Data visualisation with R

Sophie Lee

Table of contents

Welcome! **5**
How to use these materials . . . 5
Data used in the course . . . 6
Feedback and issues . . . 7
License . . . 7

1 Introduction **8**
1.1 Why data visualisation? . . . 8
1.2 A grammar of graphics . . . 8
1.3 Choosing the most appropriate visualisation . . . 11

2 Aesthetics and geometries **13**
2.1 Aesthetic markings . . . 13
2.2 Geometries . . . 16
2.3 Exporting visualisations . . . 18
Exercise 1 . . . 18

3 Scale functions **20**
3.1 Customising axes . . . 20
3.2 Customising colour scales . . . 21
3.2.1 Pre-built colour palettes . . . 22
3.2.2 Customising colour palettes . . . 24
Exercise 2 . . . 26

4 Annotations and titles **28**
4.1 Plot and axis titles . . . 28
4.2 Annotations . . . 29
4.2.1 Text labels . . . 30
4.2.2 Annotate function . . . 32
Exercise 3 . . . 34

5 Themes **36**
5.1 Pre-built themes . . . 36
5.2 Customising themes . . . 36
5.3 Creating a theme . . . 40

6 Faceting **42**
6.1 Customising facets . . . 43
6.1.1 Dealing with missing data . . . 43
6.1.2 Scales . . . 44
6.1.3 Facet appearances . . . 45
6.2 Other options for multiple plots . . . 46
6.3 Exercise 4 . . . 47

Appendices **49**

Data description **49**
What is 'CSP'? . . . 49
Descriptions of variables . . . 49
Identifier variables . . . 49
Regions of England . . . 49
Settlement Funding Assessment (SFA) . . . 50
Under-indexing business rate multipliers . . . 50
Council tax . . . 50
New Homes Bonus . . . 50
Rural Services Delivery Grant . . . 50

Further reading **51**

A Exercise solutions **52**
A.1 Exercise 1 . . . 52
Question 1 . . . 52
Solution . . . 52
Question 2 . . . 53
Solution . . . 54
Question 3 . . . 55
Solution . . . 55
Question 4 . . . 57
Solution . . . 57
A.2 Exercise 2 . . . 59
A.2.1 Question 1 . . . 59
Solution . . . 59
A.2.2 Question 2 . . . 61
Solutions . . . 61
A.3 Exercise 3 . . . 62
A.3.1 Question 1 . . . 62
Solution . . . 62
A.4 Question 2 . . . 63

Solution 64

Welcome!

Welcome to the course materials for the Data visualisation with R short course. This course introduces the `ggplot2` package and its underlying grammar of graphics. Participants will understand how to choose the most appropriate type of visualisation, based on the type and number of variables, and the intention of the plot. We will then build visualisations, layer by customisable layer, to transform simple plots into beautiful, informative graphics.

By the end of this course, you will:

- Create compelling, clear data visualisations using the `ggplot2` package
- Customise graphs using `scale` and `theme` functions
- Add annotations to graphs to make them as clear and accessible as possible
- Know when to facet graphs to show multiple plots on the same graph
- Save personalised colour palettes and theme functions to ensure visualisations are consistent
- Understand the key principles of data visualisation to ensure they are efficient, accessible, and honest

How to use these materials

This book provides a combination of written explanations, code examples, and practical exercises to allow you to practice what you have learned.

Code examples will be provided in code blocks, such as this one:

```
1 + 1
```

Code in these blocks can be copied and pasted into your R session to save time when coding. We recommend typing the code yourself to familiarise yourself with the coding process and use the copy option if you are really stuck!

Throughout the book, you will see colour-coded boxes which are used to highlight important points, give warnings, or give tips such as keyboard shortcuts.

> **Note**
>
> These boxes will be used to highlight important messages, supplementing the main text.

> **Hint**
>
> These boxes will contain useful hints, such as keyboard shortcuts, that can make your coding life a little easier!

Style tip

These boxes contain style tips to ensure that your code follows the Tidyverse style guide, making it as consistent and readable as possible.

> **Warning**
>
> These boxes will contain warnings and highlight areas where you need to be more cautious in your coding or analysis.

To make these notes as accessible as possible, they are available to view in dark mode by toggling the  button. They are also available to download as a PDF file using the  button.

All exercise solutions are available in the appendices. Please attempt the exercises yourself first, making full use of R's built in help files, cheatsheets (where available), and example R code in this book. Going straight to the solutions to copy and paste the code without thinking will not help you after the course!

Some exercises contain expandable hints, such as functions required to complete them, that can be viewed when needed. For example:

:::{callout-caution collapse="true"} ## Exercise hint

The functions you will need for this exercise are `filter` and `count`. :::

Data used in the course

The examples and exercises in these materials are based on real world data....

Data for this course can be downloaded from the `data` folder of this course's repository.

For more information about this data, including variable descriptions and sources, see the appendix.

Feedback and issues

This book is a work in progress and will be updated based on course feedback and requirements of participants. If you spot a bug or mistake in these notes, please let me know by raising an issue.

If you enjoyed using these resources and would like to find more of them or attend a live course, please visit my website, follow me on Twitter and LinkedIn, or support more free resources by buying me a coffee!.

If you are interested in organising a bespoke course or have consultancy opportunities you think I would be a good fit for, please get in touch!

License

1 Introduction

1.1 Why data visualisation?

Data visualisation is arguably one of the most important part of any analysis journey. It is a powerful tool with a wide range of uses, including:

- Exploring the data: checking for outliers, potential errors, and generally 'getting to know' our data
- Generating hypotheses: investigating potential trends in the data, identifying important variables to include in a model
- Checking parametric assumptions: validating analysis methods that require certain assumptions to be valid, e.g. distributions of variables
- Communicating results: often visualisations are far more powerful and concise than text or tables at conveying important messages to readers

Each of these intentions may require a different visualisation, but all of them must follow the same design priciples. They must be clear, appropriate, honest, and inclusive.

1.2 A grammar of graphics

Although R has a number of options to display data, this course will focus on one of the more popular and flexible approaches, `ggplot`. `ggplot2` is an R package that is a member of the `tidyverse`, a suite of R packages designed to make data analysis/science more accessible and efficient.

The `ggplot2` package implements a 'grammar of graphics' approach, in which graphs are composed of multiple layers. According to the grammar of graphics, all visualisations must contain three elements: the data, the information we wish to display, and some mapping, describing how to visualise the information.

To illustrate this, let's create a simple visualisation to investigate the relationship between the body mass and flipper length of penguins in the Palmer archipelago, Antarctica.

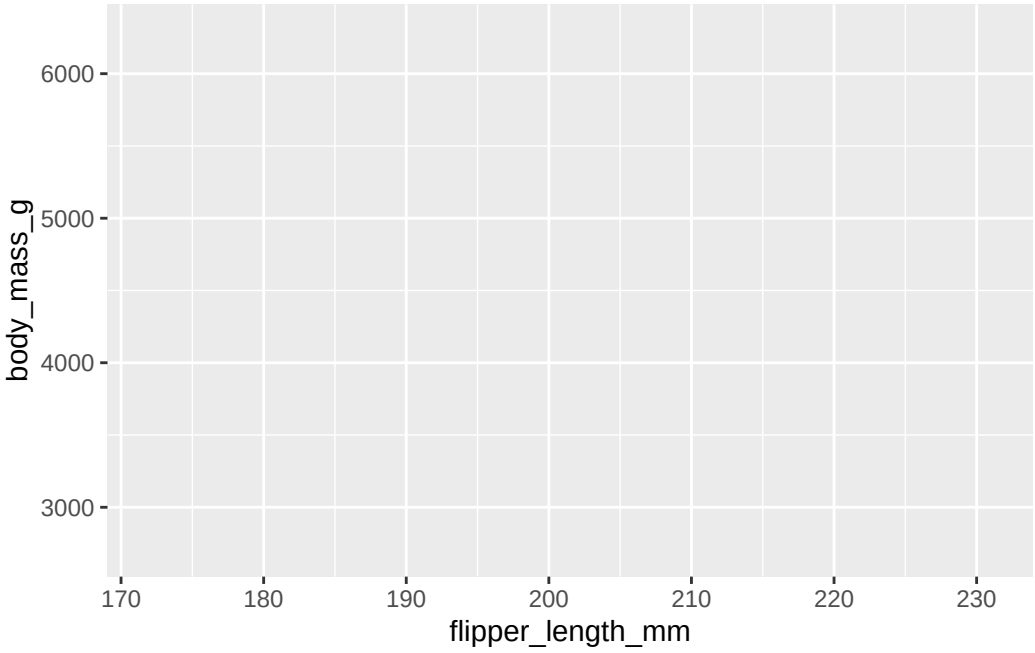
> **Warning**
>
> If you have never used the `pacman` package before, install this to your computer using the following function:
>
> ```
> install.packages('pacman')
> ```
>
> The `pacman` package is a set of package management functions which is designed to make tasks such as installing and loading packages simpler, and speeds up these processes. The `p_load` function acts as a wrapper for the `library` function, which checks to see if a package is present on the current machine, and installs them from CRAN if they are not. If a package exists on a machine, it will load it as usual.

```
pacman::p_load(tidyverse, palmerpenguins)  ①

data(penguin)  ②

ggplot(data = penguins)  ③
```

① We must first load the R packages we require. This may take a long time if you have never downloaded `tidyverse` before, do not worry if a lot of text appears, this is normal!!
② We then load the dataset we will be using for this part of the course from the `palmerpenguins` package.
③ The first element required for a `ggplot` is the data. As the other two layers are missing, this will just produce a blank plot area.

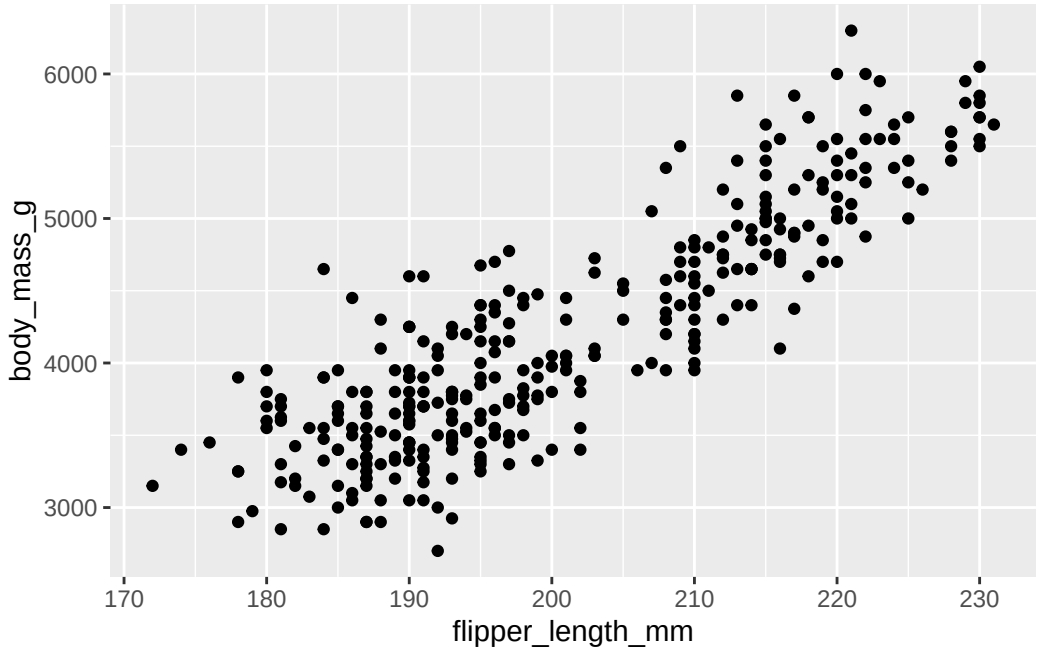
```
ggplot(data = penguins, aes(x = flipper_length_mm, y = body_mass_g))        ④
```

④ The second element that is required for a `ggplot` is the information we wish to present. This is the flipper lengths and body masses of our penguin sample.



```
ggplot(data = penguins, aes(x = flipper_length_mm, y = body_mass_g)) +      ⑤
  geom_point()
```

⑤ The final element required is some physical markings of the data. As both variables we wish to present are continuous, we could show every observation as a point in a scatterplot. In `ggplot2`, a marking can be added using a `geom` function.



> **Warning**
>
> Although `ggplot2` is part of the `tidyverse` suite of packages, the pipe symbol (`%>%` or `|>`) is replaced with `+` when adding layers to a `ggplot`.

The plot may not be pretty, but it contains all three elements required by graphics, and shows a clear positive association between penguins' flipper length and body mass. Additional layers will be introduced throughout this course to improve the design of this visualisation.

1.3 Choosing the most appropriate visualisation

The choice of visualisation should be driven first and foremost by the context, the audience, and the goal of the graphic. Often, people choose axes of graphics based on the data available to them. Although the choice of visualisation is influenced (and sometimes restricted) by

the number and type of variables available, it should be fundamentally decided based on the message we wish to convey to the readers and the most efficient way to do this.

Most common visualisations can be generated in `ggplot2` using a `geom` function. There are many options available, some of which will be covered throughout this course, many will not. Table 1.1 provides a sample of some of the most common visualisation types, classified by the number and type of variables they are compatible with. For many more options, including many non-standard graphs, visit the From data to viz or view R code examples through their R graph gallery.

Table 1.1: Common data visualisations, classified by type and number of variables, presented with the `geom` function used to generate them.

Number of variables	Type of variables	Name of visualisation	R function
One variable	Categorical	Frequency table	table
		Bar chart	geom_bar
	Numerical	Histogram	geom_histogram
	Spatial	Map	geom_sf
	Temporal	Line plot	geom_line
Two variables	Two categorical	Frequency table	table
		Stacked/side-by-side bar chart	geom_bar
	One numeric, one categorical	Dot plot	geom_point
		Box plot	geom_boxplot
	Two numerical	Scatterplot	geom_point
> 2 variables	> 2 categorical	Table	table
	2 numeric, one categorical, or > 2 numeric	Scatterplot with different colours/symbols/sizes	geom_point

2 Aesthetics and geometries

Two of the key elements to any `ggplot` object are the information we wish to present, and the way in which we would like to visualise it. This chapter will go into more details about how we specify aesthetic markings of graphs, both manually and using the data, or add layers to a graphic to improve the clarity of our message.

