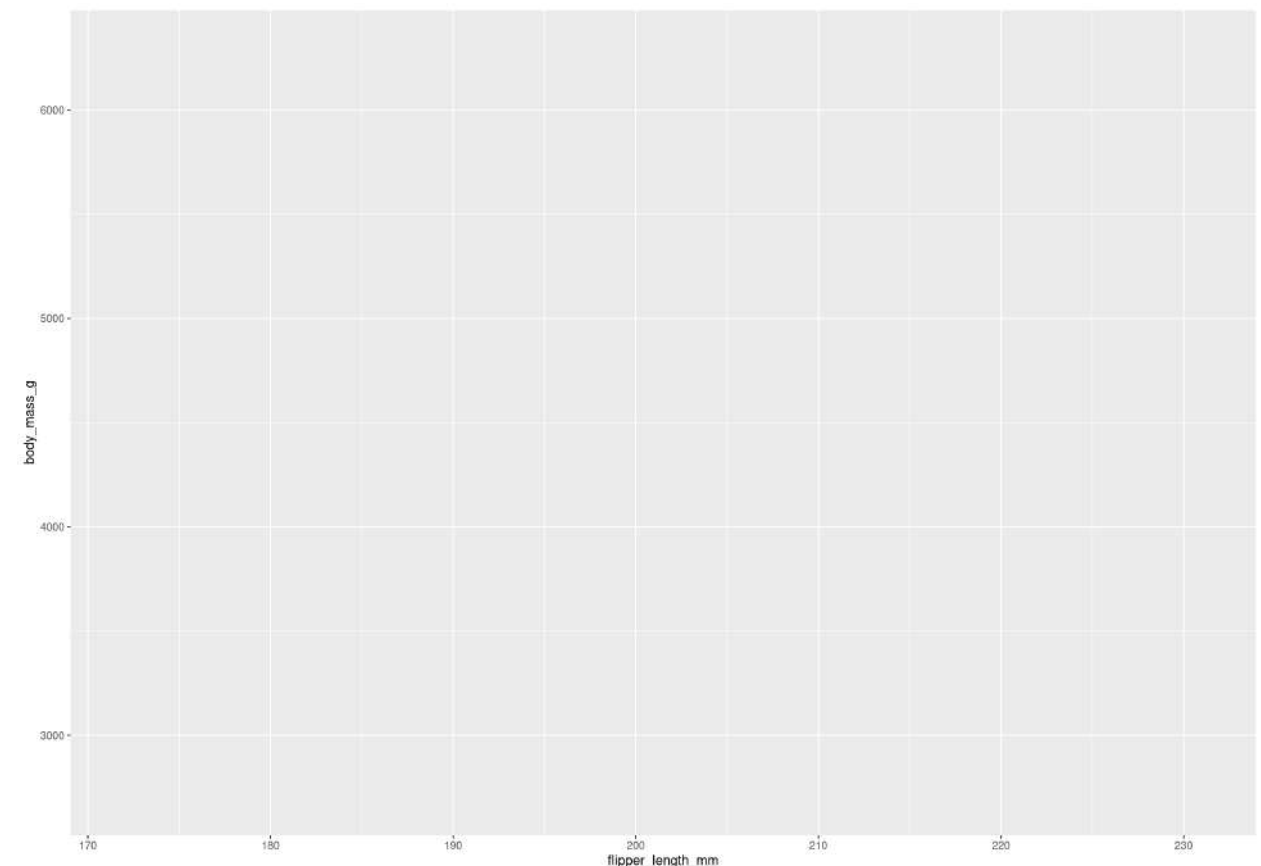


For now, we have not told `ggplot` how to visualize the data, so we only have an empty canvas. Now, we will “paint” onto this canvas in layers.

# Second step – Specifying aesthetic mappings

The second argument of `ggplot` is called `mapping`. It defines how variables in our data set are **mapped** to visual properties (**aesthetics**) of our plot. This argument is always defined in the `aes` function, and the `x` and `y` arguments of `aes` specify which variables to map to the x- and y-axes. We want flipper length on the `x` and body mass on the `y` axis.

```
1 ggplot(data = penguins,  
2       mapping = aes(x = flipper_length_mm,  
3                     y = body_mass_g))
```



Now, our empty canvas has more structure: x- and y-axis have a range, ticks and labels. But the penguins themselves are not yet on the plot.

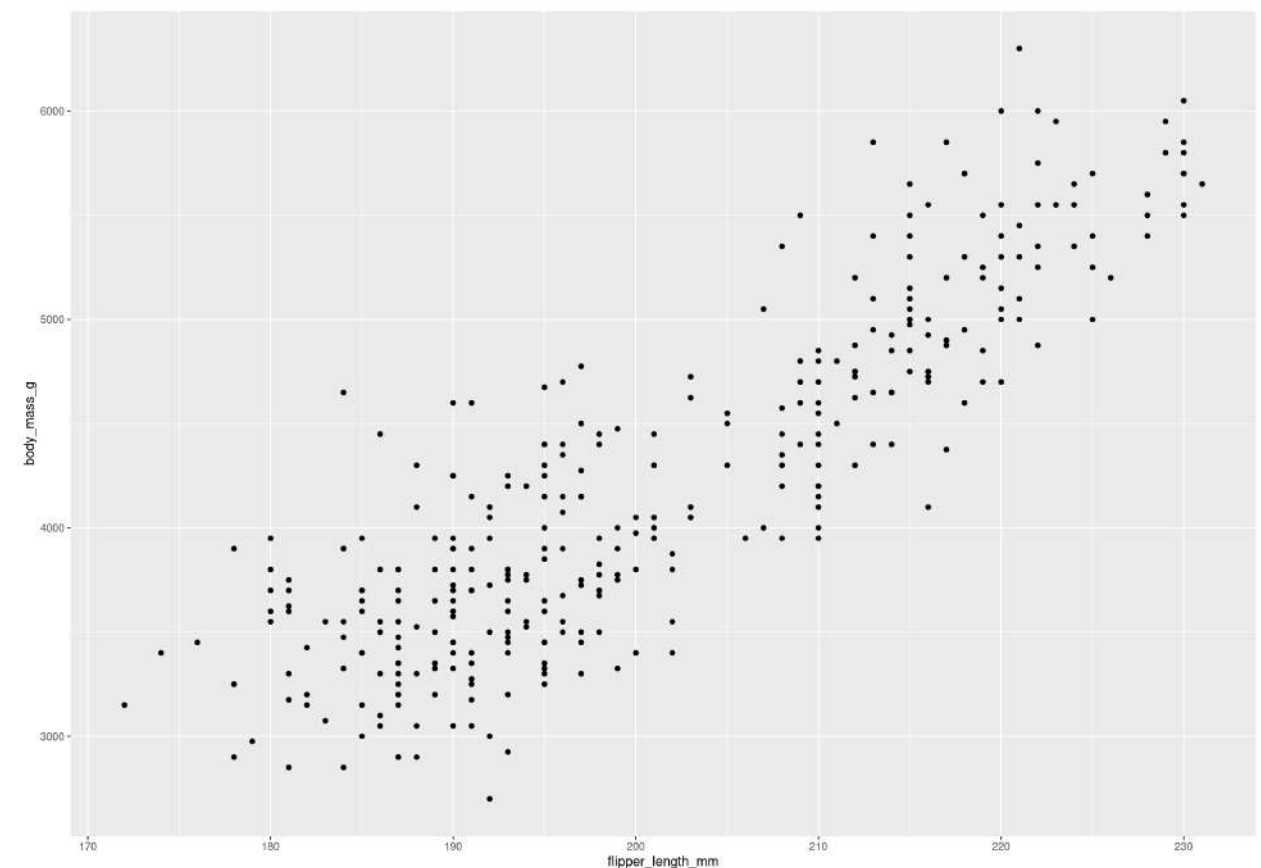


# Third step – Additively layer on geoms

This is because we have not yet articulated, in our code, **how** to represent the observations from our data frame on our plot. To do so, we need a **geom**, a geometrical object used for data representation. In the example, we want our data represented by points, so we use `geom_point()`:

```
1 ggplot(data = penguins,  
2       mapping = aes(x = flipper_length_mm,  
3                     y = body_mass_g)) +  
4   geom_point()
```

Warning: Removed 2 rows containing missing values or values outside the scale range (``geom_point()``).



Now, we start to have an actual **scatter plot**. Note that R warns us about missing data points. We will suppress this warning in the plots to come.



