Data visualisation with ggplot2 - Answers

This lesson is adapted from the Data Analysis and visualisation in R for Ecologists 2 lesson. (https://datacarpentry.github.io/R-ecology-lesson/visualizing-ggplot.html)

Questions

- How do you make plots using R?
- How do you customize and modify plots?

Objectives

- Produce scatter plots and boxplots using `ggplot2`.
- Represent data variables with plot components.
- Modify the scales of plot components.
- Iteratively build and modify `ggplot2` plots by adding layers.
- Change the appearance of existing `ggplot2` plots using premade and customized themes.
- Describe what faceting is and apply faceting in `ggplot2`.
- Save plots as image files.

This is an R Markdown (http://rmarkdown.rstudio.com/) Notebook. When you execute code within the notebook, the results appear beneath the code.

You can execute code chunks by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing `Ctrl + Shift + Enter` or OS X: `Cmd + Shift + Enter`. To execute a specific line within a code chunk, place your cursor on that line and press `Ctrl + Enter` or OS X: `Cmd + Enter`.

```
# try running this code chunk

3 + 2
```

```
## [1] 5
```

```
3 * 2
```

```
## [1] 6
```

You can add a new chunk of code in your language of choice by clicking the *Insert Chunk* button on the toolbar or by pressing `Ctrl + Alt + I` or OS X: `Cmd + Option + I`.

Tip - View source or visual mode

The R Markdown notebook (e.g. *'visualisation_ggplot_answers.Rmd'*) can be viewed or edited in either in *'Source'* mode which shows the raw Markdown commands, or in the *'Visual'* mode which shows how the Markdown will look in its formatted form. You can switch between the *'Source'* and *'Visual'* modes at any time using the tabs at the top of the notebook.

Tip - Preview

When you save the notebook, an HTML file containing the code and *output* will be saved alongside it. To preview the HTML file in the 'Viewer' tab click the *Preview* button or press `Ctrl + Shift + K` or OS X: `Cmd + Shift + K`. The preview shows you a rendered HTML copy of the current contents of the editor. Consequently, *Preview* does not run any R code chunks. Instead, the output of the chunk from when it was last run in the editor is displayed. There are other more sophisticated ways to display your notebook, such as using the *Knit* command.

1. Setup

We are going to be using **functions** from the **`ggplot2`** package to create visualisations of data. Functions are predefined bits of code that automate more complicated actions. R itself has many built-in functions, but we can access many more by loading other packages of functions and data into R.

We start by loading the required packages that you installed during the setup. **`ggplot2`** is included in the **`tidyverse`** package, so we'll load the tidyverse.

If you do not have the tidyverse installed, you can run `install.packages("tidyverse")` in the console window.

It is a good practice not to put `install.packages()` into a script. This is because every time you run that whole script, the package will be reinstalled, which is typically unnecessary. You want to install the package to your computer once, and then load it with `library()` in each script where you need to use it

```
library(tidyverse)
```

```
## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
## ✔ dplyr     1.1.4     ✔ readr     2.1.5
## ✔ forcats   1.0.0     ✔ stringr   1.5.1
## ✔ ggplot2   3.5.1     ✔ tibble    3.2.1
## ✔ lubridate 1.9.4     ✔ tidyr     1.3.1
## ✔ purrr     1.0.4
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflic
ts to become errors
```

Later we will learn how to read data from external files into R, but for now we are going to use a clean and ready-to-use dataset that is provided by the `ratdat` data package. To make our dataset available, we need to load this package too.

```
library(ratdat)
```

The `ratdat` package contains data from the Portal Project (https://github.com/weecology/PortalData), which is a long-term dataset from near town Portal, Arizona (USA), in the Chihuahuan desert.

We will be using a dataset called `complete_old`, which contains older years of survey data. Let's try to learn a little bit about the data. We can use a `?` in front of the name of the dataset, which will display the help page for the data in the 'Help' window

Tip - Comments

Lines starting with `#` are used to indicate a comment that is not interpreted when the code chunk is run. It's a good idea to use comments to organize your code or clarify what you are doing.

```
# help page

?complete_old
```

Here in the 'Help' window we can read descriptions of each variable in our data.