2.1 Aesthetic markings

Any information that we are presenting that is taken from the data must be given within the `aes` wrapper. The argument each variable takes within this wrapper depends on the element of the graph which it defines. For example, in the previous example:

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

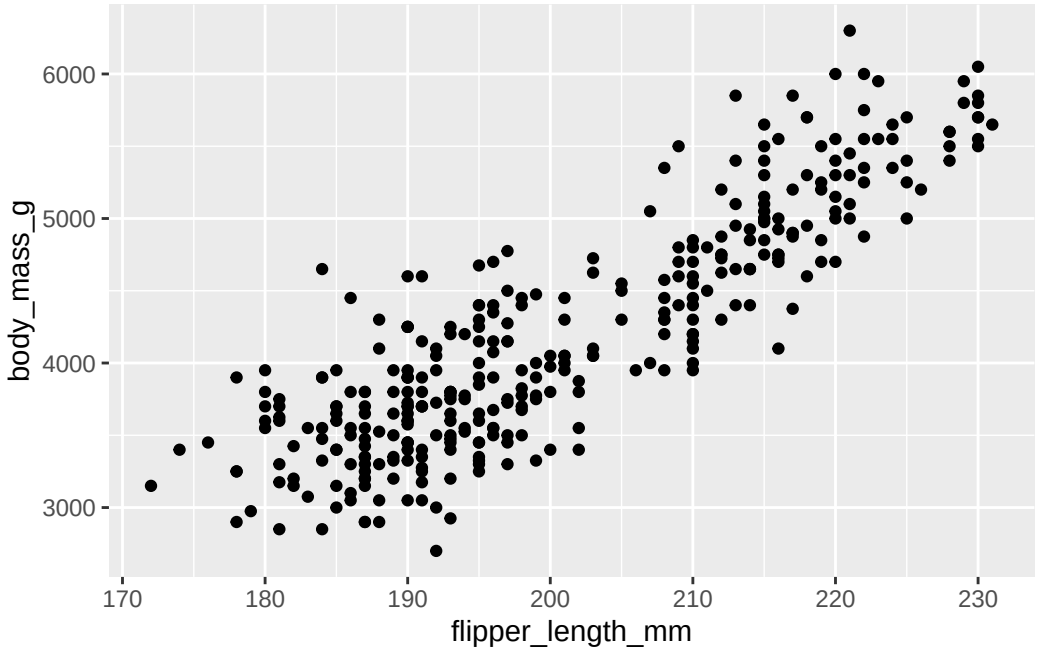


Figure 2.1: Scatterplot showing the size and mass of penguins in the Palmer Archipelago, Antarctica.

The variables define the `x` and `y` axes. Additional variables can be added to a visualisation by using them to customise other elements of a graph, such as:

- `colour`: determines the colour of points (for dot and scatterplots), lines (for line graphs), or borders (for bar charts, histograms and pie charts)
- `fill`: determines the colour of bars or segments
- `shape`: changes the symbols presented on dot and scatterplots
- `linetype`: customises the type of line displayed (solid by default, but can be used to show dashed lines, etc)
- `size`: determines the size of points
- `linewidth`: changes the line width
- `alpha`: controls the transparency of graph elements

For example, if we wish to show how the relationship between flipper length and body mass differs between penguin species, this could be included within the `aes` wrapper to change the `colour` of points:

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

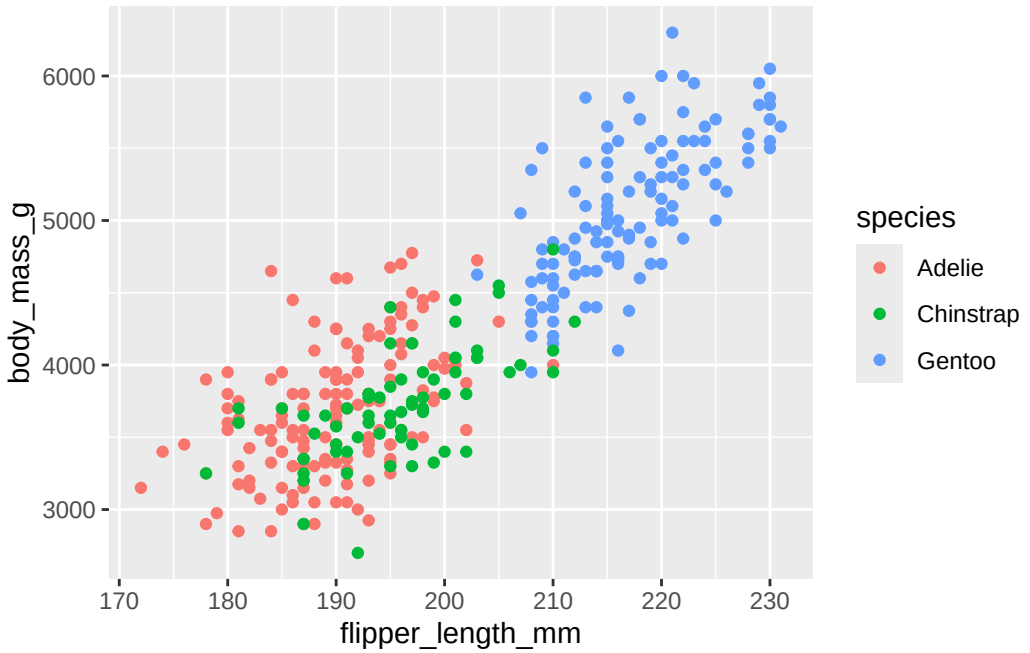


Figure 2.2: Scatterplot showing the size and mass of penguins in the Palmer Archipelago, Antarctica by species.

> **Warning**
>
> Although you should aim to show as much relevant data as possible, be careful not to overload a plot. Too many variables on the same visualisation can make it less informative, confusing the reader and hiding important messages. In this case, it would be better to have multiple, simpler graphs than a single, complicated graph.

These aesthetic options can also be changed manually outside of the `aes` wrapper, within the corresponding `geom` function. For example, if we wanted to make the points in Figure 2.2 larger, we could adjust the code to the following:

```
ggplot(data = penguins,
       aes(x = flipper_length_mm, y = body_mass_g, colour = species)) +
  geom_point(size = 5)
```

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

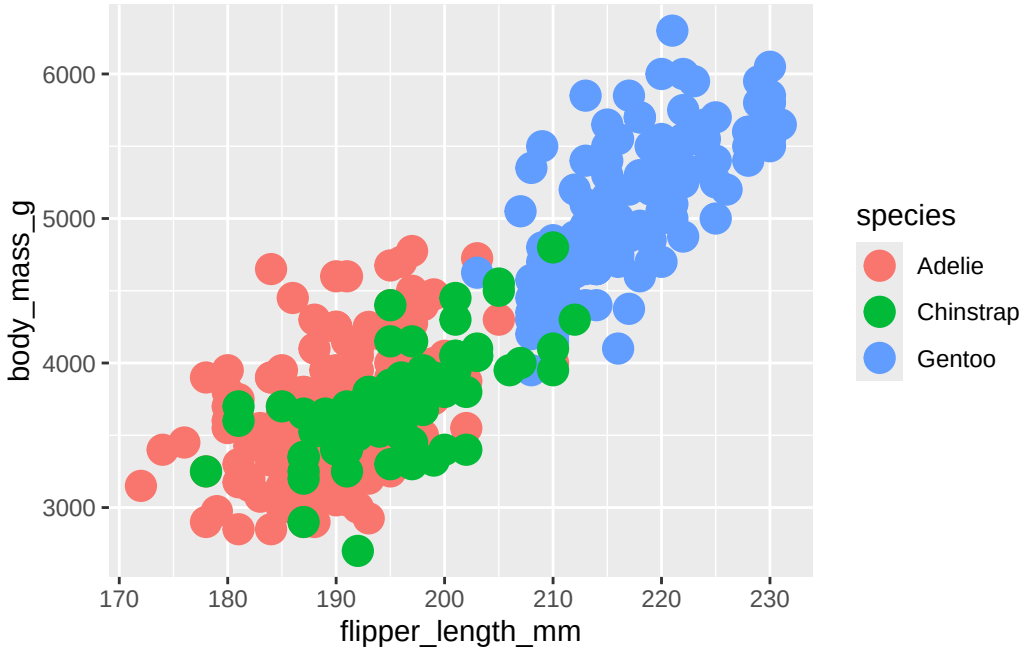


Figure 2.3: Scatterplot showing the size and mass of penguins in the Palmer Archipelago, Antarctica by species.

2.2 Geometries

Most visual markings within `ggplot` are determined by a `geom` object. There are many options within the package that can be used and each requires different combinations of `aes` markings to work. For example, the scatterplot in Figure 2.2 required `x` and `y` to be specified, but a bar chart or histogram would only require `x` as the y-axis is generated from the data.

> Hint
>
> For a full list of geometries included in `ggplot2`, visit the package's cheatsheet

Multiple geometries can be added to the same graph to add information to the plot. These can be used to add information to the graph, such as a reference value or a summary. For example, if we wished to show the difference in the distribution of body mass of penguin species, we could show every individual observation using a point and add a boxplot layer to show the differences in median and interquartile range.

> **Warning**
>
> Ensure that the `outliers` argument of `geom_boxplot` is set to `FALSE` to avoid duplicate points.

```
ggplot(data = penguins,
       aes(x = species, y = body_mass_g)) +
  geom_jitter() +                                          ①
  geom_boxplot(outliers = FALSE, alpha = 0.75)             ②
```

① Jittering points adds a small amount of noise on the x-axis to spread points out, ensuring we can see each observation, even when they overlap.
② Use alpha to make the box semi-transparent so we can still see the points underneath.

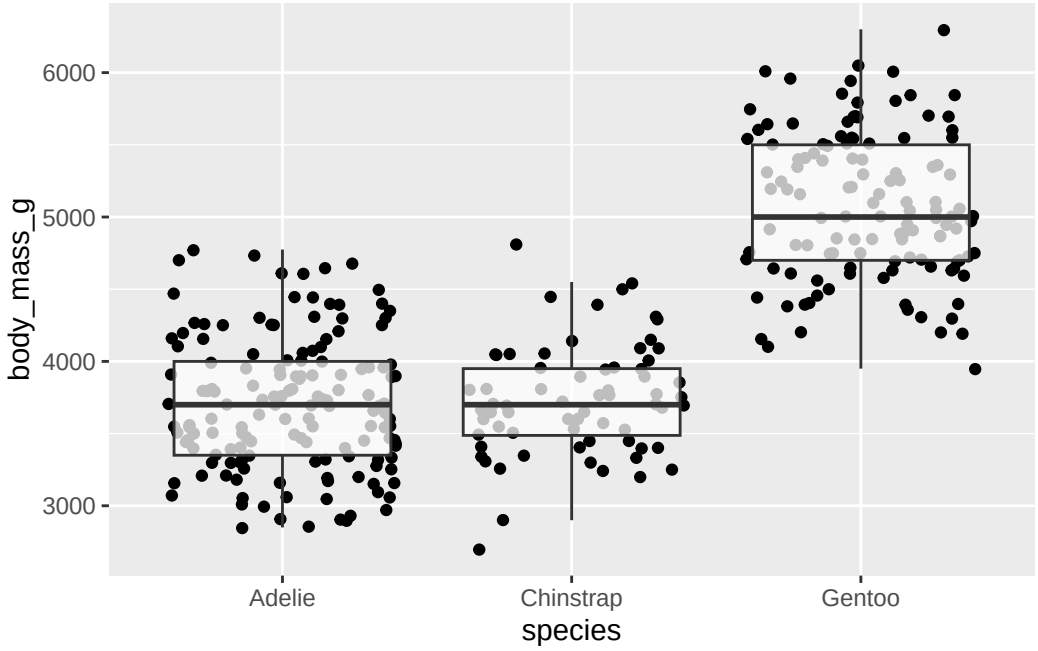
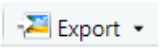


Figure 2.4: Comparison of body mass between penguin species

Other useful layers may include `geom_hline`, `geom_vline` and `geom_abline` which add a manually defined horizontal, vertical, or diagonal line respectively. To generate a line of best fit from the data, we can use `geom_smooth`.

> **Hint**
>
> By default, `geom_smooth` generates a curved line. To change this, change the `method` argument to an appropriate method. For example, `method = "lm"` generates a straight line or check the helpfile `?geom_smooth` for other options.

2.3 Exporting visualisations

Visualisations created in RStudio can be exported manually from the RStudio interface by clicking the 

button. This allows visualisations to be copied and pasted into documents (using the Copy to clipboard option) or saved as an image (for example, `.png` or `.jpg`) or PDF. Visualisations can also be stored as objects using the `<-` symbol. These objects can then be saved using the `ggsave` function:

```
species_mass_plot <- ggplot(data = penguins,
       aes(x = species, y = body_mass_g)) +
  geom_jitter() +
  geom_boxplot(alpha = .75)

ggsave(species_mass_plot, filename = "output/species_mass_plot.png")
```

The `ggsave` function can be customised to change the file type, `height`, `width` and resolution (using the `dpi` argument).

> **Hint**
>
> `ggsave` is compatible with a range of file types, including `png`, `jpg`, `pdf` and `svg`. Saving these visualisations in a vectorised format, such as `svg` allows graph elements to be edited outside of R. For example, after pasting an exported `svg` file into Microsoft Word, ungroup the image. This allows customisation of axes text, legends, background colours, etc.
> External editing is not recommended as output would no longer be reproducible via R.

Exercise 1

1. Load the `CSP_2020.csv` file into R and save it as an object named `csp_2020`. Ensure that all variables are correctly specified by type. For more information about this dataset, including the source and variable descriptions, check the appendix.

Exercise hint

To read a csv file in, use the `read_csv` function. Consider storing the raw data in its own folder.
All variables should be categorised as numberic (or `dbl` in `tidyverse`), apart from the `authority` variable (which should be `character`) and the `region` variable (which should be `factor`). Use the `mutate` and `factor` functions to fix this.

2. The following code was intended to produce a scatterplot showing the relationship between the `sfa_2020` and `ct_total_2020` variables with the points in blue. Debug the code to fix the problem:

```
ggplot(data = csp_2020) +
  geom_point(aes(x = sfa_2020, y = ct_total_2020, colour = "blue"))
```

Exercise hint

Check which aesthetic elements are defined by the data and which are defined manually. Consider which of these need to be in the `aes` wrapper.

3. Add a straight line of best fit to the scatterplot and interpret the result. What is the shaded area surrounding the line and how can we remove it?

Exercise hint

Use the `geom_smooth` function and check the `method` argument.
To figure out what this shaded area is, check the helpfile `?geom_smooth`, specifically the arguments `se` and `level`.

4. Use an appropriate visualisation to check the distribution of the `sfa_2020` variable. Interpret this visualisation.

Exercise hint

Histograms are used to check the distribution of numeric variables, and are generated using `geom_histogram`.
Based on this visualisation, do you think the data are normally distributed? Are there any outliers or potential errors? Use the `filter` function to investigate outliers.

3 Scale functions

Visual markings of a `ggplot` object that are defined within the `aes` wrapper can be customised using a `scale` function. These functions are added to a `ggplot` object as additional layers. Each `scale` function will have different arguments related to the aesthetic they customise.

3.1 Customising axes

Scale functions that customise axes generally take the form `scale_axis to customise_scale of variable`. For example, Figure 2.4 would require `scale_y_continuous` to customise the numeric y-axis and `scale_x_discrete` to customise the categorical x-axis.