# A note about geoms

In **ggplot2**, geometric objects are made available with functions that start with **geom\_**. People often describe plots by the type of geom that the plot uses, for example:

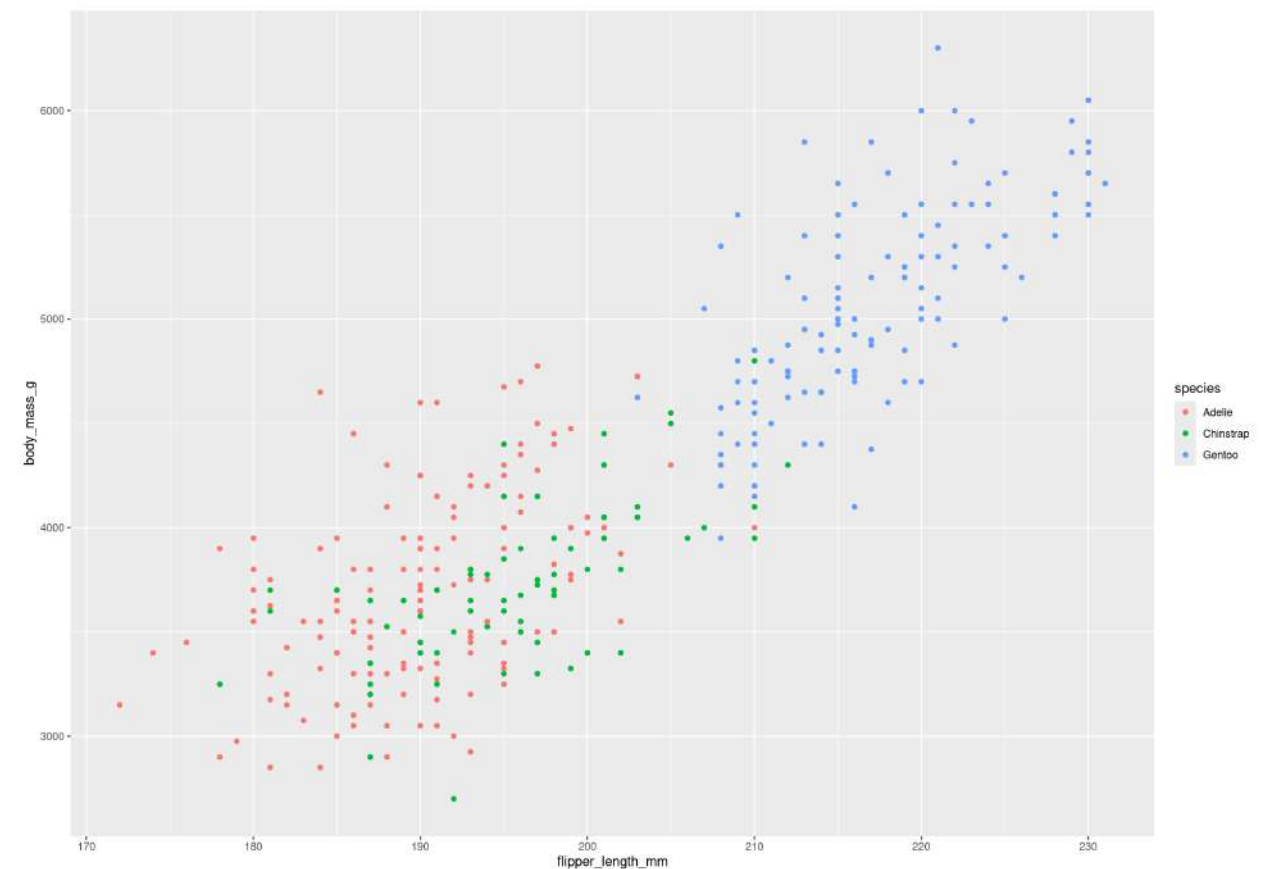
- Scatter plots use point geoms (**geom\_point**) as we just saw.
- Bar charts use bar geoms (**geom\_bar**)
- Line charts use line geoms (**geom\_line**)
- Boxplots use boxplot geoms (**geom\_boxplot**)
- ...

We layer a **geom** onto a ggplot literally in an **additive way**, i.e. by combining the two with a **+** sign. We will learn how to deal with many different geoms in this way.

# Fourth step – Adding aesthetics and layers

Does the relationship between flipper length and body mass differ by species? To answer this, let's represent species with different coloured points. To achieve this, we modify the **aesthetic** to additionally map species to colour:

```
1 ggplot(data = penguins,  
2       mapping = aes(x = flipper_length_mm,  
3                     y = body_mass_g,  
4                     color = species)) +  
5   geom_point()
```

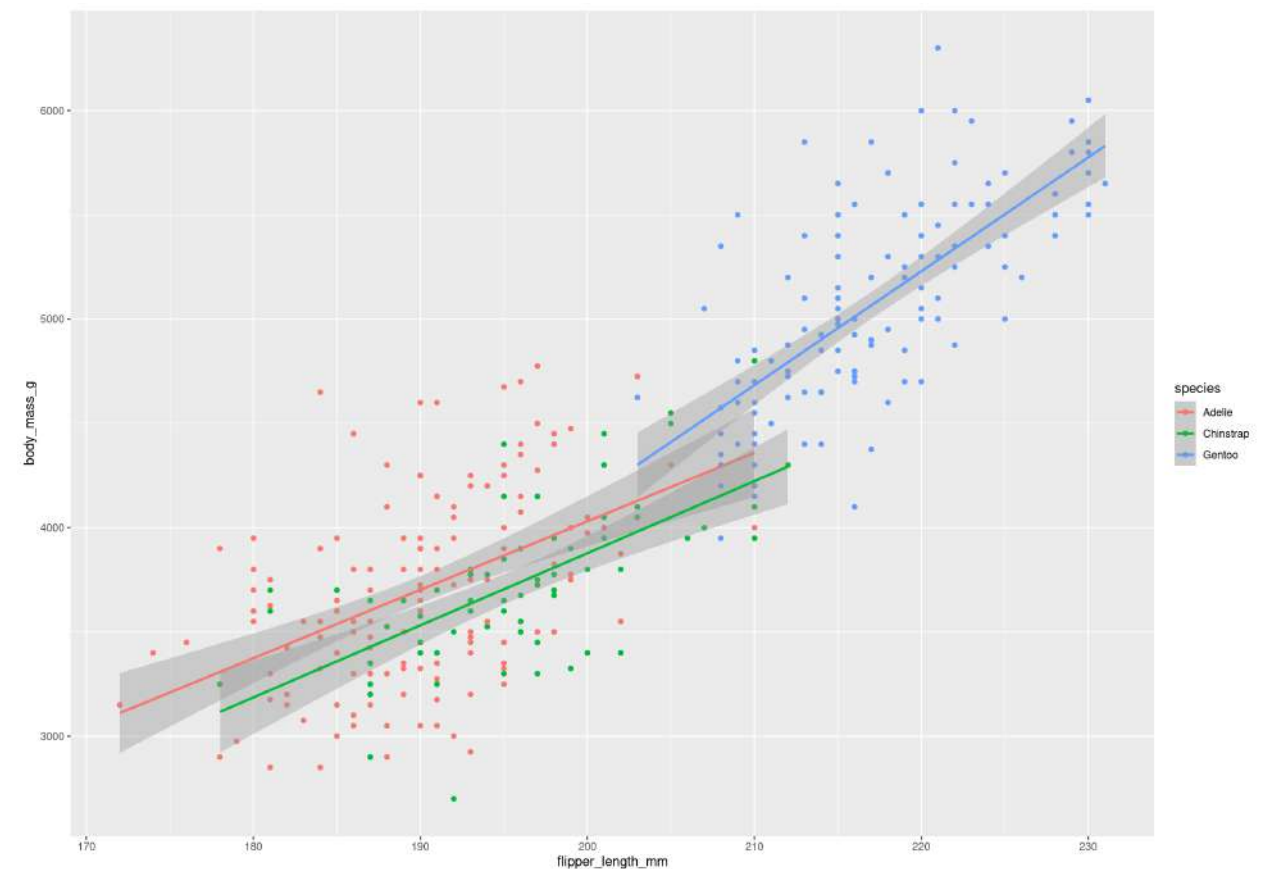


In mapping a categorical variable to the colour aesthetic, **ggplot2** automatically assigns a unique colour to each factor level (i.e. each species), a process known as **scaling**. **ggplot2** will also add a legend that explains which values correspond to which levels.

# Fourth step – Adding aesthetics and layers

Now, let's add one more layer, a smooth curve displaying the relationship between body mass and flipper length. Since this is a new geometric object representing our data, we will add a new **geom**, namely **geom\_smooth** based on a linear model (**lm**):

```
1 ggplot(data = penguins,  
2       mapping = aes(x = flipper_length_mm,  
3                     y = body_mass_g,  
4                     color = species)) +  
5   geom_point() +  
6   geom_smooth(method = "lm")
```

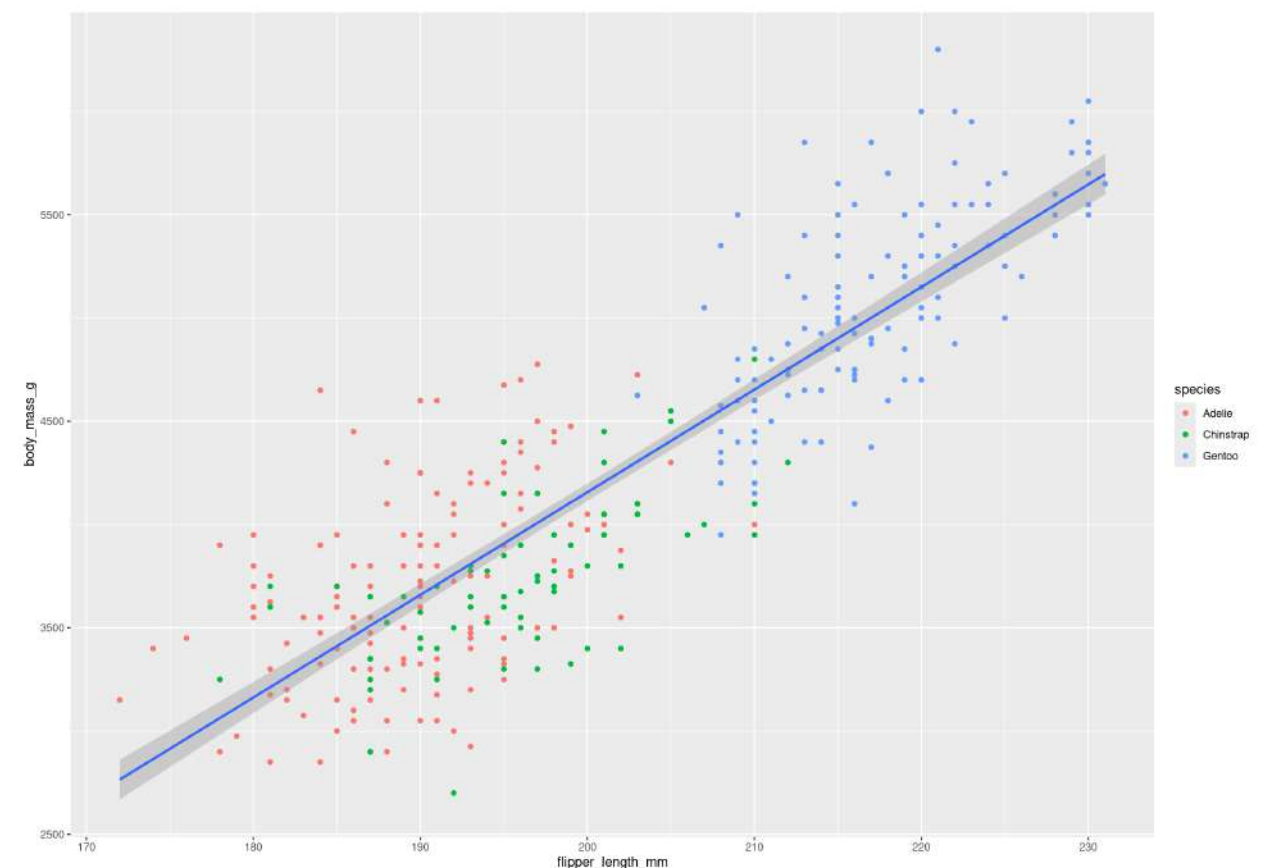


We have successfully added lines, but this plot does not look like our ultimate goal. Instead of having only one line for the entire data set, we have separate lines for each of the three penguin species. What went wrong?

# Fourth step – Adding aesthetics and layers

When aesthetics are defined in `ggplot()`, at the top level, they're passed down to each of the subsequent **geom** layers of the plot. However, each geom function can also take a mapping argument, which allows for aesthetics at the local level. Here, we want point colours, but not lines separated by species, so we should specify `color = species` locally for `geom_point` only:

```
1 ggplot(data = penguins,
2       mapping = aes(x = flipper_length_mm,
3                     y = body_mass_g)) +
4   geom_point(mapping = aes(color = species))
5   geom_smooth(method = "lm")
```

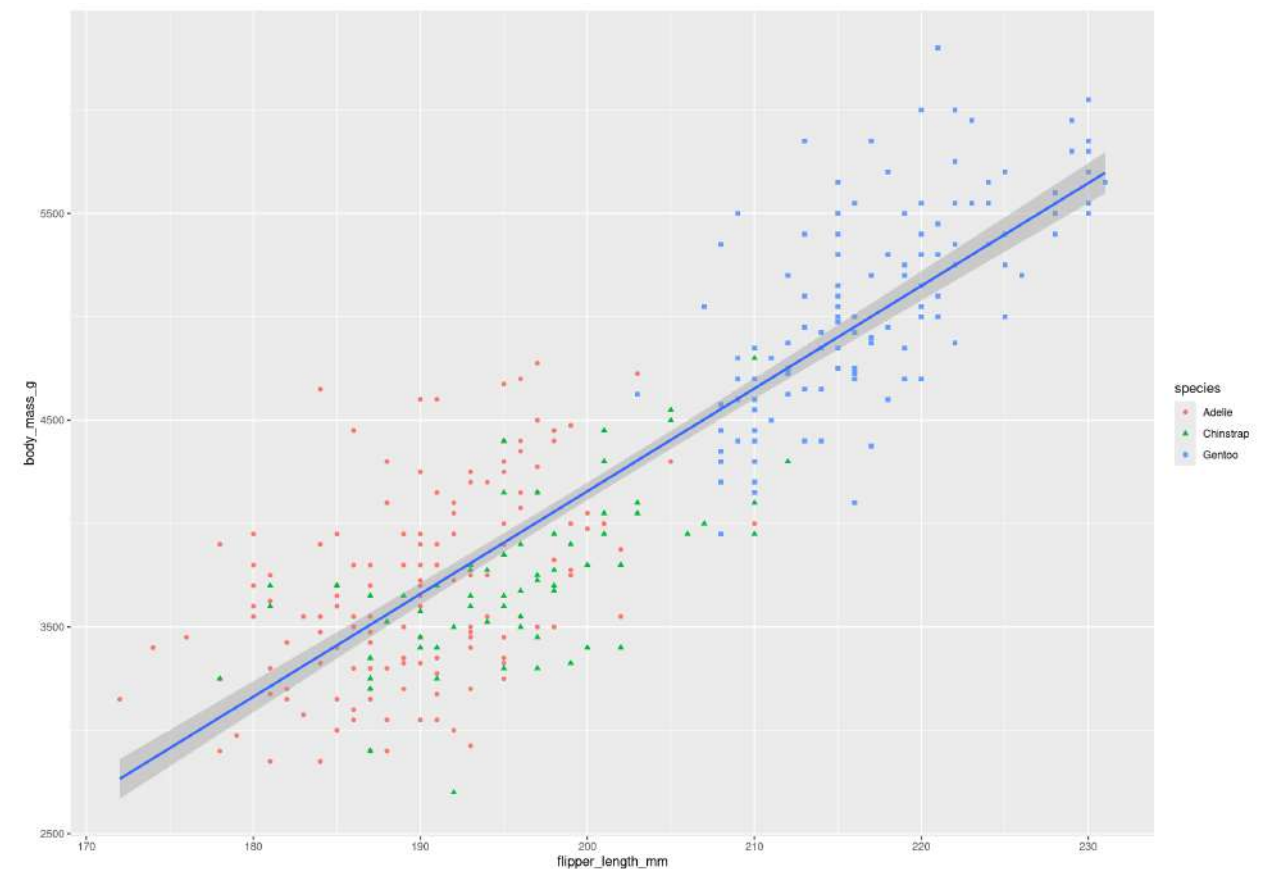


We are getting close to our ultimate goal... Some minor details are still missing though.

# Fourth step – Adding aesthetics and layers

Due to differences in colour perception (e.g. colour blindness), it is generally not a good idea to represent information using only colours on a plot. Therefore, in addition to colour, we can also map **species** to the **shape** aesthetic:

```
1 ggplot(data = penguins,  
2       mapping = aes(x = flipper_length_mm,  
3                     y = body_mass_g)) +  
4   geom_point(mapping = aes(color = species,  
5                           shape = species))  
6   geom_smooth(method = "lm")
```



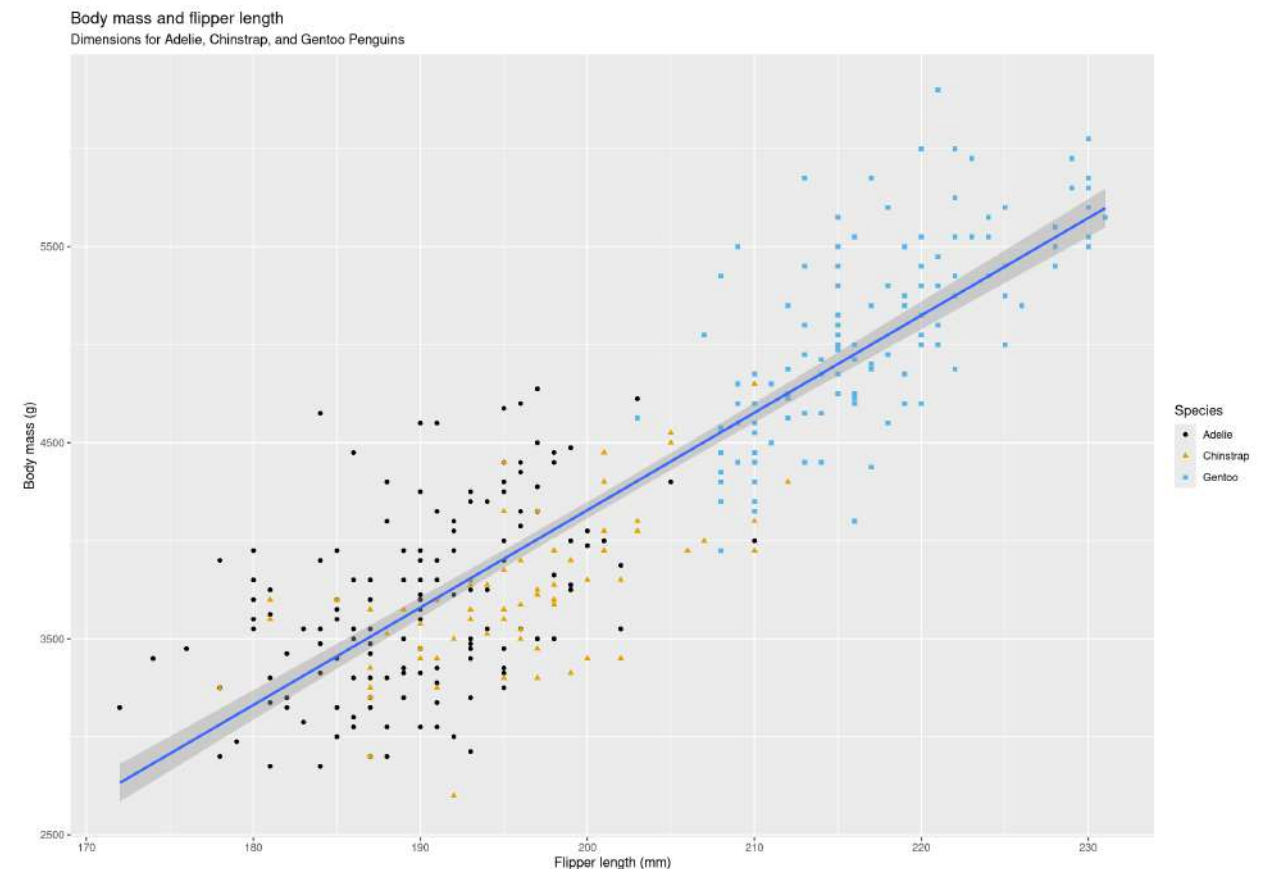
Note that the legend is automatically updated to reflect the different shapes of the points as well.