Axes scale functions contain many options that can be used to change the axis title, limits and ticks, amongst other things. Some of the most common arguments include:

- `name =` changes the axis title
- `limits = c(...)` sets the axis limits
- `breaks = c(...)` defines tick marks
- `labels = c(...)` attaches labels to break points
- `expand = expansion(0)` removes the default blank space around the axis limits (this can also be used to add space by replacing `0` with either `add =` or `mult =` depending if this change is additive or multiplicative)
- `transform =` transform the scale the axis is shown on. Transformations include `reverse`, `sqrt`, `log`, etc. For a full list, view the appropriate help file

```
ggplot(data = penguins,
       aes(x = species, y = body_mass_g)) +
  geom_jitter() +
  geom_boxplot(alpha = .75) +
  scale_x_discrete(name = "penguin species", expand = expansion(mult = .2)) +
  scale_y_continuous(name =  "body mass (g)", breaks = c(3000, 4000, 5000))
```

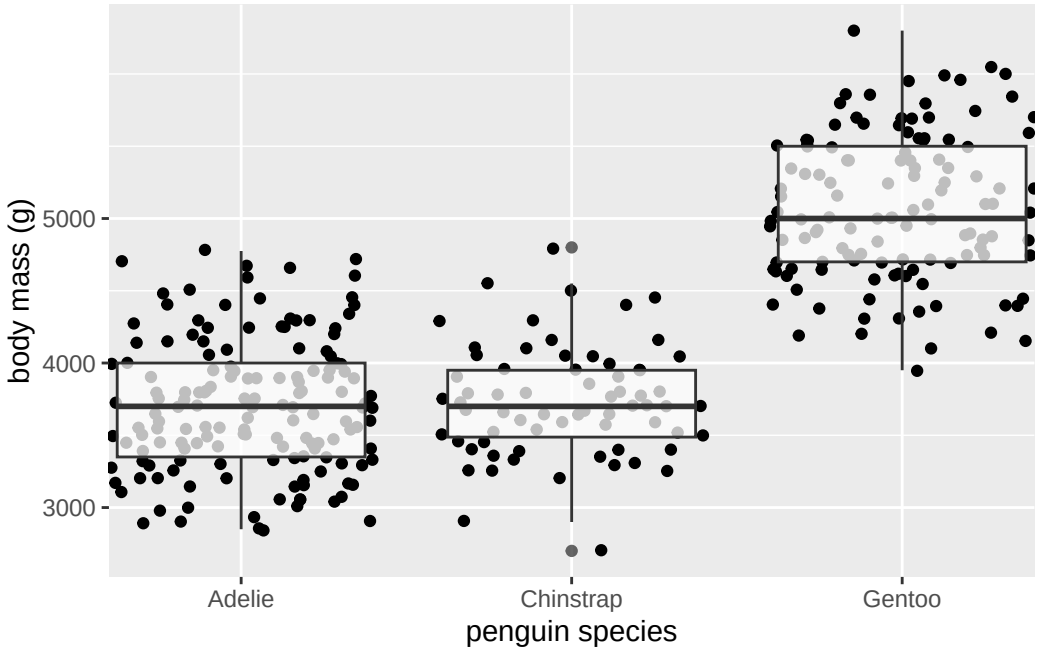


Figure 3.1: Comparison of body mass between penguin species

> **Warning**
>
> Do not adjust axis limits if this can potentially distort the data. Visualisations should adhere to the principle of proportional ink. That is, the amount of ink used in a visualisation should be proportional to the quantities it represents. Bar charts with a y-axis that does not begin at 0 is a common example of a violation of this principle.

3.2 Customising colour scales

There is a wide range of options available for customising `colour` and `fill` aesthetics within `ggplot2`. The choice will depend on the type of variable determining colours (whether it is numeric or categorical) and whether we want to use a pre-defined colour palette or manually specify our own.

> **Warning**
>
> When choosing a colour palette, be sure that all colours are distinct to everyone, including those with colour-vision deficiencies. To help check this is the case, use a colour blindness

simulator to see what a visualisation looks like under different types of colour blindness. Avoid potentially harmful stereotypes when choosing colours to represent groups, and avoid cyclical palettes, such as the rainbow palette, to avoid confusion between high and low values.

3.2.1 Pre-built colour palettes

There are thousands of colour palettes that are available within R. Some of them are included within the `ggplot2` package, but there are many others that require additional package installation. This website gives a list and preview of all palettes currently available.

Colour palettes included within the `ggplot2` package (and therefore don't require any additional packages) are the viridis and colorbrewer scales. Both contain palettes that are colour-blind friendly and can be used for either continuous or discrete scales.

For continuous data, use `scale_colour_viridis_c` or `scale_colour_distiller` to select one of the in-built colour palettes (replace `colour` with `fill` when dealing with bars). For discrete or categorical variables, use `scale_colour_viridis_d` or `scale_colour_brewer` instead.

For example, we could use a stacked bar chart to show the different number of penguins recorded per year by species. Each bar will represent the total observations per year, which will be separated into smaller bars per species. Each species will be assigned a different colour using the `Dark2` colour scheme:

```
ggplot(data = penguins,
       aes(x = year, fill = species)) +                    ①
  geom_bar(colour = "black") +                             ②
  scale_fill_brewer(palette = "Dark2")
```

① Use the `species` variable to change the colour of the bars.
② Manually set the border of each bar to black to make it easier to see.

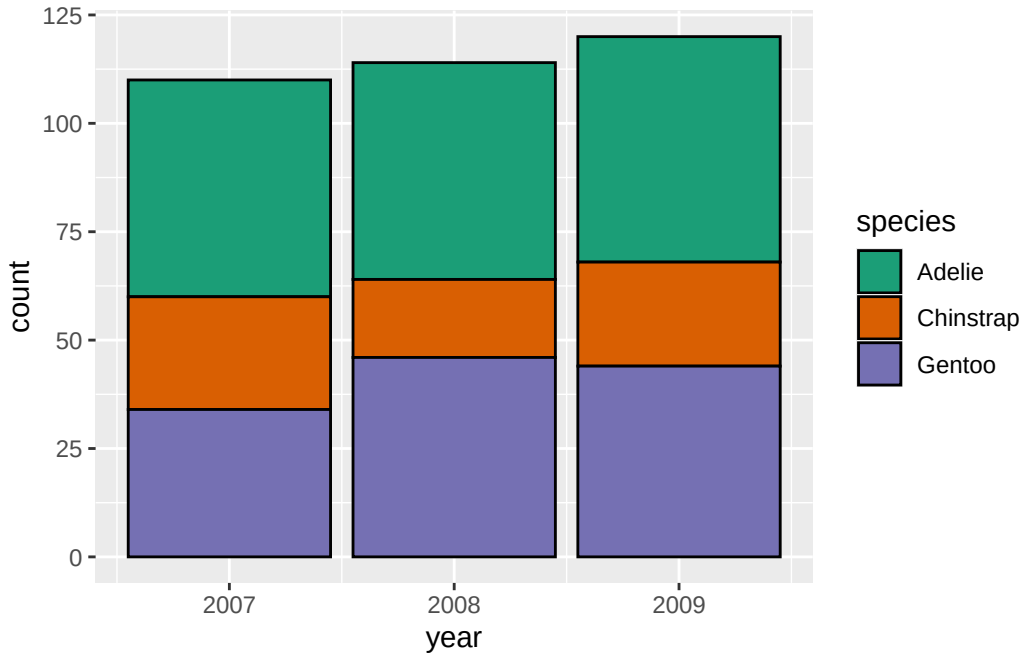
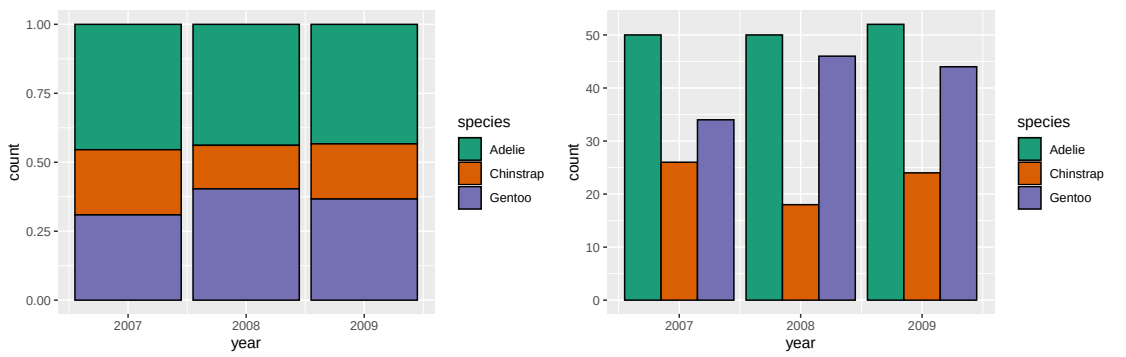


Figure 3.2

Hint

The `geom_bar` function contains the argument `position` which is set to `position = "identity"` by default, producing a stacked bar chart. Changing this option to `"fill"` would convert the graph to a proportional bar chart where each bar has the same height, comparing proportions across groups. Changing the `position` argument to `"dodge"` creates a side-by-side bar chart:

```
ggplot(data = penguins,
       aes(x = year, fill = species)) +
  geom_bar(colour = "black", position = "fill") +
  scale_fill_brewer(palette = "Dark2")
ggplot(data = penguins,
       aes(x = year, fill = species)) +
  geom_bar(colour = "black", position = "dodge") +
  scale_fill_brewer(palette = "Dark2")
```



3.2.2 Customising colour palettes

There are various way of creating your own colour palette if you (or the organisation you are working with) have preferred colours.

For discrete or categorical variables, the `scale_colour_manual` (or `scale_fill_manual`) function allows colours to be specified using the `values` argument.

Style tip

R contains a list of 657 pre-programmed colours that can be used to create palettes (run `colours()` in the console for a full list).

Hexadecimal codes can also be included instead in the form `#rrggbb` (where `rr` (red), `gg` (green), and `bb` (blue) are numbers between 00 and 99 giving the level of intensity of each colour).

```
species_palette <- c("orchid", "turquoise", "rosybrown")                    ①

ggplot(data = penguins,
       aes(x = year, fill = species)) +
  geom_bar(colour = "black") +
  scale_fill_manual(values = species_palette)                               ②
```

① To avoid repetitive coding, define the colour palette as an object.
② Either list colour values or include the palette object. Ensure there are the same number of values as categories.

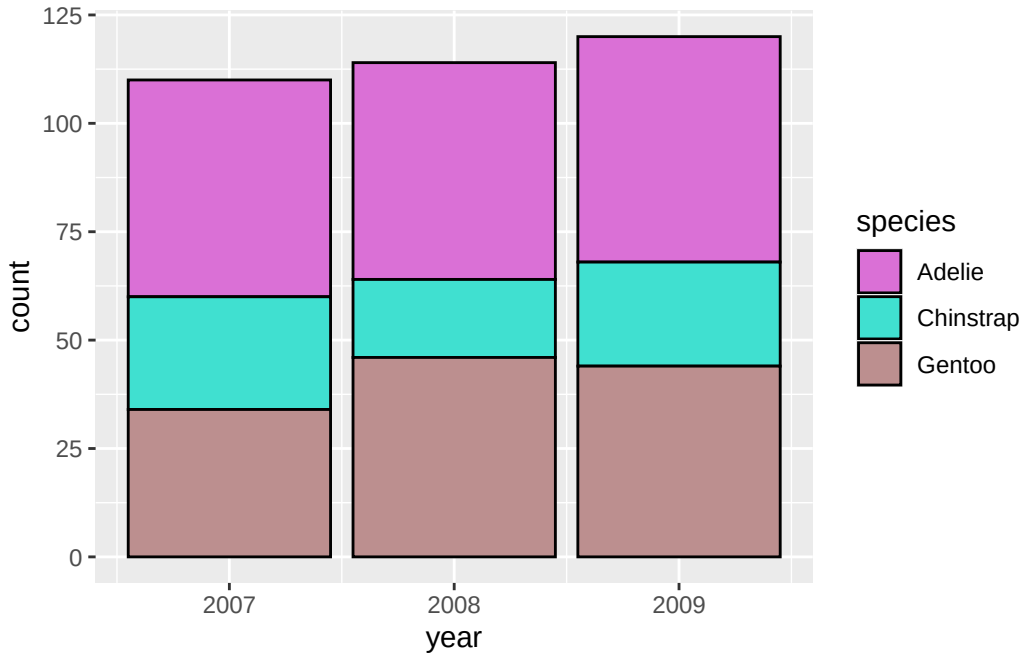


Figure 3.3

When including a continuous variable, palettes can be created using gradients. The choice of function depends on the number of gradients required:

- `scale_colour_gradient` / `scale_fill_gradient`: specifies a two colour gradient based on a `low` and `high` value
- `scale_colour_gradient2` / `scale_fill_gradient2`: specifies a three colour gradient based on a `low`, `mid` (defined by the `midpoint` argument), and `high` value
- `scale_colour_gradientn` / `scale_fill_gradientn`: specifies a palette with more than three colours, customised by setting `colours` and corresponding `values`.

For example, Figure 2.1 could be extended to include information about the bill length of penguins:

```
ggplot(data = penguins,
       aes(x = flipper_length_mm, y = body_mass_g,
           colour = bill_length_mm)) +
  geom_point() +
  scale_colour_gradient(name = "bill length (mm)",
                        high = "#8e3e63", low = "#F85525")
```

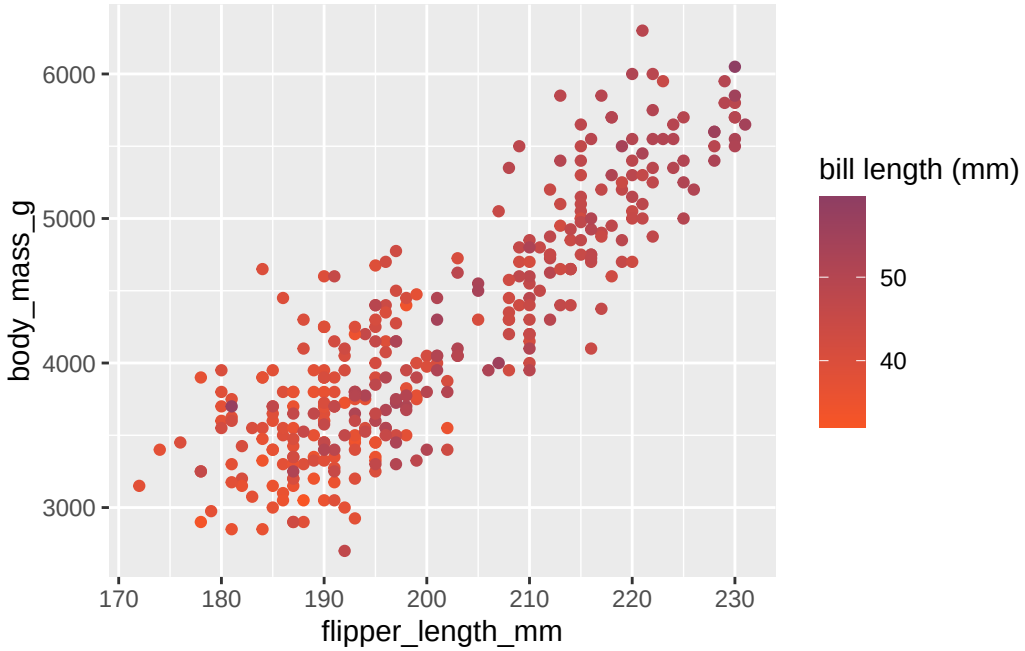


Figure 3.4: Scatterplot showing the flipper length, bill length and body mass of penguins in the Palmer Archipelago, Antarctica.

> **Warning**
>
> Ensure that the colours used to define gradients are distinct enough to make the graph clear (unlike the colours I used in Figure 3.4!).

Exercise 2

1. Use an appropriate visualisation to compare the total core spending power in local authorities 2020 across regions of England. Highlight the London region in a different colour to the other regions to make it stand out more.

> **Exercise hint**
>
> If the total spend is not in the current dataset, create it! Also consider how colour could be included in the `aes` wrapper, and if this does not currently exist in the data, create it!
> You will need the `mutate` and `if_else` functions.

2. Customise the graph above to ensure the axes are labelled appropriately. Add axis breaks every £100million on the y-axis.

> Exercise hint
>
> `breaks` can either be manually specified or a function can be used to generate the list (check the `seq` function).