# Fifth step – Adjust scales, labels, titles, ...

The “clean-up” work involves setting appropriate labels for title, subtitle, axes, and legend using the `labs` function and using the colourblind safe colour palette `scale_color_colorblind` from the `ggthemes` package (an extension of `ggplot2`):

```
1 library(ggthemes)
2
3 ggplot(data = penguins,
4       mapping = aes(x = flipper_length_mm,
5                     y = body_mass_g)) +
6   geom_point(mapping = aes(color = species,
7                             shape = species))
8   geom_smooth(method = "lm") +
9   labs(x = "Flipper length (mm)",
10        y = "Body mass (g)",
11        title = "Body mass and flipper length",
12        subtitle = "Dimensions for Adelie, Chi",
13        color = "Species",
14        shape = "Species") +
15   scale_color_colorblind()
```



Now, we are done, we have built the plot of our ultimate goal step-by-step from the ground up. The example highlights the core principles of how `ggplot2` allows us to build nice-looking informative graphs with relatively little coding effort.

# Basic types of plots with ggplot2

# Steps in creating a plot with `ggplot2`

Conceptually, the steps we went through are the same for every plot created with `ggplot2`. Based on a **tidy** data set, we always proceed as follows:

- Telling `ggplot` about our data set.
- Specifying **aesthetic mappings**, i.e. what relationships we want to see and by what we want them represented.
- Additively layer on **geoms** as needed.
- Add and adapt aesthetics and layers until all relationships are displayed.
- Adjust scales, labels, titles, ...

These steps are always the same. We will now only learn in greater detail about how to tell `ggplot` what to do by going through several examples of different data sets, aesthetic mappings, geoms and more. As with anything, the key to improvement is **LOTS** of trial and error...



# Visualizing a numerical variable – Histogram

Let's say we want to visualize the distribution of body mass (in grams) across all penguins in the sample using a **histogram**. For this, we map the variable `body_mass_g` to `x` and use `geom_histogram`:

Code

Plot

```
1 ggplot(penguins, aes(x = body_mass_g)) +  
2   geom_histogram()
```

# Visualizing a numerical variable – Histogram

A histogram divides the x-axis into equally spaced bins and then uses the height of a bar to display the number of observations that fall in each bin. `ggplot2` by default creates 30 such bins. We can control this by setting `bins` ourselves:

Code

Plot

```
1 ggplot(penguins, aes(x = body_mass_g)) +  
2   geom_histogram(bins = 15)
```

# Visualizing a numerical variable – Histogram

We do not like the colour of this histogram. Previously, we set the `color` aesthetic, but here we do not want to **map** colour to a variable. Instead we want to **set** it to a specific colour. This we do in the **geom**, not in the `aes`:

Code

Plot

```
1 ggplot(penguins, aes(x = body_mass_g)) +  
2   geom_histogram(color = "deepskyblue4", bins = 15)
```

# Visualizing a numerical variable – Histogram

Hm, this is not what we wanted... `ggplot` only coloured the border line of the bars. Turns out that if we want to **fill** the bars in a specific colour, we need to use the `fill` argument of the `geom_histogram` function:

Code

Plot

```
1 ggplot(penguins, aes(x = body_mass_g)) +  
2   geom_histogram(fill = "deepskyblue4", bins = 15)
```

# Visualizing a numerical variable – Density plot

An alternative visualization for distributions of numerical variables is a **density plot**, which is a smoothed-out version of a histogram. To create one, we simply switch out our **geom** to `geom_density`:

Code

Plot

```
1 ggplot(penguins, aes(x = body_mass_g)) +  
2   geom_density()
```

# Visualizing a numerical variable – Density plot

That's a bit boring, let's change the colour again (both of the line itself using `color` and the area below the curve using `fill`):

Code

Plot

```
1 ggplot(penguins, aes(x = body_mass_g)) +  
2   geom_density(color = "deepskyblue4", fill = "deepskyblue4")
```

# Visualizing a categorical variable – Barplot

A simple **bar chart** visualizes the (absolute or relative) frequencies of the levels of a categorical variable. While we could use `ggplot` directly on the data, it is generally preferable to compute frequencies explicitly ourselves, e.g. for species:

Code

Plot

```
1 species_freq <- as.data.frame(table(penguins$species))
2 names(species_freq)[1] <- "species"
3
4 # As we have already computed frequencies, we have to tell the bar geom
5 # that it should use the values as they are, i.e. use the "identity":
6 ggplot(species_freq, aes(x = species, y = Freq)) +
7   geom_bar(stat = "identity", fill = "deepskyblue4")
```

# Visualizing a categorical variable – Barplot

If the variable has a nominal (and not ordinal) scale, then it is usually preferable to order the bars based on their frequency. For this, we have to **reorder** the factor levels for the species:

Code

Plot

```
1 ggplot(species_freq, aes(x = reorder(species, Freq, decreasing = TRUE), y = Freq)) +  
2   geom_bar(stat = "identity", fill = "deepskyblue4")
```



# Visualizing a categorical variable – Barplot

Often, we want the bars to be horizontal rather than vertical. To achieve this, we simply flip the axes using `coord_flip`. Additionally, we add nice axis labels:

Code

Plot

```
1 ggplot(species_freq, aes(x = reorder(species, Freq, decreasing = TRUE), y = Freq)) +  
2   geom_bar(stat = "identity", fill = "deepskyblue4") +  
3   labs(x = "Penguin Species", y = "Observed frequency") +  
4   coord_flip()
```

# A categorical and a numerical variable

To visualize the relationship between a categorical and a numerical variable, we can create **parallel boxplots**. For example, for the distribution of body weight by species, we could try:

Code

Plot

```
1 ggplot(penguins, aes(x = species, y = body_mass_g, fill = species)) +  
2   geom_boxplot()
```

# A categorical and a numerical variable

An alternative to boxplots are so-called **violin plots**. In their simplest form, these are rotated density plots of the numerical variable in each category mirrored to create a symmetric, “violin-like” object:

Code

Plot

```
1 ggplot(penguins, aes(x = species, y = body_mass_g, fill = species)) +  
2   geom_violin()
```

# A categorical and a numerical variable

Since density plots can also mask certain aspects of the distribution of the data, it is often advisable to additionally plot the actual data points on top of the violin plots. We can do this by layering a **point geom** on top:

Code

Plot

```
1 ggplot(penguins, aes(x = species, y = body_mass_g, color = species)) +  
2   geom_violin() +  
3   geom_point()
```

# A categorical and a numerical variable

If we have a lot of points close together, then they overlap and it becomes impossible to tell their distribution. To avoid this, we can slightly perturb the points in the `x` direction. This is achieved by switching to `geom_jitter`.

Code

Plot

```
1 ggplot(penguins, aes(x = species, y = body_mass_g, color = species)) +  
2   geom_violin() +  
3   geom_jitter()
```

# A categorical and a numerical variable

Another alternative to displaying the distribution of body weights in the three penguin species is to combine their densities into one plot in the usual way, i.e. with the numerical variable on the **x** axis.

Code

Plot

```
1 ggplot(penguins, aes(x = body_mass_g, fill = species)) +  
2   geom_density()
```

# A categorical and a numerical variable

Now, the densities are overlapping. To avoid this, we could of course not map the `fill` aesthetic. A nice alternative is to set the `alpha` aesthetic, which controls **transparency**. 0 is max transparency, 1 is max opaqueness (default).

Code

Plot

```
1 ggplot(penguins, aes(x = body_mass_g, fill = species)) +  
2   geom_density(alpha = 0.5)
```

# Two categorical variables

We can use **grouped or stacked bar plots** to visualize the relationship between two categorical variables. For example, we might want to see how often the different species occur on the different islands.

Code

Plot

```
1 species_freq <- as.data.frame(table(penguins$species, penguins$island))
2 names(species_freq)[1:2] <- c("species", "island")
3
4 ggplot(species_freq, aes(x = island, y = Freq, fill = species)) +
5   geom_bar(stat = "identity")
```



# Two categorical variables

We can create variations of this plot by changing the `position` argument of `geom_bar`. The default is `stack`. Most useful for comparing distributions are relative frequency plots achieved with `position = "fill"`:

Code

Plot

```
1 ggplot(species_freq, aes(x = island, y = Freq, fill = species)) +  
2   geom_bar(stat = "identity", position = "fill")
```

# Two numerical variables

In the introductory example, we have already seen **scatter plots** and **smooth curves** as ways to illustrate the relationship between two numerical variables. Scatter plots are an absolute staple in a data scientist's toolbox:

Code

Plot

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

# Three or more variables

The introductory example displayed three variables: `flipper_length_mm` (x), `body_mass_g` (y) and `species` (`color` and `shape`). In general, we have the following options to introduce additional variables into a scatter plot:

- Use additional **aesthetics**:
  - Colour → `color` (categorical or numerical variables)
  - Shape → `shape` (categorical)
  - Size → `size` (categorical or numerical variables)
  - Transparency → `alpha` (categorical or numerical variables)
- Split plot into **facets**, subplots that each display one subset of the data (categorical variables).
- Animate the plot (categorical or numerical variables, especially time).
- Create a 3D scatter plot (categorical or numerical variables).