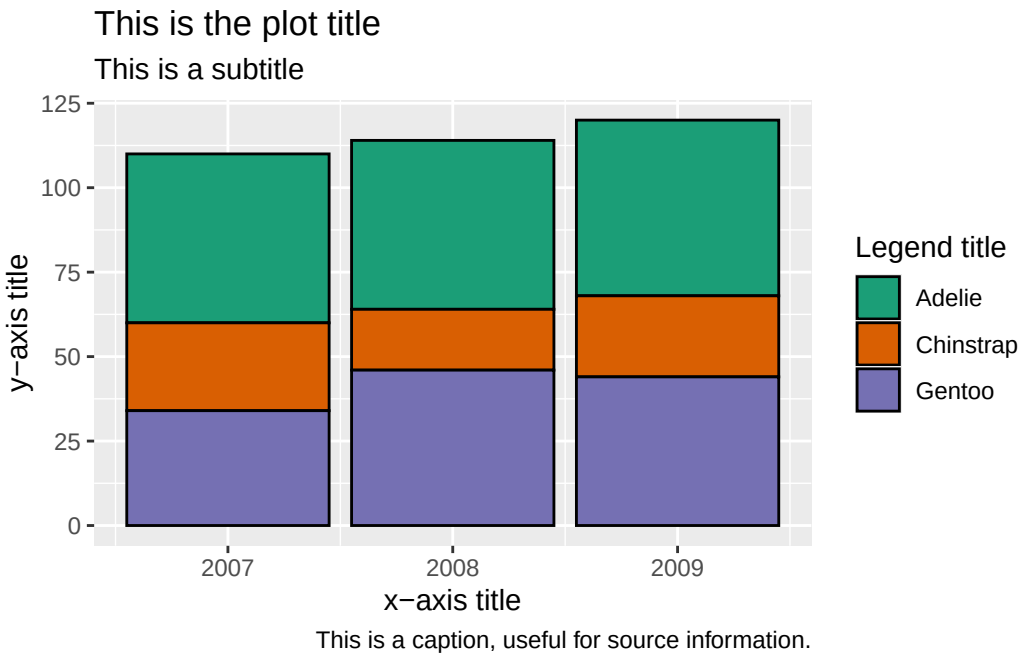
4 Annotations and titles

Besides the layers required to generate a visualisation, additional layers can be added enhance the messages given by the data, drawing readers' attention to interesting findings and the story you are trying to tell.

4.1 Plot and axis titles

In the previous section, we saw how axis and legend titles can be added within `scale` functions using the `name` argument. `ggplot2` also contains the `labs` function which can be added as a separate layer to control these titles and can also add plot titles, subtitles and footnotes:

```
ggplot(data = penguins,
       aes(x = year, fill = species)) +
  geom_bar(colour = "black") +
  scale_fill_brewer(palette = "Dark2") +
  labs(title = "This is the plot title",
      subtitle = "This is a subtitle",
      caption = "This is a caption, useful for source information.",
      x = "x-axis title", y = "y-axis title",
      fill = "Legend title")
```



Style tip

Mathematical equations can be added into `labs` arguments by surrounding the text with the `quote()` function. Check `?plotmath` for examples of equation syntax.

`\n` can be used to specify line breaks within the `labs` arguments.

Specifying any of these arguments as `NULL` (no speech marks) removes the title from the visualisation.

4.2 Annotations

Annotations can be useful to include context to visualisations and draw attention to important messages. Annotations can include text labels, reference lines and shading, amongst others. `ggplot2` contains a number of `geom` objects that can be used to add annotation layers to a visualisation. As these annotations are added within `geom`s, they can be specified using values from the data (when wrapped in the `aes` function) or manually. This section will cover some common annotations but there are many others available (see the ggplot ebook for a more comprehensive list).

4.2.1 Text labels

Text labels can either be added using `geom_text` or `geom_label` (which adds text surrounded by a rectangular box, making it easier to read on busy backgrounds). Aesthetics such as `x`, `y` and `colour` can be used to customise text labels (either manually or from the data). Other aesthetics that can be added include:

- `label` defines the text displayed
- `angle` rotates the text
- `family` defines the font
- `fontface` can be changed to make text `"bold"` or `"italic"`

```
ggplot(data = penguins,
       aes(x = flipper_length_mm, y = body_mass_g,
           colour = species)) +
  geom_point() +
  scale_colour_brewer(palette = "Dark2") +
  labs(x = "Flipper length (mm)", y = "Body mass (g)",
       colour = "Species") +
  geom_text(label = "Bold text", y = 6000, x = 180, fontface = "bold",
            colour = "black") +
  geom_label(label = "Italicised \n text", y = 3500, x = 220,
             fontface = "italic", colour = "black")
```

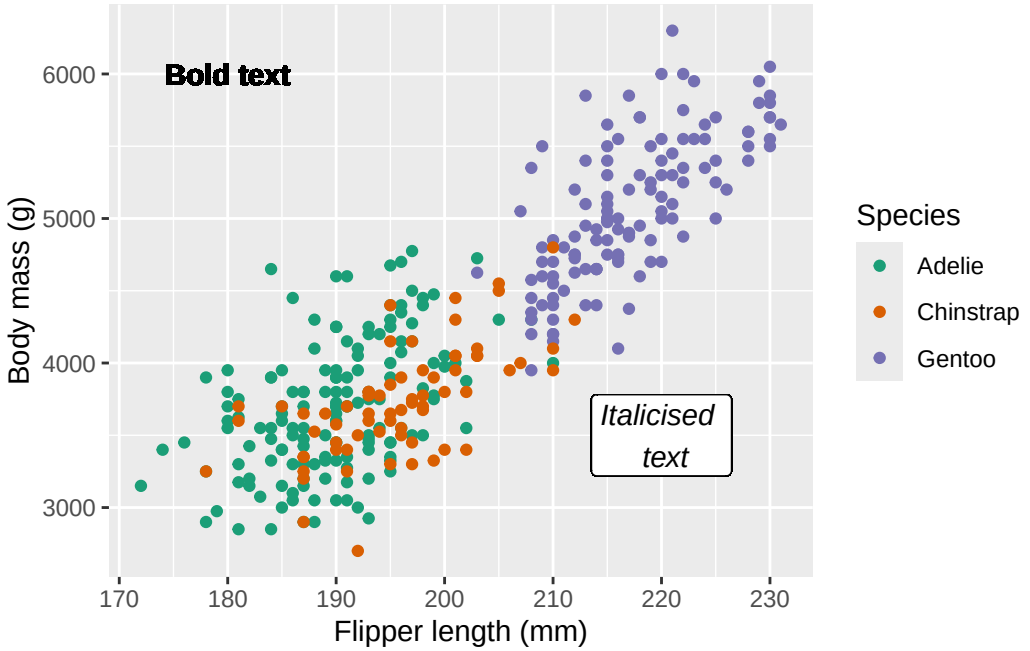


Figure 4.1: Scatterplot showing the size and mass of penguin species in the Palmer Archipelago, Antarctica.

> **Warning**
>
> As the `aes` wrapper has been specified at the `ggplot` layer, this will be applied to all `geoms`, including out text and labels. To overwrite this, we can specify the `colour` manually.

> **Hint**
>
> Where text and label positions are determined by data, you may also wish to utilise the `nudge_x` and `nudge_y` arguments to shift annotations and avoid overlap.

Adding text through `geoms` will work but notice that the annotations looks a little blurry on the text. This is because `geom` layers take the data into account and assume that you want the same number of layers/markings as observations in the data. This means that rather than adding a single text or label, `ggplot` is actually adding 342. To overcome this, we can use the `annotate` function instead.

4.2.2 Annotate function

The `annotate` function will add single `geom` layers to a visualisation while disregarding the rest of the data. This is useful when adding annotations such as text, labels, shapes or arrows. `annotate` functions require the same arguments as the corresponding `geom`, with an additional argument that specifies the `geom` we require.

For example, we can adapt the code used to create Figure 4.1 to utilise the `annotate` functions instead:

```
ggplot(data = penguins,
       aes(x = flipper_length_mm, y = body_mass_g,
           colour = species)) +
  geom_point() +
  scale_colour_brewer(palette = "Dark2") +
  labs(x = "Flipper length (mm)", y = "Body mass (g)",
       colour = "Species") +
  annotate("text", label = "Bold text", y = 6000, x = 180,
           fontface = "bold") +                                          (1)
  annotate("label", label = "Italicised \n text", y = 3500, x = 220,
             fontface = "italic")
```

(1) The `annotate` functions no longer require `colour` to be specified (unless we want to change it) as it is not generated using the data.

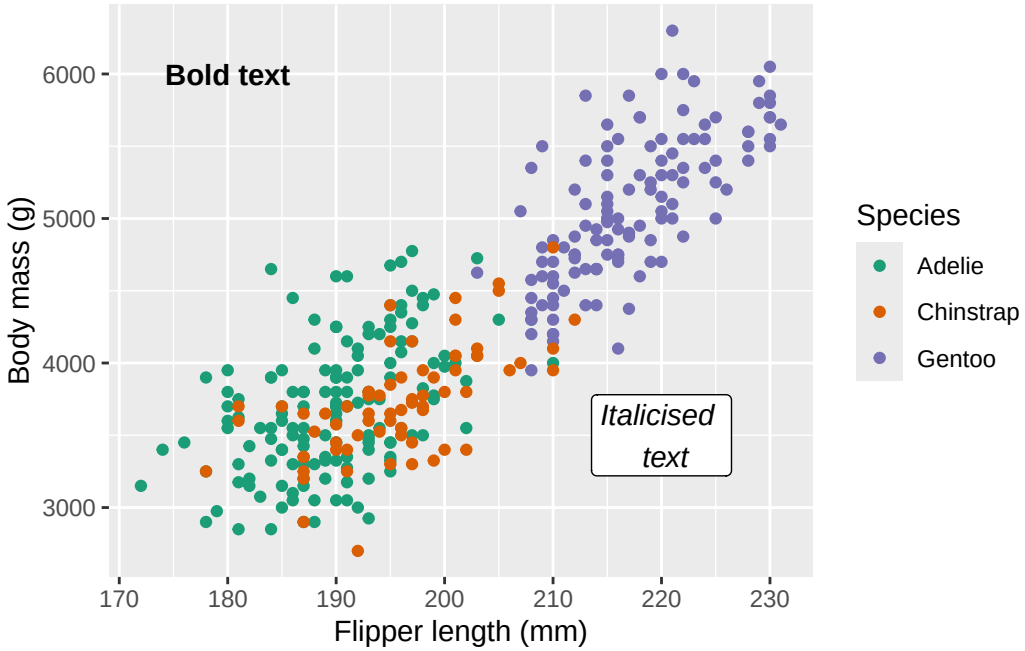


Figure 4.2: Scatterplot showing the size and mass of penguin species in the Palmer Archipelago, Antarctica.

Text labels can be combined with curves and arrows to make them clearer, using the `curve` or `segment geoms`. Both contain the optional argument `arrow` which adds an arrow to the curved line (this must be defined within the `arrow` function, which can be used to adjust the size or shape of the arrow):

```
ggplot(data = penguins,
       aes(x = flipper_length_mm, y = body_mass_g,
           colour = species)) +
  geom_point() +
  scale_colour_brewer(palette = "Dark2") +
  labs(x = "Flipper length (mm)", y = "Body mass (g)",
       colour = "Species") +
  annotate("curve", x = 185, y = 5500, xend = 195, yend = 4700,
           curvature = -.3, arrow = arrow(length = unit(.25, "cm"))) +
  annotate("text", x = 185, y = 5600, label = "Important penguin")
```

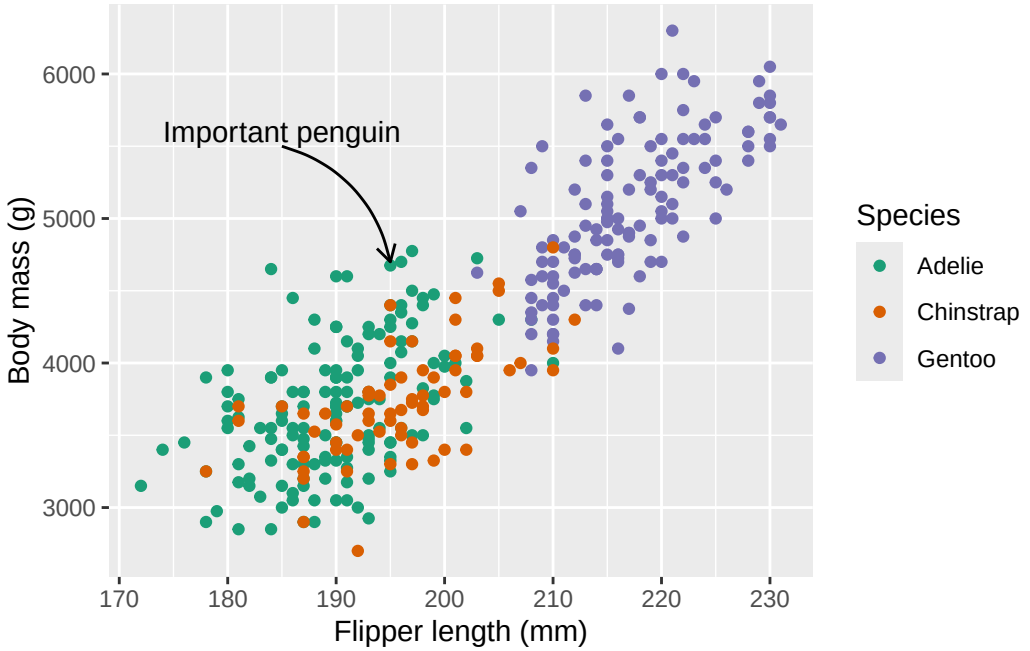


Figure 4.3: Scatterplot showing the size and mass of penguin species in the Palmer Archipelago, Antarctica.

Exercise 3

1. Using the `csp_2020` dataset, investigate the relationship between `sfa_2020` and `ct_total_2020`, and show whether that differs between regions. Ensure that this visualisation:

- Has appropriate axes, legend and plot titles
- Has annotations that explain interesting points and make these relationships clearer to a reader

> Exercise hint
>
> Include `region` in the `aes` wrapper. Consider adding a line of best fit per region to make these relationships clearer.
> This dataset has a clear outlier. Highlight this for readers.
> Your `ggplot` should contain `annotate` and `labs` layers.

2. Generate an alternative visualisation that investigates the relationship between `sfa_2020` and `ct_total_2020`, adding an annotation to highlight just the North West region

(`region == "NW"`). Add a label to the graph area that makes it clear what the highlighted points represent (rather than a legend).

Exercise hint

Add an extra `geom_point` layer to highlight the North West region.
Use `annotate` layers to define the highlighted point meaning in the graph area. If you are really stuck, check this ebook for inspiration.

5 Themes

We have already seen how data elements of a `ggplot` can be customised using `scale` functions. `theme` functions allow us to customise all other elements of the visualisation, including the plot background, title fonts and sizes, legend positioning, and gridlines.

There are many options that can be adjusted within the `theme` function (see the helpfile `?theme` for a complete list). Often, it is efficient to begin with a pre-built theme and tweak elements that do not suit our purpose.

5.1 Pre-built themes

There are 8 complete themes programmed in the `ggplot2` package. These are:

Each `theme` function can be added as a layer in a `ggplot` object. For other pre-built themes, check out the `ggthemes` package).