# Three or more variables

- When using aesthetics to represent additional variables, we must be aware that different sorts of variables can be represented more or less well by different kinds of visual representations.
- While there are no one-size-fits-all recipes, consider the following order as a heuristic when introducing additional variables into a 2D scatter plot:
  - For **continuous variables**: size, colour, transparency.
  - For **categorical variables**: colour, shape.
- For example, in our scatter plot of penguin flipper lengths vs. body masses, we might want to introduce categorical information on **species** and **island**. By the heuristic, we use colour for the first and shape for the second.

# Three or more variables – Colour and shape

In the default setting, the points are quite small, which makes it hard to distinguish the shapes. To mitigate this, we additionally set the `size` aesthetic to something larger, namely 3.

Code

Plot

```
1 ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g,  
2                       color = species, shape = island)) +  
3   geom_point(size = 3)
```

# Three or more variables – Colour and size

Let's say we wanted a scatter plot of penguin bill length vs. bill depth for different species. Additionally, we want information on body mass included as well. A logical choice to represent the latter would be **size**:

Code

Plot

```
1 ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,  
2                       color = species, size = body_mass_g)) +  
3   geom_point()
```

# Three or more variables – Multiple aesthetics

If we now really wanted to go mad, we could add the island back in via the **shape** aesthetic and additionally include the flipper length via the **alpha** aesthetic, so that longer flippers are represented by more opaque points:

Code

Plot

```
1 ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,  
2                       color = species, size = body_mass_g,  
3                       shape = island, alpha = flipper_length_mm)) +  
4   geom_point()
```

# Three or more variables – Faceting

- Clearly, there is now **too much information** in this plot.
- Adding too many aesthetic mappings to a plot makes it **cluttered** and difficult to make sense of.
- Another way to introduce additional categorical variables is to split your plot into several **facets**.
- A facet is a **subplot** that displays one subset of the data. For example, there could be one facet for each factor level of a categorical variable.
- We can use facets to explore **conditional relationships**, like the relationship between flipper length and body weight on each of the three islands.



# Three or more variables – Faceting

To facet a plot by a single variable, we use `facet_wrap`. It takes a so-called **formula** as its first argument, which we create with `~` followed by the name of a categorical variable.

Code

Plot

```
1 ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g,  
2                       color = species)) +  
3   geom_point() +  
4   facet_wrap(~island)
```

# Three or more variables – Faceting

To facet a plot by two variables, we use `facet_grid`. We specify the variable to be faceted in the **rows** before the `~` and the variable to be faceted in the **columns** after it. For example, we could additionally facet `sex` in the rows:

Code

Plot

```
1 ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g,  
2                       color = species)) +  
3   geom_point() +  
4   facet_grid(sex~island)
```

# Three or more variables – Labels

- For some data sets, we can also display an additional variable in a scatter plot by creating informative **labels** for the points.
- Our penguin data set is not a good example to illustrate this, so we consider the `elections_historic` data set from the `socviz` package instead:

```
1 library(socviz)
2 elections_historic <- as.data.frame(elections_historic)
3 head(elections_historic[, c("year", "winner", "win_party", "ec_pct",
4                             "popular_pct", "winner_label")])
```

	year	winner	win_party	ec_pct	popular_pct	winner_label
1	1824	John Quincy Adams	D.-R.	0.3218	0.3092	Adams 1824
2	1828	Andrew Jackson	Dem.	0.6820	0.5593	Jackson 1828
3	1832	Andrew Jackson	Dem.	0.7657	0.5474	Jackson 1832
4	1836	Martin Van Buren	Dem.	0.5782	0.5079	Buren 1836
5	1840	William Henry Harrison	Whig	0.7959	0.5287	Harrison 1840
6	1844	James Polk	Dem.	0.6182	0.4954	Polk 1844

- This data set contains information on US presidential elections from 1824 to 2016. We will have a look at the winner's share of the electoral college vote and the popular vote.

# Three or more variables – Labels

In a scatter plot, we want to have the popular vote share on the **x** axis and the electoral college vote share on the **y** axis. If we only map these two aesthetics, the plot looks quite uninformative...

Code

Plot

```
1 ggplot(elections_historic, aes(x = popular_pct, y = ec_pct)) +  
2   geom_point()
```

# Three or more variables – Labels

We do not know who any of these points represent. This is a perfect use case for **labeling** the points. For this, we can use the `label` aesthetic together with the `text` geom (there is also a `label` geom, which would work as well):

Code

Plot

```
1 ggplot(elections_historic, aes(x = popular_pct, y = ec_pct, label = winner_label)) +  
2   geom_text() +  
3   geom_point()
```

# Three or more variables – Labels

That is much better already. However, now, there is a lot of overlap. To remedy this, we need an additional library called `ggrepel`, which provides us with a `text_repel` geom to avoid overlapping labels:

Code

Plot

```
1 library(ggrepel)
2 ggplot(elections_historic, aes(x = popular_pct, y = ec_pct, label = winner_label)) +
3   geom_text_repel() +
4   geom_point()
```

# Customizing our plots

- So far, we have seen how to create and extend basic plots, like bar plots, density plots, histograms, and scatter plots by tailoring aesthetic mappings and layering on suitable **geoms** according to our needs.
- However, the customization usually does not stop there. We might want to ...
  - ... add informative titles, subtitles, captions and axis labels to the plot.
  - ... use colours and shapes different from the ones **ggplot** uses by default.
  - ... adapt the scale of the **x** and / or **y** axis.
  - ... change the font type and size for labels and titles.
  - ... set the whole plot in a different theme to suit corporate guidelines.
  - ... and so much more...
- In **ggplot2**, virtually every aspect of a plot is **customizable**. We will only have a look at how to customize some of the most important ones.

# Getting the details right – Scales, guides and themes



# Understanding scales, guides and themes

- Outside of aesthetic mappings and geoms, there are three types of functions that do most of the heavy lifting when it comes to customizing our plot:
  - The family of `scale_` functions
  - The `guides` function and
  - The `theme` function.
- Due to the sheer amount of options for customizability and the partial overlap between some of these functions, things can become confusing very quickly when working through these functions.
- Let's start with a basic delineation of scales, guides and themes.

# Understanding scales, guides and themes

- A rough guide on the use of these functions is the following:
  - **Scales:** every aesthetic mapping has a scale. If we want to adjust how that scale is marked or graduated, we use a `scale_` function.
  - **Guides:** Many scales come with a legend to help us interpret the graph. These are called guides and can be adjusted with the help of the `guides` function.
  - **Themes:** graphs have other features not strictly connected to the structure of the data being displayed. These include background colour, font size, legend placement, etc. To adjust these, we use the `theme` function.
- We will now dig deeper into each of these three domains of customization and see examples of how the corresponding functions are used to change the appearance of the data displayed.

# Understanding scales

Let's come back to an earlier example from the penguin data set:

Code

Plot

```
1 ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g,  
2                       color = species, shape = island)) +  
3   geom_point(size = 3)
```

# Understanding scales

This plot has four aesthetic mappings:

- The variable `flipper_length_mm` is mapped to `x`.
- The variable `body_mass_g` is mapped to `y`.
- The variable `species` is mapped to `color`.
- The variable `island` is mapped to `shape`.

And each of these mappings has a **scale**:

- `flipper_length_mm` is a **continuous** variable, so the `x` scale is continuous.
- `body_mass_g` is a **continuous** variable, so the `y` scale is continuous.
- `species` is an unordered **categorical** variable, so the `color` scale is discrete.
- `island` is an unordered **categorical** variable, so the `shape` scale is discrete.

# Understanding scales

- Scales for these mappings may have labels, axis tick marks at particular positions, or specific colours or shapes. If we want to adjust them, we use one of the `scale_` functions.
- These functions have the following general structure:

`scale_<MAPPING>_<KIND>()`

- So, for example:
  - `scale_x_continuous` controls `x` scales for continuous variables
  - `scale_y_continuous` controls `y` scales for continuous variables
  - `scale_color_discrete` controls `color` scales for discrete variables
  - `scale_shape_discrete` controls `shape` scales for discrete variables

# Scales in action

Let's see this in action. Say we wanted to override the default for tick marks by having a mark every 5 mm on the **x** axis and every 500 g on the **y** axis. We can use the **breaks** argument to control the position of the tick marks:

**Code****Plot**

```
1 ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g,  
2                       color = species, shape = island)) +  
3   geom_point(size = 3) +  
4   scale_x_continuous(breaks = seq(170, 230, by = 5)) +  
5   scale_y_continuous(breaks = seq(3000, 6000, by = 500))
```

# Scales in action

Now, let's say we also wanted to change the colours and shapes used to represent species and island, respectively. For **manually** adapting them, we use `scale_color_manual` and `scale_shape_manual`:



























Code

Plot

```
1 ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g,  
2                       color = species, shape = island)) +  
3   geom_point(size = 3) +  
4   scale_x_continuous(breaks = seq(170, 230, by = 5)) +  
5   scale_y_continuous(breaks = seq(3000, 6000, by = 500)) +  
6   scale_color_manual(values = c("red", "blue", "green")) +  
7   scale_shape_manual(values = c(17, 18, 8))
```

# A word on shapes

- The following shapes are most commonly used in R:

0	1	2	3	4	
					
5	6	7	8	9	
					
10	11	12	13	14	
					
15	16	17	18	19	
					
20	21	22	23	24	25
					

- You can reference them by using the numbers indicated and passing them to `scale_shape_manual` in the order of the factor levels.
- In the example before, we had `Biscoe = 17`, `Dream = 18` and `Torgersen = 8`.