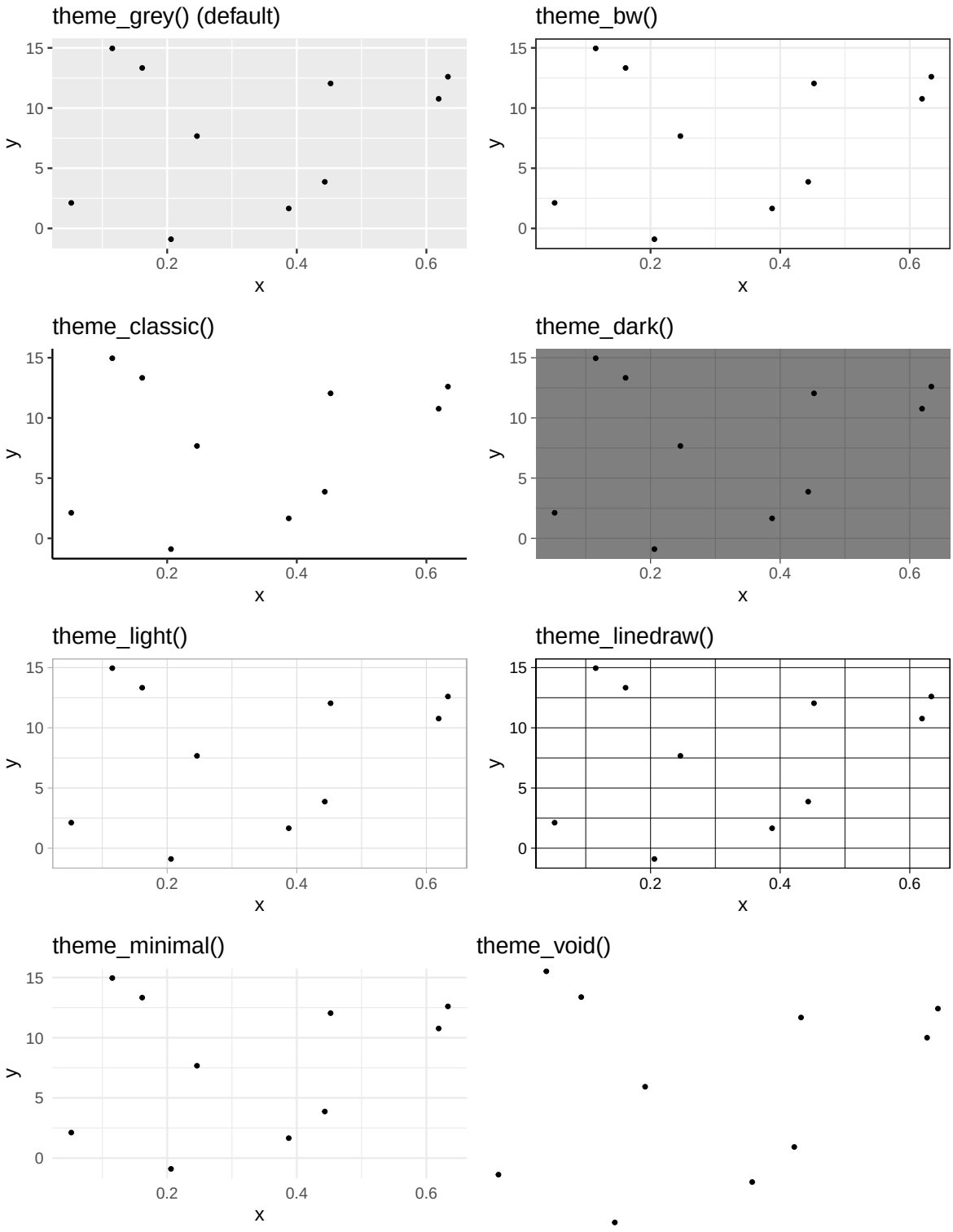
Style tip

Although pre-built `theme` functions do not require arguments to run, they all contain the optional argument `base_size` which set the default font size (defaulted to 11). To ensure visualisations are as accessible and inclusive as possible, ensure this is set to at least 12 for printed graphs or 36 for presentations.

5.2 Customising themes

Individual elements of a visualisation's theme can be customised within the `theme` functions. Many elements that can be customised using the `theme` require an `element` wrapper. This wrapper is determined by the type of object that we are customising, the four options are:

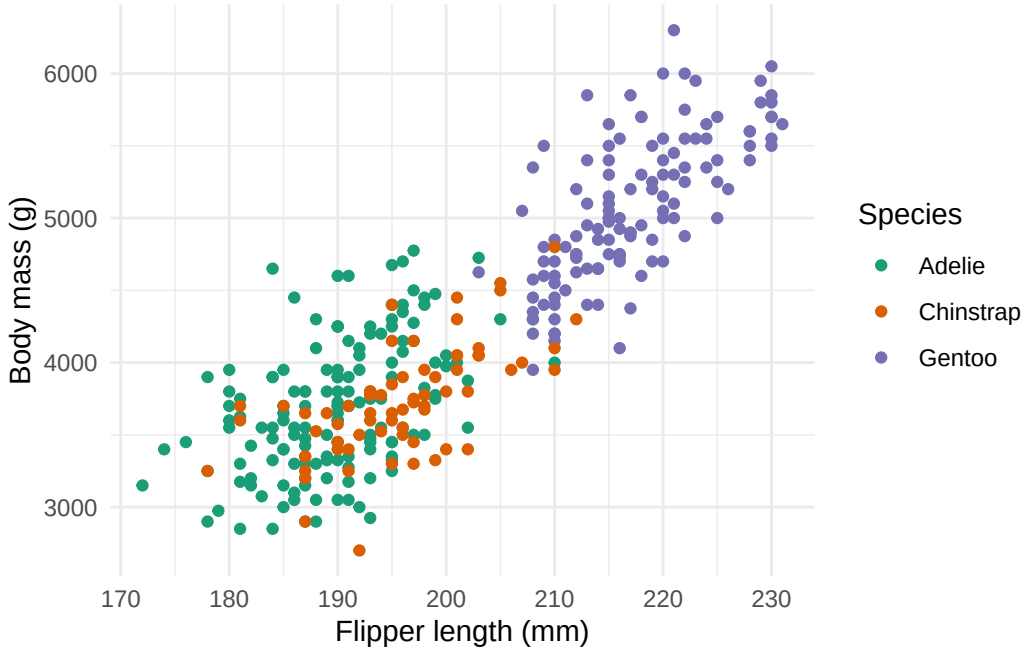
- `element_text` when customising text, e.g. axis titles and labels
- `element_rect` when customising backgrounds, e.g. the graph area
- `element_line` when customising lines, e.g. gridlines
- `element_blank` to remove elements


theme_grey() (default)
theme_bw()
theme_classic()
theme_dark()
theme_light()
theme_linedraw()
theme_minimal()
theme_void()
x
y
0
5
10
15
0.2
0.4
0.6

Elements that do not require these wrappers are often related to positioning. A common example of this is the `legend.position` argument which can be set to `"left"`, `"right"` (default), `"top"`, `"bottom"`, or removed using `"none"`.

For example, we will customise elements of the scatterplot in Figure 4.1 using the `theme` functions:

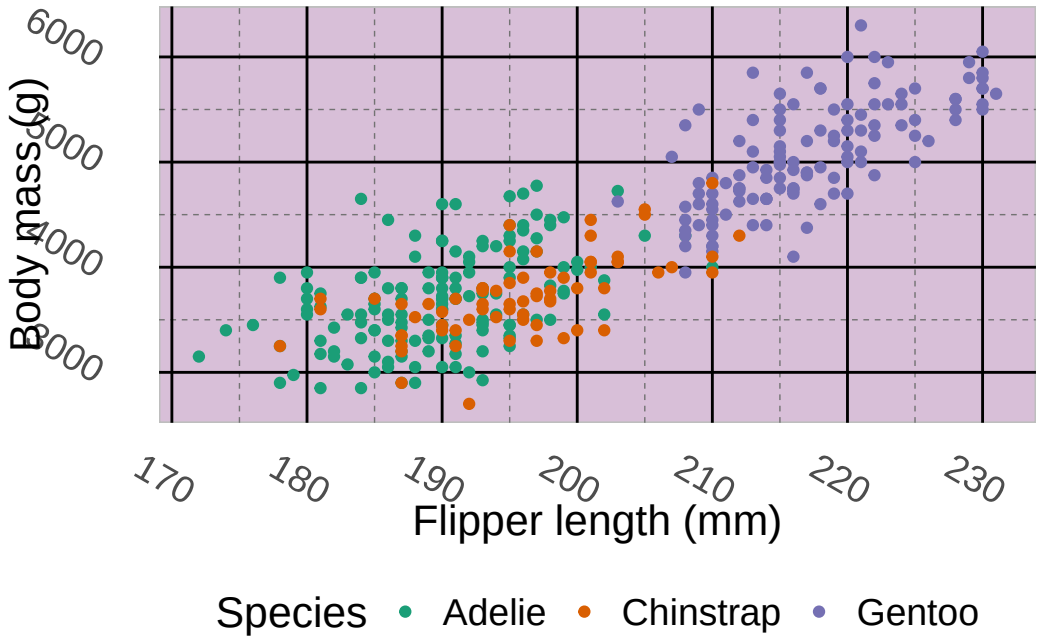
```
minimal_plot <- ggplot(data = penguins,
       aes(x = flipper_length_mm, y = body_mass_g,
           colour = species)) +
  geom_point() +
  scale_colour_brewer(palette = "Dark2") +
  labs(x = "Flipper length (mm)", y = "Body mass (g)",
       colour = "Species") +
  theme_minimal()

minimal_plot
```




```
minimal_plot +
  theme(legend.position = "bottom",                                    ①
        axis.title = element_text(size = 16),                          ②
        axis.text = element_text(size = 14, angle = -30),              ③
        legend.title = element_text(size = 16),
```

```
        legend.text = element_text(size = 14),
        panel.background = element_rect(fill = "thistle",                    ④
                                        colour = "grey"),
        panel.grid.major = element_line(colour = "black"),                   ⑤
        panel.grid.minor = element_line(colour = "grey45",                   ⑥
                                        linetype = "dashed"))
```

① Moves the legend to the bottom of the graph.
② Sets the axis title labels to size 16.
③ Sets the axis text (tick marks) to size 14 and rotates them 30 degrees clockwise.
④ Sets the plot background to thistle with a grey outline.
⑤ Adds black major grid lines.
⑥ Adds grey, dashed minor grid lines.



Style tip

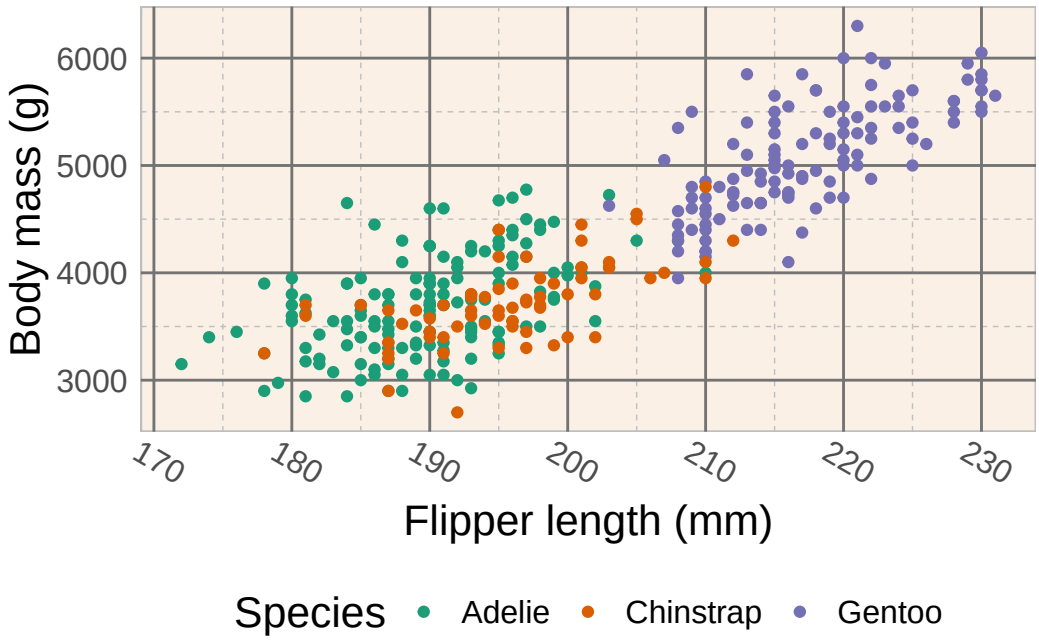
Good visualisations require a middle ground between overly minimal design, which can make interpretation difficult, and charts overloaded with clutter. The bold grid lines on this visualisations are a good example of where 'chart junk' can distract from the data.

Visualisations should strive to make the data are the most important part of the graphic, whilst ensuring there is sufficient context provided by non-data elements.

5.3 Creating a theme

One benefits of using `theme` functions is that visualisations will remain consistent in terms of their design. Custom themes can be saved as functions and added to `ggplot` objects in place of the in-built themes. For example,

```
theme_dataviz <- function() {
  theme_minimal() +
  theme(legend.position = "bottom",
        axis.title = element_text(size = 16),
        axis.text.x = element_text(size = 12, angle = -30,
                                   vjust = 0),
        axis.text.y = element_text(size = 12, hjust = 0),
        legend.title = element_text(size = 16),
        legend.text = element_text(size = 12),
        panel.background = element_rect(fill = "linen",
                                        colour = "grey"),
        panel.grid.major = element_line(colour = "grey45"),
        panel.grid.minor = element_line(colour = "grey75",
                                        linetype = "dashed"))
}

ggplot(data = penguins,
       aes(x = flipper_length_mm, y = body_mass_g,
           colour = species)) +
  geom_point() +
  scale_colour_brewer(palette = "Dark2") +
  labs(x = "Flipper length (mm)", y = "Body mass (g)",
       colour = "Species") +
  theme_dataviz()
```


Body mass (g)
6000
5000
4000
3000
170
180
190
200
210
220
230
Flipper length (mm)
Species
Adelie
Chinstrap
Gentoo

6 Faceting

Faceting allows us to divide a plot into subplots based on some grouping variable within the data. This allows us to show multiple variables in the same visualisation without risking overloading the plot and losing the intended message.

There are two approaches to faceting within `ggplot2`:

- `facet_wrap` separates observations into separate graphs based on a single variable. This variable is entered into the function with a `vars` wrapper, the number of rows or columns are specified using the `nrow` and `ncol` arguments.
- `facet_grid` produces a 2d grid of panels, where the rows and columns are defined by variables. Variables are included separated by `~` (`facet_grid(rows ~ cols)`).