# A word on colour

- Whenever we map `color` or `fill` as an aesthetic, `ggplot2` uses default colour or fill scales that do a fine enough job for exploratory data analysis, but typically have to be tweaked for publication-ready plots.
- In the example before, we have **manually** overridden the default using `scale_color_manual` and the names of three colours (red, blue and green).
- Choosing the right colour(s) for data visualization is actually a very complex problem that has to consider adequate representation of the underlying data as well as differences in colour perception.
- We will only briefly skim the surface of colour choice by looking into some of the excellent options for setting colours that already exist out of the box in `ggplot2`.

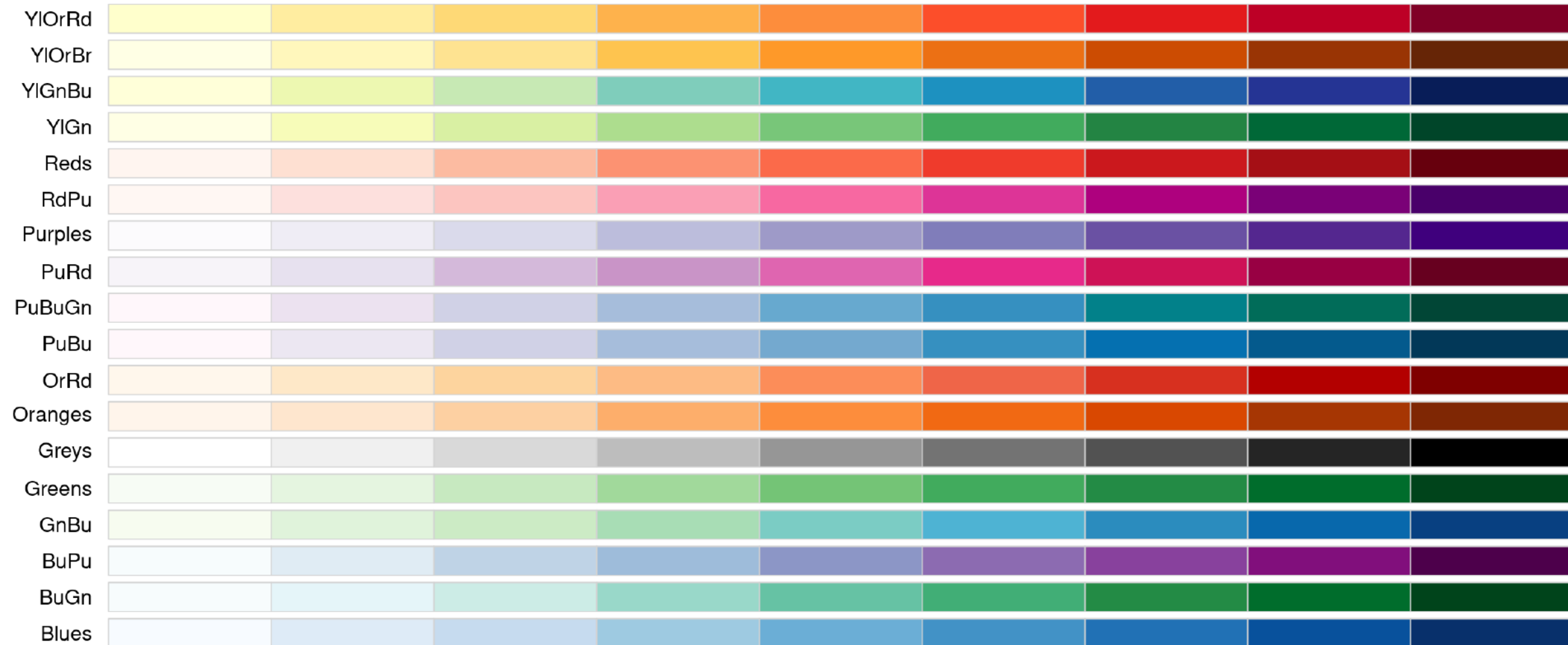
# A word on colour

- A collection of colours used for representing data is called a **palette**.
- A very popular family of such palettes available in `ggplot2` are the `colorbrewer` palettes. For **discrete** variables, they come in three different types:
  - **Qualitative**: best suited to represent unordered categorical data, e.g. penguin species.
  - **Sequential**: best suited to represent ordered data that progresses from low to high, e.g. school grades or counts.
  - **Diverging**: best suited to represent ordered data with a neutral midpoint and extremes diverging in both directions, e.g. Likert scales.
- Depending on the type of data we want represented, we can choose a palette from the corresponding family.

# A word on colour – Qualitative palettes



# A word on colour – Sequential palettes



# A word on colour – Diverging palettes

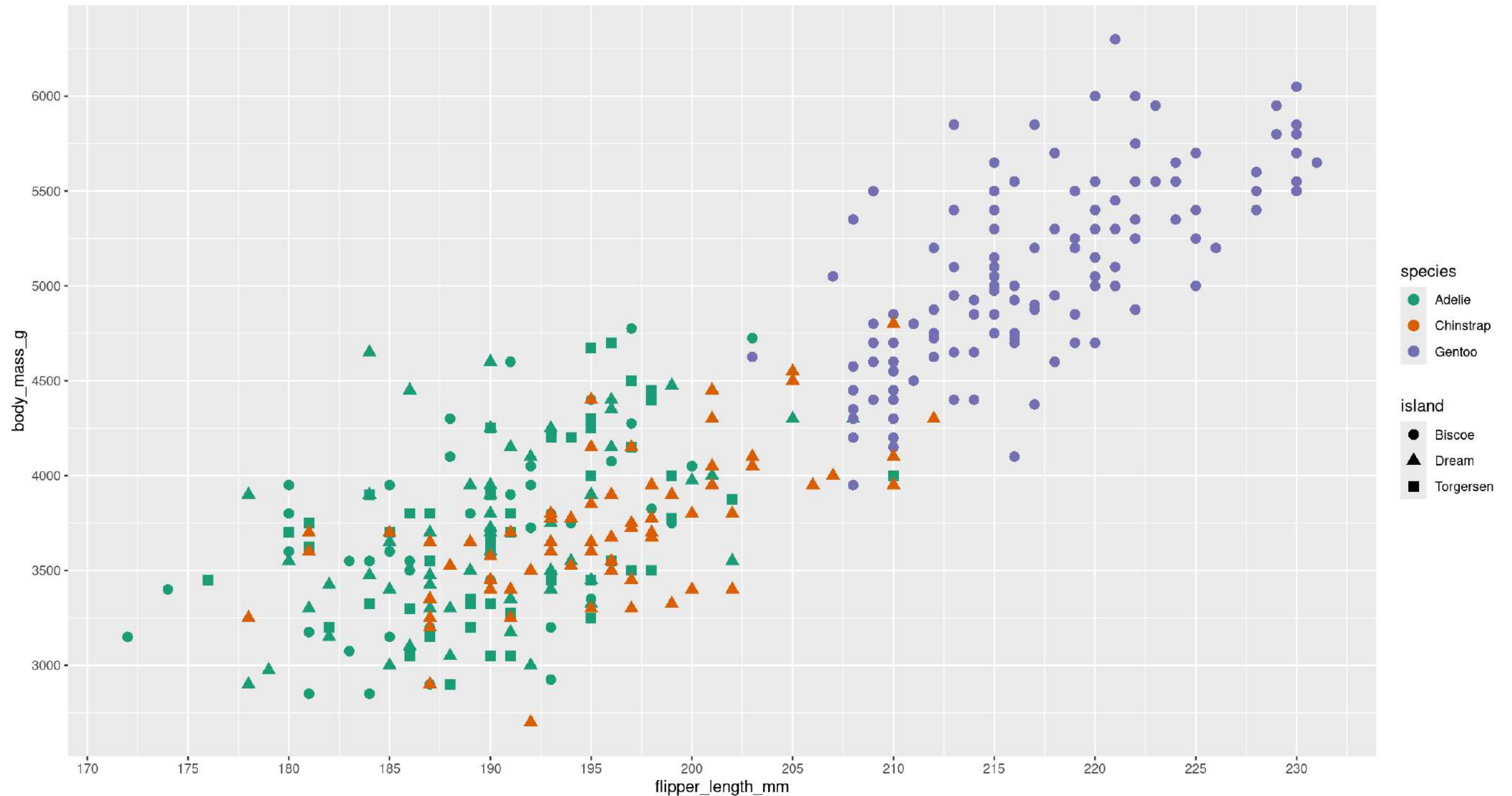


# A word on colour – How to use colorbrewer

- We can access all of these palettes in our plots by referencing them using the names given in the functions `scale_color_brewer` and `scale_fill_brewer`, respectively.
- In our example, we have the `species` variable mapped to colour, which is an unordered categorical variable. A suitable palette to represent it might therefore be `Dark2`, for example.
- To do this, we would run the following code (result on the next slide):

```
1 ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g,  
2                     color = species, shape = island)) +  
3   geom_point(size = 3) +  
4   scale_x_continuous(breaks = seq(170, 230, by = 5)) +  
5   scale_y_continuous(breaks = seq(3000, 6000, by = 500)) +  
6   scale_color_brewer(palette = "Dark2")
```