The layout of these facets can be customised by specifying the `nrow` or `ncol` arguments.

```
ggplot(data = penguins,
       aes(x = flipper_length_mm, y = body_mass_g,
           colour = species)) +
  geom_point() +
  scale_colour_brewer(palette = "Dark2") +
  labs(x = "Flipper length (mm)", y = "Body mass (g)",
       colour = "Species") +
  facet_wrap(vars(sex)) +
  theme_light(base_size = 16)
```

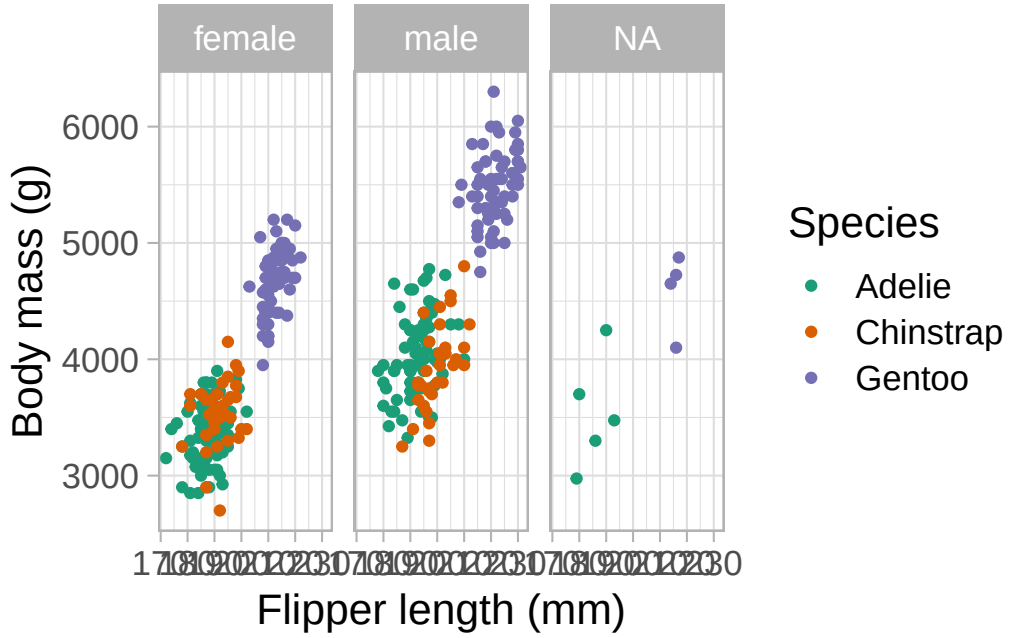


Figure 6.1: Scatterplot showing the relationship between body mass, flipper length and species of penguins in Antarctica, facetted by sex.

6.1 Customising facets

6.1.1 Dealing with missing data

As we can see from Figure 6.1, facet functions will treat missing values as a value and produce a facet for these observations. If we do not want to show these missing values, they would need to be removed from the data used to produce the plot:

```
penguins %>%
  filter(!is.na(sex)) %>%                                          ①
  ggplot(aes(x = flipper_length_mm, y = body_mass_g,
             colour = species)) +
  geom_point() +
  scale_colour_brewer(palette = "Dark2") +
  labs(x = "Flipper length (mm)", y = "Body mass (g)",
       colour = "Species") +
  facet_wrap(vars(sex)) +
  theme_light(base_size = 16) +
  theme(axis.text.x = element_text(angle = 45))                    ②
```

① Remove observations missing the `sex` variable.
② Rotate the x-axis labels to make them easier to read.

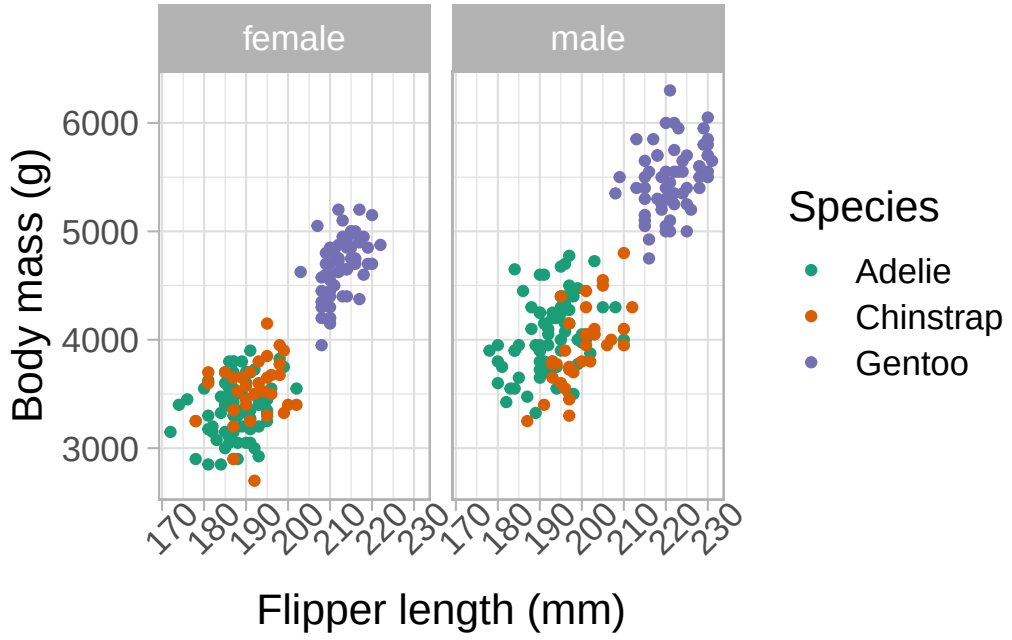


Figure 6.2: Scatterplot showing the relationship between body mass, flipper length and species of penguins in Antarctica, facetted by sex (with missing data removed).

6.1.2 Scales

One of the benefits of using facets when producing multiple graphs to make comparisons is that they are shown on equivalent scales by default. However, if this is not appropriate, this default can be changed using the `scales` argument. Scales can either be `"fixed"` (the default) or `"free"`, either across both axes or by a single dimension (`"free_x"` or `"free_y"`).

Style tip

Where facets are used to make comparisons across panels, `scales` should be set to `fixed`. If not, the differences in scales may not be immediately obvious to readers which could be misleading where like-for-like comparisons are not appropriate. These `scales` should only be set as `free` where panels show different outcomes (e.g. multiple time series across different variables).

6.1.3 Facet appearances

The appearance of facets, such as the colour of the labels and panels, can be customised within the `theme` function. Elements related to facets tend to begin `strip.`, for example `strip.background` changes the background colour, and `strip.placement` chooses whether facet labels should be added inside or outside the chart area:

```
penguins %>%
  filter(!is.na(sex)) %>%
  ggplot(aes(x = flipper_length_mm, y = body_mass_g,
             colour = species)) +
  geom_point() +
  scale_colour_brewer(palette = "Dark2") +
  labs(x = "Flipper length (mm)", y = "Body mass (g)",
       colour = "Species") +
  facet_wrap(vars(sex)) +
  theme_light(base_size = 16) +
  theme(axis.text.x = element_text(angle = 45),
        strip.background = element_rect(fill = "thistle"),
        strip.text = element_text(face = "bold", colour = "navy"))
```

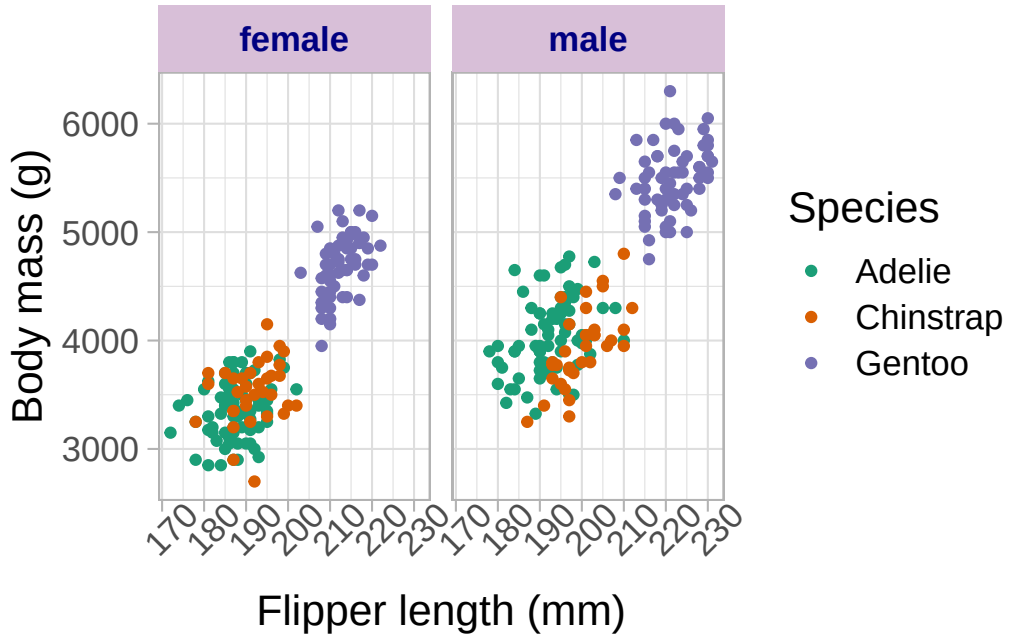


Figure 6.3: Scatterplot showing the relationship between body mass, flipper length and species of penguins in Antarctica, facetted by sex (with missing data removed).

6.2 Other options for multiple plots

There are various packages that allow multiple `ggplot2` objects to be shown on the same output. One of the most popular is the `patchwork` package. This is particularly useful where multiple points are necessary for investigating or communicating an aspect of the data that would overload a single plot.

For example, it we wanted to investigate the relationships between body mass, flipper length, species and bill length of the Palmer penguins, we would struggle to show this on a single plot. However, combining a scatterplot to investigate the relationship between body mass and flipper length, a box plot to compare the average bill length across species, and a histogram to show the different distribution of body mass across species, would give a clear message of these relationships.

We first have to create an object for each plot we would like to display:

```
p1 <- ggplot(penguins) +
  geom_point(aes(x = flipper_length_mm, y = body_mass_g,
                 colour = species)) +
  scale_colour_brewer(name = "Species", palette = "Dark2") +
  labs(x = "Flipper length (mm)", y = "Body mass (g)") +
  theme_light(base_size = 16)

p2 <- ggplot(penguins) +
  geom_boxplot(aes(x = species, y = bill_length_mm, fill = species)) +
  scale_fill_brewer(name = "Species", palette = "Dark2") +
  labs(x = "Species", y = "Bill length (mm)") +
  theme_light(base_size = 16) +
  theme(legend.position = "none")

p3 <- ggplot(penguins) +
  geom_histogram(aes(x  = body_mass_g, fill = species),
                 colour = "black", alpha = .75) +
  scale_fill_brewer(name = "Species", palette = "Dark2") +
  labs(x = "Count", x = "Body mass (g)") +
  theme_light(base_size = 16) +
  theme(legend.position = "none")
```

Using the `patchwork` package, `ggplot` objects can be displayed next to one another using the `+` symbol, and can be stacked on top of one another using the `/` symbol. These plots can be nested together by surrounding them by brackets. For example, if we wished to show the scatterplot on the first row, with the boxplot and histograms next to one another on the bottom, we would use the following code:

> **Warning**
>
> If you have never used the `patchwork` package, install it to your machine from CRAN using the following:
> `install.packages("patchwork")`
> If you have installed `patchwork`, ensure it is loaded into this section by adding the `library(patchwork)` command into your script.

```
p1 / (p2 + p3)
```

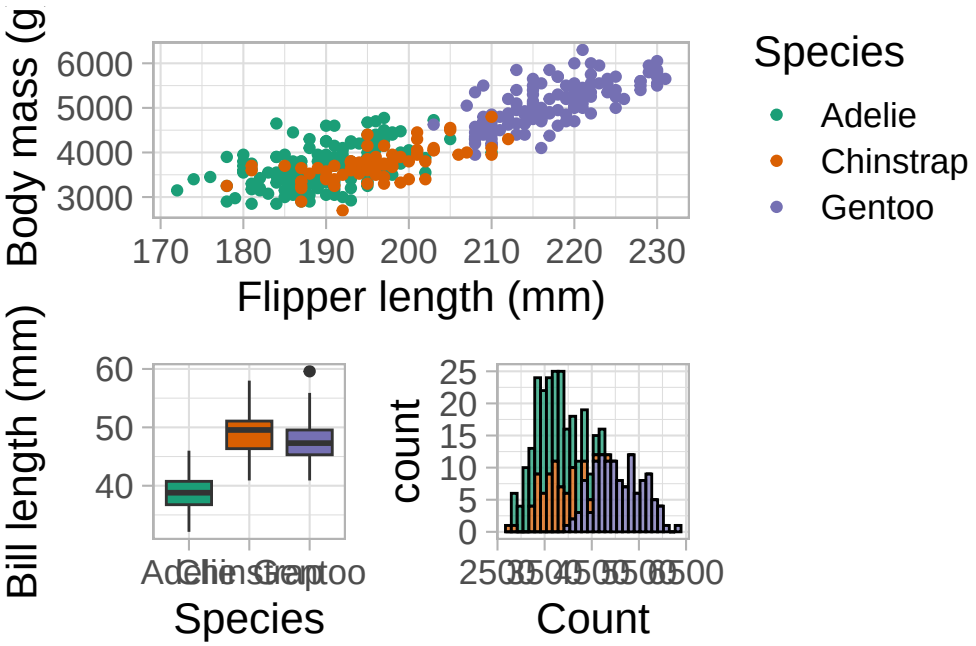


Figure 6.4: Plots investigating the relationship between penguins species and body mass, flipper length, and bill length

In this example, we removed the legend from the bottom two plots to avoid duplication. However `patchwork` contains options that control the layout of outputs, including 'collecting' and condensing these into a single legend (ssee the patchwork webpage for more details).

6.3 Exercise 4

The `csp_201520.csv` data file contains information about core spending power in English local authorities between 2015 and 2020 (for more information about these variables, see the data

description appendix). Using this information, generate a plot that investigates the change in spending power over this period, and whether this is different between regions of England. Adapt your visualisation to ensure that it is informative, compelling and accessible. This can include:

- Adding an appropriate title and caption with data source information
- Ensuring all colours and text are legible and inclusive
- Adding labels to commmunicate important findings to the reader
- Adjusting the theme to ensure text is large enough, values are clear but not overwhelmed by 'chart junk'

Exercise hint

There are a lot of options that we could use to display this information. Temporal data are often displayed using line graphs or, if we wished to show the decomposition of spending power by the different variables, a stacked area plot may be of interest.
This data does not contain the overall total spend per local authority so if this is something you would like to display, you would need to create this variable using the `mutate` function.
Differences between regions could be added to a visualisation by using a different colour or by using facets if these differences are not clear on a single graph.
Check that any colours are distinct, even to those with different forms of colour blindness.
Ensure that text is clear and large enough. If there are multiple messages you would like to display, consider using `patchwork` to arrange multiple graphs on a single output.

Data description

What is 'CSP'?

The data we will be using throughout this course relates to the Core Spending Power (CSP) of English local authorities. This is a measure of the resources available to local authorities in England to fund service delivery. The CSP is broken down into several components, presented as variables in the data. These components include:

- Settlement Funding Assessment (`sfa`)
- Compensation for under-indexing the business rates multipler (`under_index`)
- Income from council tax (`ct_total`)
- New Homes Bonus (`nhb`)
- Rural Services Delivery Grant (`rsdg`)

Spending power is given in millions of pounds (£). The data were provided by the UK government's Department for Levelling Up, Housing and Communities. Full guidance on the data can be found on the Department's website. A brief description of the variables included in the data are given below.

Descriptions of variables

Identifier variables

Each dataset contains a unique identifier code variable, `ons_code`. This is a code given by the Government's Office for National Statistics (ONS), and is used to join different datasets. There is also an `authority` variable which contains the local authority name (to see where each local authority lies on a map, you can visit the Government's geoportal website).

Regions of England

In addition to each local authority's unique code and name, we are given the region that they lie within. England is separated into 9 regions (shown on this map) which are given as acronyms in the data. These are:

- `L` = London
- `NW` = North West
- `NE` = North East
- `YH` = Yorkshire and the Humber
- `WM` = West Midlands
- `EM` = East Midlands
- `EE` = East England
- `SW` = South West
- `SE` = South East

Settlement Funding Assessment (SFA)

The Settlement Funding Assessment (`sfa` in the data) is the baseline funding level of local authorities, and includes the Revenue Support Grant (a central government grant given to local authorities).