# A word on colour – How to use colorbrewer



# A word on colour – How to use colorbrewer

- With these **colorbrewer** palettes, we can represent data from **discrete** variables, like categorical or count data.
- However, what if we want to represent a **continuous** variable using colour, such as penguin bill length?
- All **sequential** and **diverging** **colorbrewer** palettes can also be used for continuous scales. Depending on the aesthetic, we only have to pass them into the functions **scale\_color\_distiller** or **scale\_fill\_distiller**.
- Penguin bill length does not have a neutral midpoint, so we use a sequential palette like **YlOrRd** to represent it (result on the next slide):

```
1 ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g,
2                     color = bill_length_mm)) +
3   geom_point(size = 3) +
4   scale_x_continuous(breaks = seq(170, 230, by = 5)) +
5   scale_y_continuous(breaks = seq(3000, 6000, by = 500)) +
6   scale_color_distiller(palette = "YlOrRd")
```

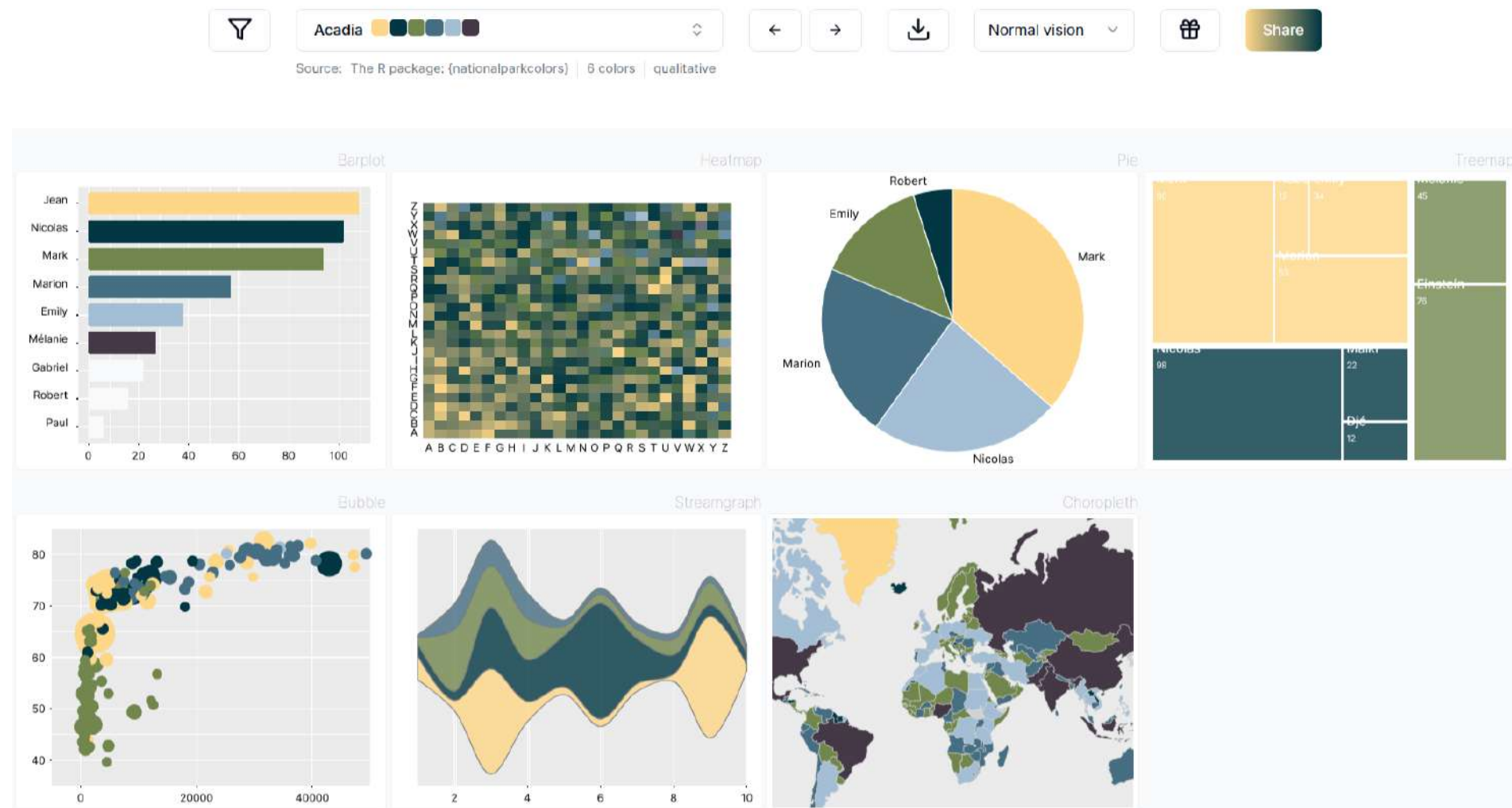


# A word on colour – How to use colorbrewer



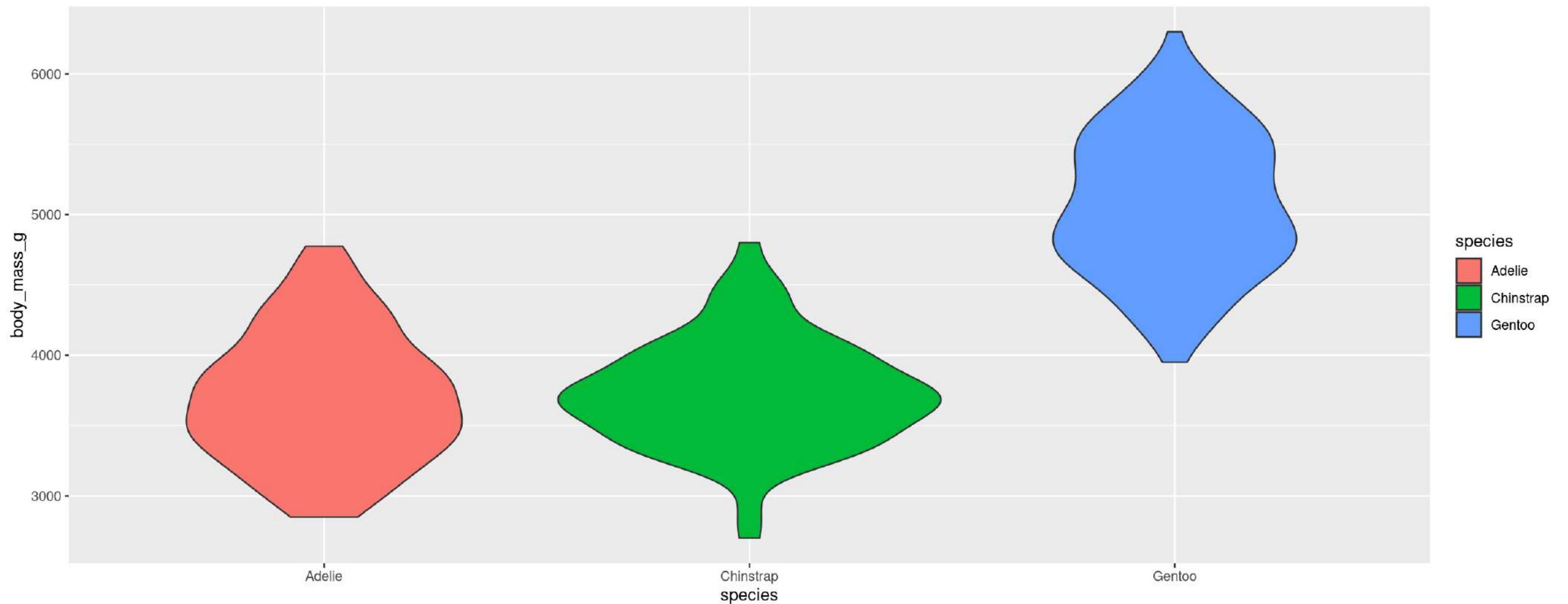
# A final word on colour

- The options for colour choice are virtually endless. The goal here was merely to introduce colour scales and how to change them.
- If you find yourself unhappy with the colour palettes provided by [colorbrewer](#), have a look at the [Palette Finder](#) in the R Graph Gallery:



# Understanding guides

Now let's discuss **guides**. For this, consider again our violin plot from earlier:



This plot contains redundant information: the species can be inferred from the axis labels, we do not need the additional legend for **species**.

# Understanding guides

To “switch off” the guide for a particular aesthetic, we can simply call the `guides` function with the name of the aesthetic as a named argument set to `"none"`:

Code

Plot

```
1 ggplot(penguins, aes(x = species, y = body_mass_g, fill = species)) +  
2   geom_violin() +  
3   guides(fill = "none")
```

# Understanding guides

- Of course, the `guides` function can do much more than simply to “switch off” legends for individual scales.
- In fact, it can be used to customize virtually every aspect of a legend such as:
  - where the title of the legend should go
  - where the labels should be placed
  - in which orders the labels should be displayed
  - and much more...
- As with many aspects of `ggplot2`, this bridge is best crossed, when you first come to it...

# Understanding themes

- Finally, a very important domain of plot customization is opened up by the `theme` function. It allows us to customize all aspects that are not directly related to the data being displayed.
- A very common basic use of the `theme` function is to place the legend at a different location in the plot and to change font sizes of different labels (result on the next slide):

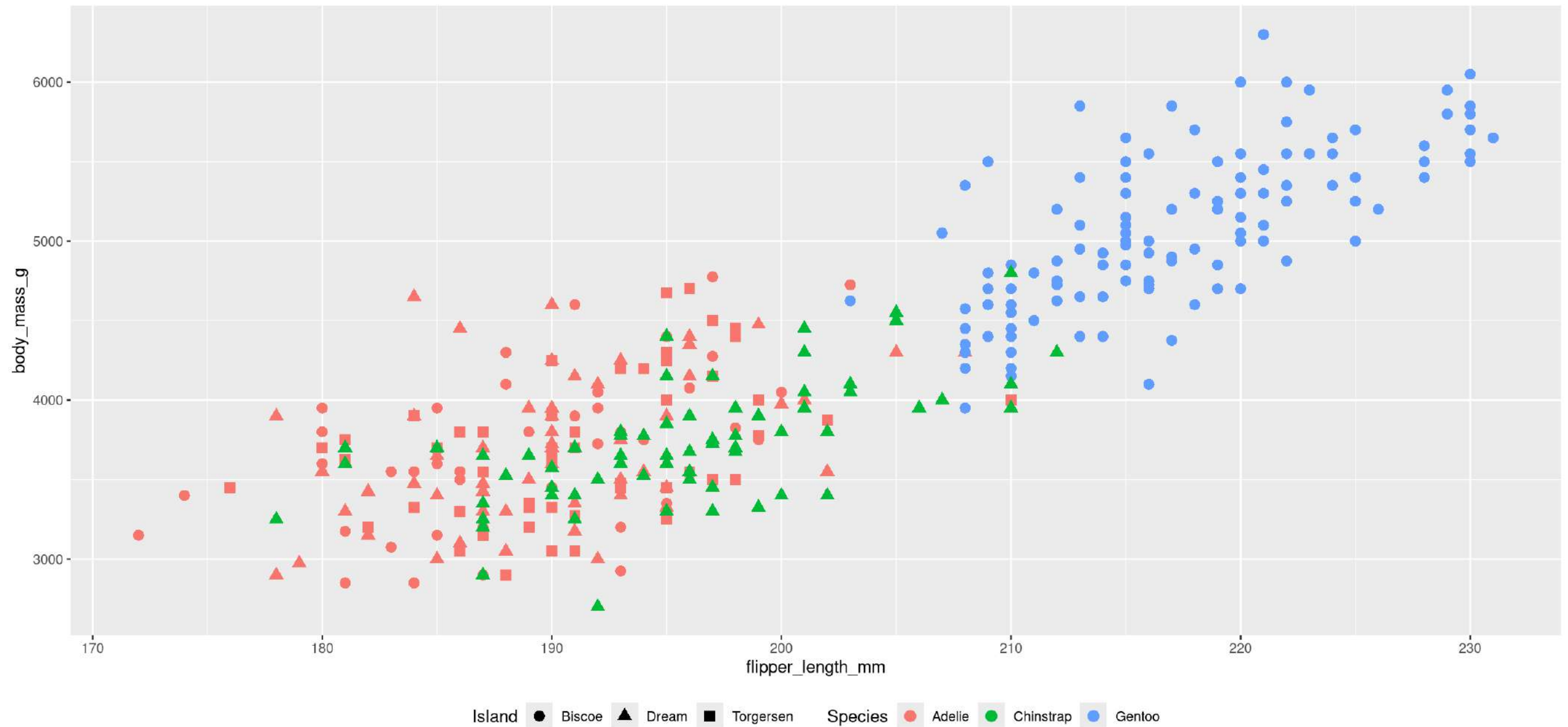
```
1 ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g,  
2                       color = species, shape = island)) +  
3   geom_point(size = 3) +  
4   labs(title = "Body mass and flipper length",  
5         subtitle = "Dimensions for Adelie, Chinstrap, and Gentoo Penguins",  
6         caption = "Source: palmerpenguins package",  
7         color = "Species",  
8         shape = "Island") +  
9   theme(legend.position = "bottom",  
10         plot.title = element_text(size = rel(1.75)),  
11         plot.subtitle = element_text(size = rel(1.5)),  
12         plot.caption = element_text(size = rel(0.75)))
```



# Understanding themes

## Body mass and flipper length

Dimensions for Adelie, Chinstrap, and Gentoo Penguins



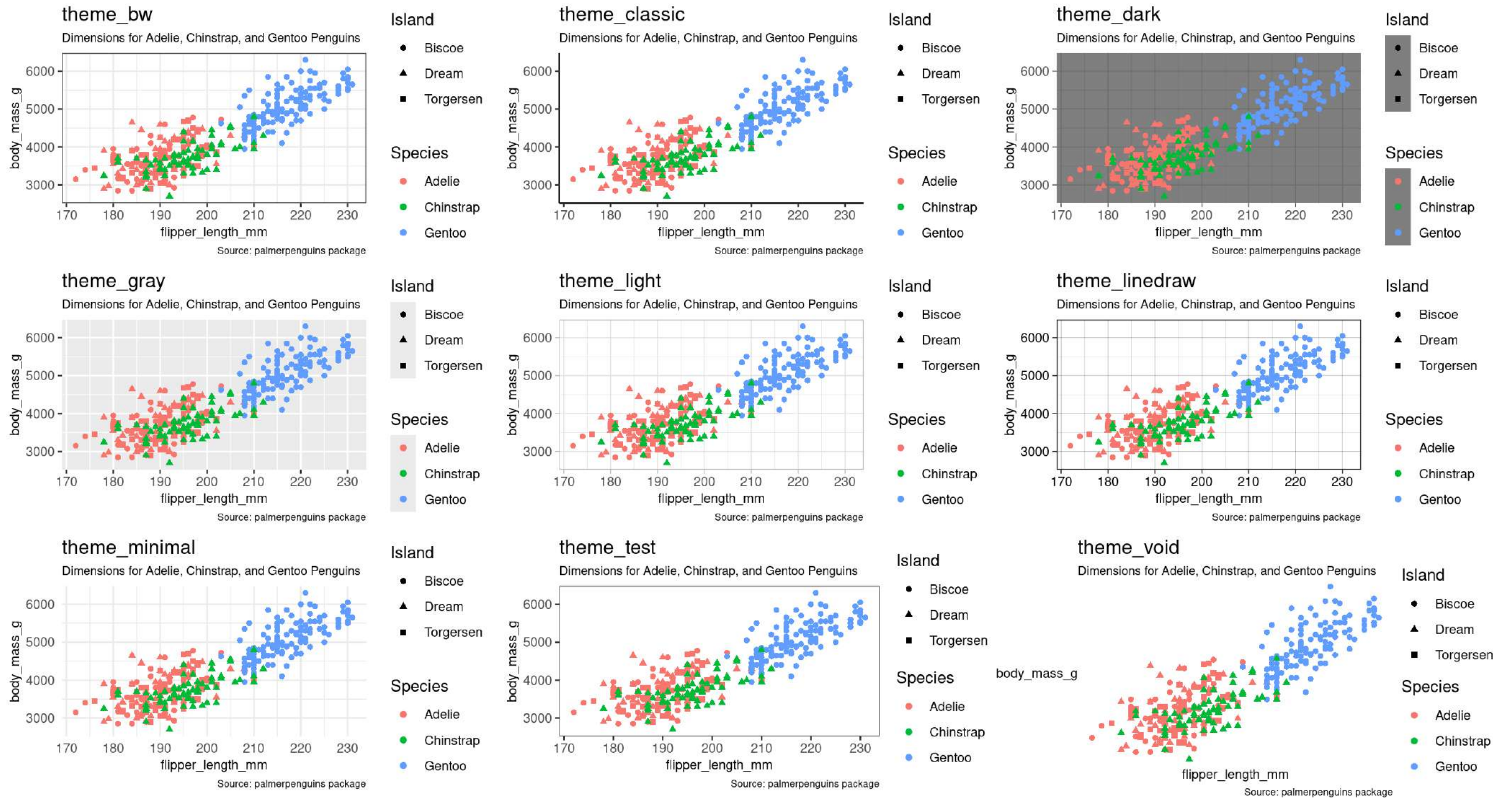
Source: palmerpenguins package

# Using pre-built themes

- While customizing every single aspect of a plot is definitely possible in `ggplot2`, it is also incredibly cumbersome and usually not necessary.
- Instead, we can simply layer on a pre-built **theme** that changes the overall look of a plot all at once.
- **Themes** that come shipped with `ggplot2` are: `bw`, `classic`, `dark`, `gray` (default), `light`, `linedraw`, `minimal`, `test` and `void`.
- The following slide gives an overview of our plot would look like in each of these themes. If none of those suit your needs, you can have a look at the `ggthemes` package that has plenty more to offer.



# Using pre-built themes



# Saving your work

- Now that we know how to create and customize our own plots, we might want to **save** them to include them in a presentation or send them to a colleague.
- The easiest way to do this is to use the function `ggsave`. By default, it will save the most recently created plot into the file name you provide:

```
1 ggsave(filename = "my_figure.png")
```

- You can save the plot as PDF, JPG, PNG (or several other formats) by changing the **file ending** accordingly.
- You can also change the dimensions and the resolution of the plot by changing arguments `width`, `height` and `dpi`, respectively.
- As always, for more details, refer to the documentation via `?ggsave`.

# A sneak peek into what is possible

# A sneak peek into what is possible