Under-indexing business rate multipliers

The `under_index` variable is given to compensate local authorities that under-indexed business rate multipliers in previous years (i.e. those that used a measure of inflation that was lower than that should have been used).

Council tax

Council tax (`ct_total`) is the income made by each local authority from council tax. In England, the amount of council tax charged to residents is set by each local authority to make up additional revenue needed to cover planned spending.

New Homes Bonus

The `nhb` variables is the funding received as part of the New Homes Bonus, a government inncentive to encourage local authorities to promote new housing delevopment.

Rural Services Delivery Grant

The `rsdg` variable is funding received as part of the Rural Services Delivery Grant, provided to rural councils to recognise additional costs in these areas.

Further reading

One of the benefits of using `ggplot2` to produce outputs is the huge range of free resources available online. Below is a list of online resources that I would recommend that help continue your `ggplot2` learning journey or provide guidance on data visualisation in general.

- ggplot2: Elegant Graphics for Data Analysis
- Design principles for data visualisation
- RSS's Best Practices for Data Visualisation
- From Data to Viz
- Fundamentals of Data Visualization
- The R-graph gallery
- R colour palette finder
- ggplot2 extensions

A Exercise solutions

A.1 Exercise 1

Question 1

Load the `CSP_2020.csv` file into R and save it as an object named `csp_2020`. Ensure that all variables are correctly specified by type.

Solution

To load in a `csv` file, we use the `read_csv` function and save the data as an object:

```
csp_2020 <- read_csv("data/CSP_2020.csv")
```

Style tip

Object names should only contain lower case letters, numbers and underscores `_`. Ensure names are clear and concise. Names must begin with a letter.

Next, we must ensure the data has been read correctly. This includes checking the variables and observations have been organised correctly, checking the names of variables are appropriate, and checking the type of variables are correctly specified:

```
View(csp_2020)                                                    ①

str(csp_2020)                                                     ②
```

① `View` produces a preview of the dataset that appears as a tab by the script files.
② `str` returns information about the structure of an object, including its type, and for data frames, the variable names and types.

```
spc_tbl_ [315 x 8] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ ons_code        : chr [1:315] "E07000223" "E07000026" "E07000032" "E07000224" ...
 $ authority       : chr [1:315] "Adur" "Allerdale" "Amber Valley" "Arun" ...
 $ region          : chr [1:315] "SE" "NW" "EM" "SE" ...
 $ sfa_2020        : num [1:315] 1.77 3.85 3.23 3.67 4.08 ...
 $ under_index_2020: num [1:315] 0.0708 0.1465 0.1292 0.147 0.1557 ...
 $ ct_total_2020   : num [1:315] 6.53 5.4 6.85 11.61 6.42 ...
 $ nhb_2020        : num [1:315] 0.0881 0.6061 1.5786 2.2949 1.1547 ...
 $ rsdg_2020       : num [1:315] 0 0.326 0 0 0 ...
 - attr(*, "spec")=
  .. cols(
  ..   ons_code = col_character(),
  ..   authority = col_character(),
  ..   region = col_character(),
  ..   sfa_2020 = col_double(),
  ..   under_index_2020 = col_double(),
  ..   ct_total_2020 = col_double(),
  ..   nhb_2020 = col_double(),
  ..   rsdg_2020 = col_double()
  .. )
 - attr(*, "problems")=<externalptr>
```

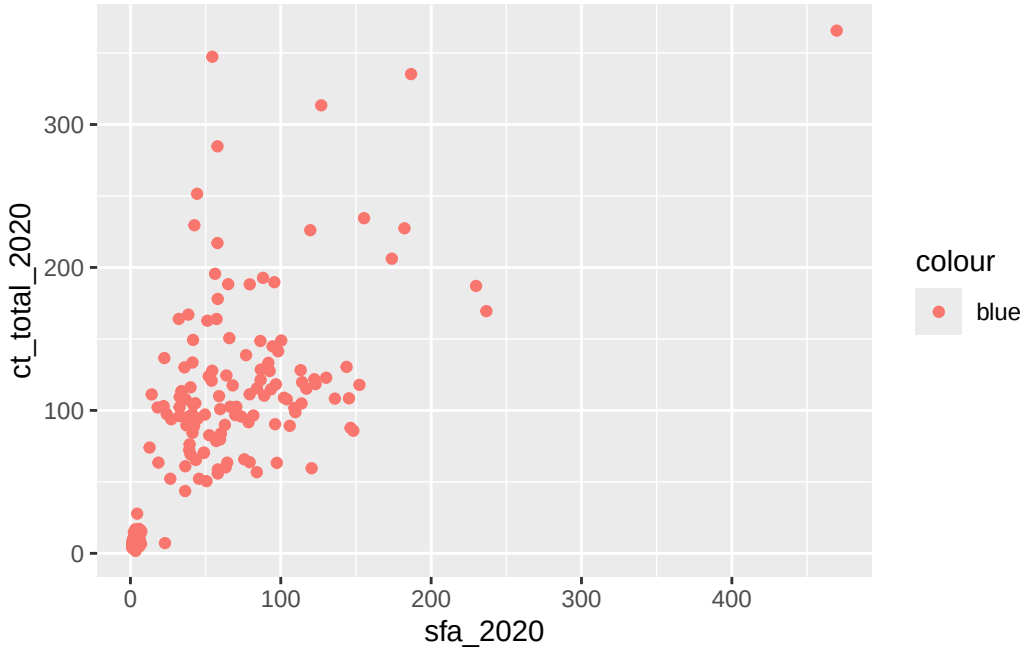
All variables appear to be correctly specified besides the `region` variable which is a grouped category, or `factor` in R. To ensure this is correctly specified, we should convert this variable. It is possible to assign order to `factor` variables. Without assigning an order, all outputs including visualisations, will show regions in alphabetical order. Here, we may with to set London as the control region, followed by regions ordered from north to south:

```
csp_2020 <- csp_2020 %>%
  mutate(region_fct = factor(region,
                             levels = c("L", "NW", "NE", "YH", "WM",
                                        "EM", "EE", "SW", "SE")))
```

Question 2

The following code was intended to produce a scatterplot showing the relationship between the `sfa_2020` and `ct_total_2020` variables with the points in blue. Debug the code to fix the problem:

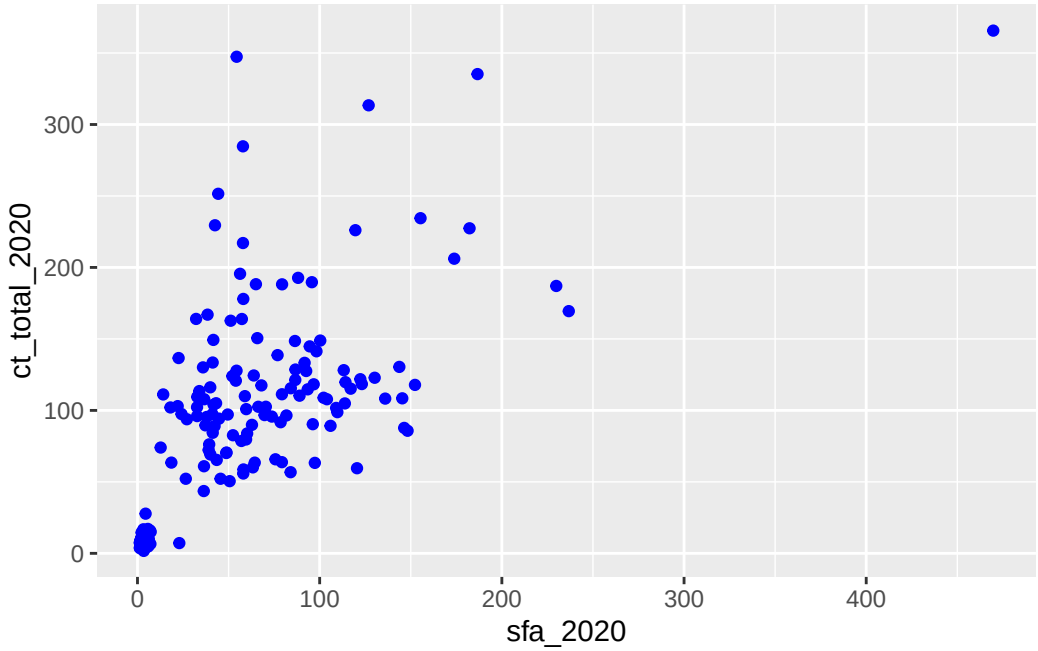
```
ggplot(data = csp_2020) +
  geom_point(aes(x = sfa_2020, y = ct_total_2020, colour = "blue"))
```



Solution

As the colour is specified manually, rather than by a variable in the data, it must be placed outside the `aes` wrapper:

```
ggplot(data = csp_2020) +
  geom_point(aes(x = sfa_2020, y = ct_total_2020), colour = "blue")
```



> **Note**
>
> Aesthetic markings (defined in the `aes` wrapper) can be included in the `ggplot` function if they are constant for each layer, or separately within each `geom` function.

Question 3

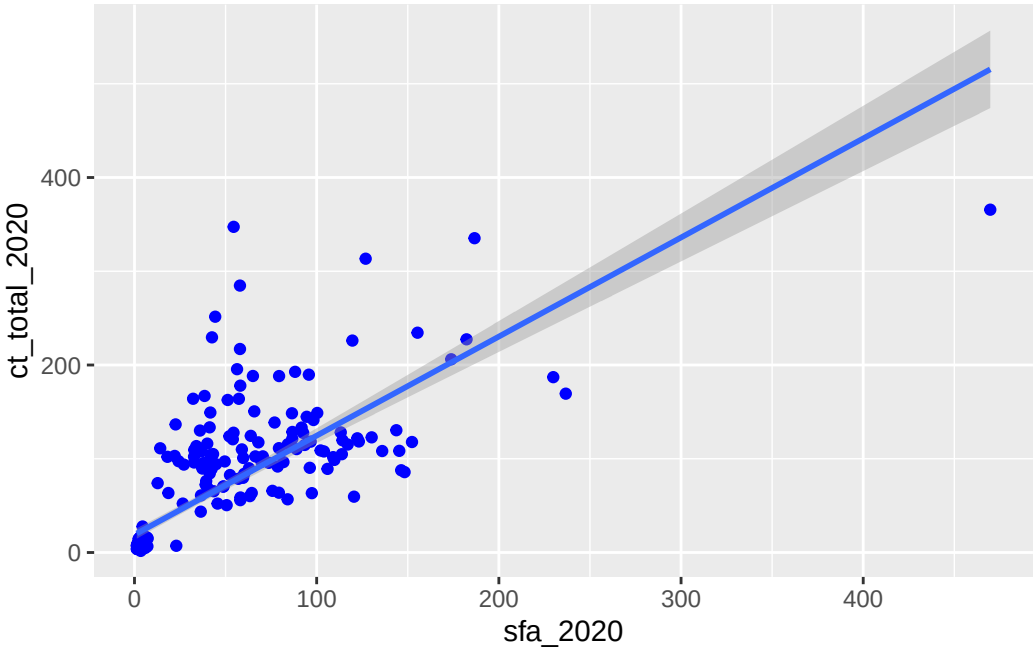
Add a straight line of best fit to the scatterplot and interpret the result. What is the shaded area surrounding the line and how can we remove it?

Solution

Lines of best fit can be added using the `geom_smooth` function. Be sure to set `method = "lm"` to fit a straight line:

```
ggplot(data = csp_2020, aes(x = sfa_2020, y = ct_total_2020)) +
  geom_point(colour = "blue") +
  geom_smooth(method = "lm")
```

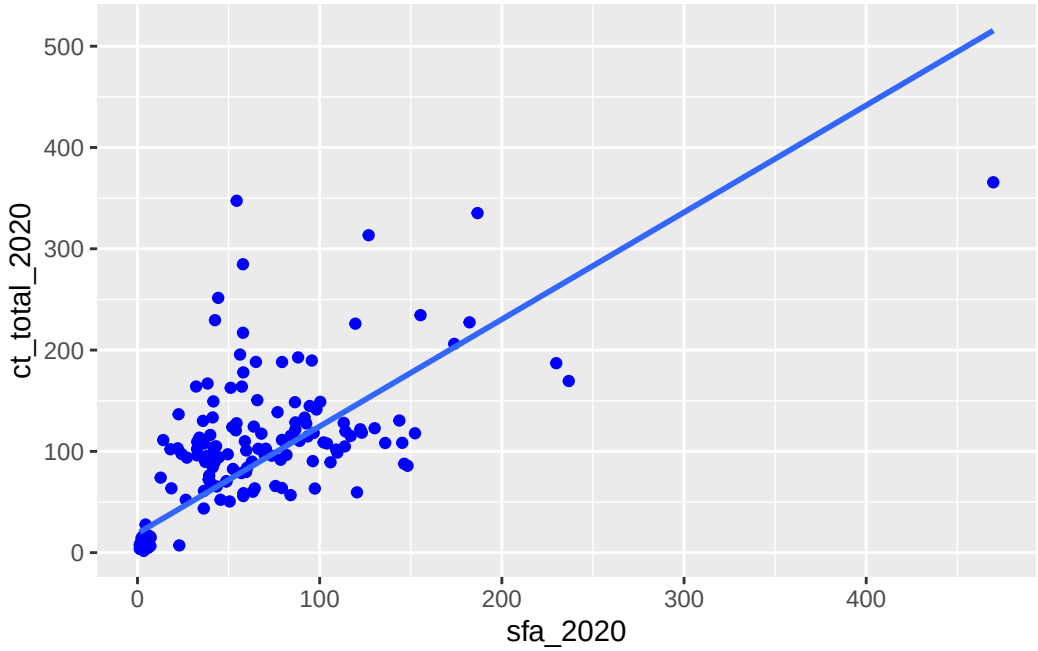
```
`geom_smooth()` using formula = 'y ~ x'
```



The shaded area around the line represents the 95% confidence interval (see the `se` argument of the helpfile `?geom_smooth`). This can be removed by changing the default setting of this argument:

```
ggplot(data = csp_2020, aes(x = sfa_2020, y = ct_total_2020)) +
  geom_point(colour = "blue") +
  geom_smooth(method = "lm", se = FALSE)
```

```
`geom_smooth()` using formula = 'y ~ x'
```



Question 4

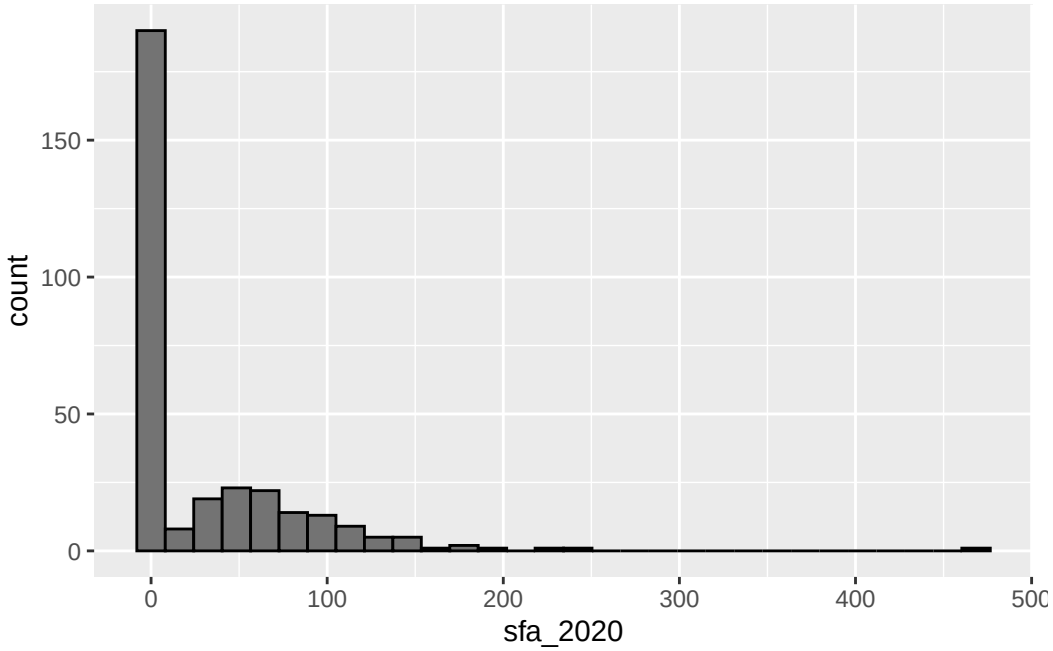
Use an appropriate visualisation to check the distribution of the `sfa_2020` variable. Interpret this visualisation.

Solution

Histograms are used to investigate the distribution of numeric variables:

```
ggplot(data = csp_2020) +
  geom_histogram(aes(x = sfa_2020),                                    ①
                 colour = "black", fill = "grey45")                    ②
```

① `geom_histogram` only requires `x` to be specified as the y-axis is a count.
② Optional customisation to make the plot nicer to look at!



The `sfa_2020` variable is not normally distributed, it is positively skewed (also known as right-skewed and upwardly skewed). The majority of local authorities had relatively low values. There is one very high outlier, with an SFA of almost £500 million. We can learn more about this observation by extracting it from the data:

```
filter(csp_2020, sfa_2020 > 400)
```

```
# A tibble: 1 x 9
  ons_code  authority  region sfa_2020 under_index_2020 ct_total_2020 nhb_2020
  <chr>     <chr>      <chr>     <dbl>            <dbl>         <dbl>    <dbl>
1 E08000025 Birmingham WM         470.             14.4          366.     7.24
# i 2 more variables: rsdg_2020 <dbl>, region_fct <fct>
```

The outlying value is Birmingham. This is a large city in England and one of the most highly populated local authorities in Europe! As this local authority contains such a high population, it makes sense that it would receive more funding than smaller local authorities. Therefore, this outlier is not an error and should not be removed from analysis, although we should make a note of it as it may skew results.

A.2 Exercise 2

A.2.1 Question 1

Use an appropriate visualisation to compare the total core spending power in local authorities 2020 across regions of England. Highlight the London region in a different colour to the other regions to make it stand out more.

Solution

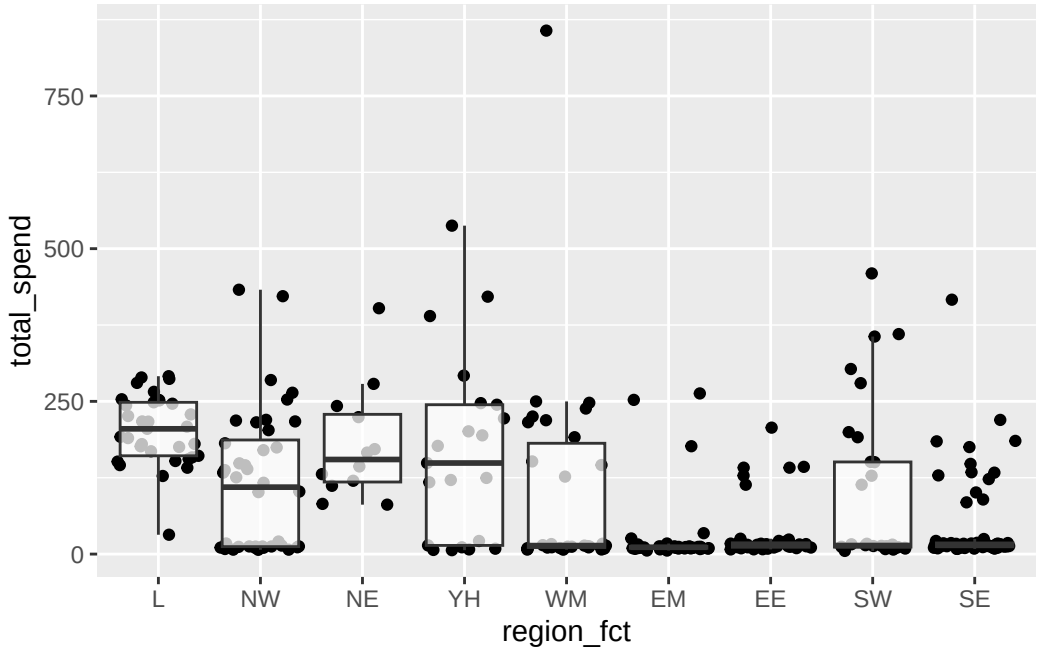
The current data set does not contain a the total core spending power. To create this, we can use the `mutate` function.

> Hint
>
> To avoid typing each of the variable names out, use the `names(csp_2020)` function to return them in the console, then copy and paste them.

```
csp_2020 <- mutate(csp_2020,
                   total_spend = sfa_2020 + under_index_2020 +
                   ct_total_2020 + nhb_2020 + rsdg_2020)
```

A dot plot or boxplot would be appropriate to compare the total CSP between regions. To show as much information as possible, we can include both on the same graph:

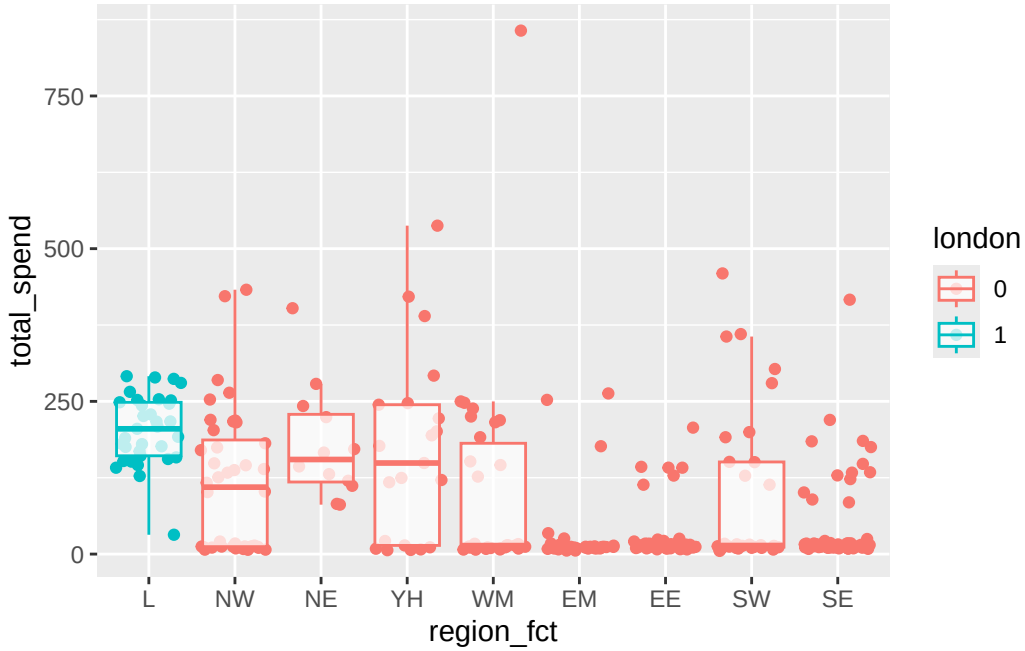
```
ggplot(data = csp_2020,
       aes(x = region_fct, y = total_spend)) +
  geom_jitter() +
  geom_boxplot(alpha = .75, outliers = FALSE)
```



To highlight the London region, we can add a variable to the data that takes the value 1 for London, and 0 otherwise. This can then be included in the visualisation within the `aes` wrapper.

> **Hint**
>
> Where a variable is created just for a visualisation, this can be carried out in a single process using pipes %>%. The `data` argument is no longer required as it is specified using the pipe.

```
mutate(csp_2020,
       london = factor(if_else(region == "L", 1, 0))) %>%
ggplot(aes(x = region_fct, y = total_spend, colour = london)) +
  geom_jitter() +
  geom_boxplot(outliers = FALSE, alpha = .75)
```

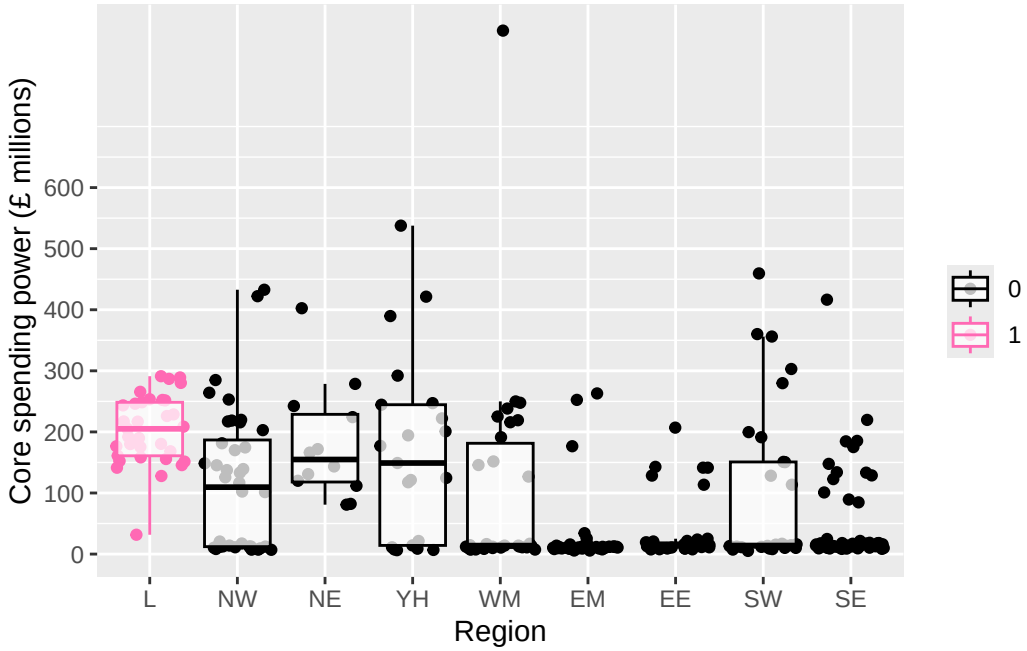


A.2.2 Question 2

Customise the graph above to ensure the axes are labelled appropriately. Add axis breaks every £100million on the y-axis.

Solutions

```
mutate(csp_2020,
       london = factor(if_else(region == "L", 1, 0))) %>%
ggplot(aes(x = region_fct, y = total_spend, colour = london)) +
  geom_jitter() +
  geom_boxplot(outliers = FALSE, alpha = .75) +
  scale_x_discrete(name = "Region") +
  scale_y_continuous(name = "Core spending power (£ millions)",
                     breaks = seq(0, 600, by = 100)) +
  scale_colour_manual(name = "", values = c("black", "hotpink"))
```



A.3 Exercise 3

A.3.1 Question 1

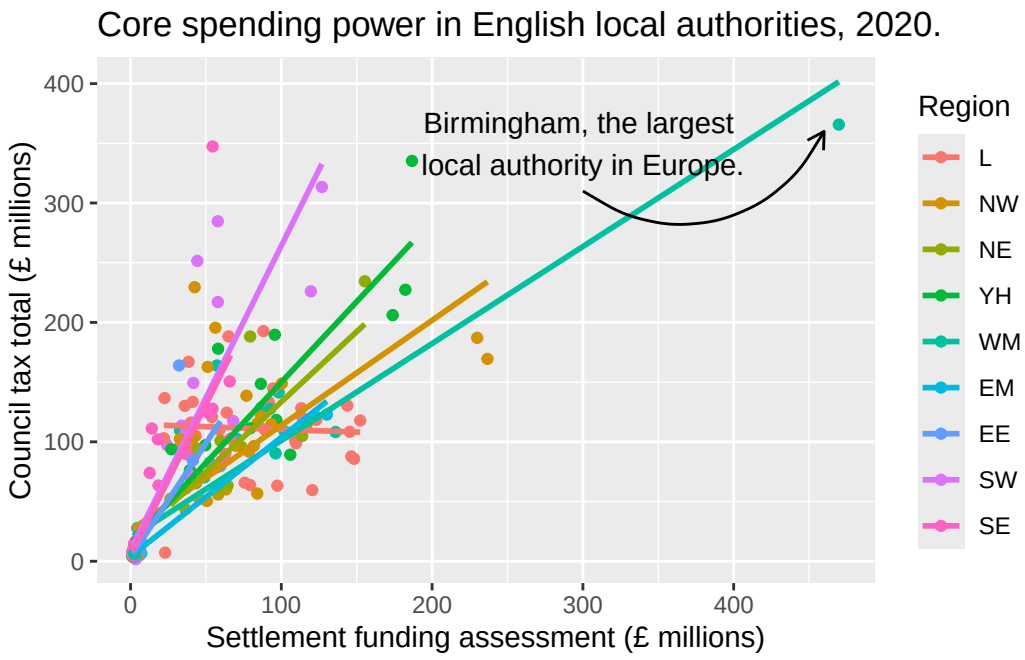
Using the `csp_2020` dataset, investigate the relationship between `sfa_2020` and `ct_total_2020`, and show whether that differs between regions. Ensure that this visualisation:

- Has appropriate axes, legend and plot titles
- Has annotations that explain interesting points and make these relationships clearer to a reader

Solution

A scatterplot is the most appropriate method of investigating the relationship between continuous variables. Include region as a colour. Add lines of best fit to make these relationships clearer. Add a `labs` layer to add appropriate titles. Add an annotation to explain the outlier (Birmingham).

```
ggplot(data = csp_2020,
       aes(x = sfa_2020, y = ct_total_2020, colour = region_fct)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  labs(title = "Core spending power in English local authorities, 2020.",
       x = "Settlement funding assessment (£ millions)",
       y = "Council tax total (£ millions)",
       colour = "Region") +
  annotate("text", x = 300, y = 350,
           label = "Birmingham, the largest \nlocal authority in Europe.") +
  annotate("curve", x = 300, y = 310, xend = 460, yend = 360,
           arrow = arrow(length = unit(.25, "cm")))
```



A.4 Question 2

Generate an alternative visualisation that investigates the relationship between `sfa_2020` and `ct_total_2020`, adding an annotation to highlight just the North West region (`region == "NW"`). Add a label to the graph area that makes it clear what the highlighted points represent (rather than a legend).

Solution

Points in the North West can be highlighted by adding coloured points beneath these observations. An annotation can then be added to the graph area with a highlighted point and "North West" text label:

```
ggplot(data = csp_2020,
       aes(x = sfa_2020, y = ct_total_2020)) +
  geom_point(data = filter(csp_2020, region == "NW"),
             size = 3, colour = "springgreen") +
  geom_point() +
    labs(title = "Core spending power in English local authorities, 2020.",
       x = "Settlement funding assessment (£ millions)",
       y = "Council tax total (£ millions)") +
  annotate("point", x = 300, y = 100, size = 3, colour = "springgreen") +
  annotate("point", x = 300, y = 100) +
  annotate("text", x = 350, y = 100, label = "North West",
           size = 5)
```