To actually take a look at the data, we can use the `View()` function to open an interactive viewer, which behaves like a simplified version of a spreadsheet program. It's a handy function, but somewhat limited when trying to view large datasets.

Tip - Naming code chunks

You'll see we have started naming some of our our 'code chunks', like 'data-help'. The name is optional but if used, every code chunk needs a distinct name. The advantage of giving each chunk a name is that it will be easier to understand where to look for errors, should they occur. Also, any figures that are created will be given names based on the name of the code chunk that produced them.

```
# view dataset

View(complete_old)
```

If you hover over the tab for the interactive `View()`, you can click the "x" that appears, which will close the tab.

We can find out more about a dataset by using the `str()` function to examine the structure of the data.

```
# see structure of dataset

str(complete_old)
```

```
## tibble [16,878 × 13] (S3: tbl_df/tbl/data.frame)
##  $ record_id      : int [1:16878] 1 2 3 4 5 6 7 8 9 10 ...
##  $ month          : int [1:16878] 7 7 7 7 7 7 7 7 7 7 ...
##  $ day            : int [1:16878] 16 16 16 16 16 16 16 16 16 16 ...
##  $ year           : int [1:16878] 1977 1977 1977 1977 1977 1977 1977 1977 1977 197
7 ...
##  $ plot_id        : int [1:16878] 2 3 2 7 3 1 2 1 1 6 ...
##  $ species_id     : chr [1:16878] "NL" "NL" "DM" "DM" ...
##  $ sex            : chr [1:16878] "M" "M" "F" "M" ...
##  $ hindfoot_length: int [1:16878] 32 33 37 36 35 14 NA 37 34 20 ...
##  $ weight         : int [1:16878] NA NA NA NA NA NA NA NA NA NA ...
##  $ genus          : chr [1:16878] "Neotoma" "Neotoma" "Dipodomys" "Dipodomys" ...
##  $ species        : chr [1:16878] "albigula" "albigula" "merriami" "merriami" ...
##  $ taxa           : chr [1:16878] "Rodent" "Rodent" "Rodent" "Rodent" ...
##  $ plot_type      : chr [1:16878] "Control" "Long-term Krat Exclosure" "Control"
"Rodent Exclosure" ...
```

`str()` will tell us how many observations/rows (obs) and variables/columns we have, as well as some information about each of the variables. We see the name of a variable (such as year), followed by the kind of variable ('int' for integer, 'chr' for character), and will display the first 10 entries in that variable. We will talk more about different data types and structures later on.

2. Plotting with `ggplot2`

`ggplot2` is a powerful package that allows you to create complex plots from tabular data (ie data that in a table format with rows and columns). The **gg** in **`ggplot2`** stands for "grammar of graphics", as the package uses consistent vocabulary to create plots of widely varying types. Therefore, we only need small changes to our code if the underlying data changes, or we decide to make a box plot instead of a scatter plot. This approach helps you create publication-quality plots with minimal adjusting and tweaking.

`ggplot2` is part of the **`tidyverse`** series of packages, which tend to like data in the "long" or "tidy" format, which means that each column represents a single variable, and each row represents a single observation. Well-structured data will save you lots of time making figures with **`ggplot2`** . For now, we will use data that are already in this format. We start learning R by using **`ggplot2`** because it relies on concepts that we will need when we talk about data transformation in the next lessons.

`ggplot` plots are built step by step by adding new layers, which allows for extensive flexibility and customization of plots.

Tip - Indentating code

Some languages, like Python, require certain spacing or indentation for code to run properly. This isn't the case in R, so if you see spaces or indentation in the code from this lesson, it is to improve readability.

2.1 Basic template for `ggplot`

To build a plot, we will use a basic template that can be used for different types of plots:

```
ggplot(data = <DATA>, mapping = aes(<MAPPINGS>)) +  <GEOM_FUNCTION>()
```

2.2 Bind a data frame

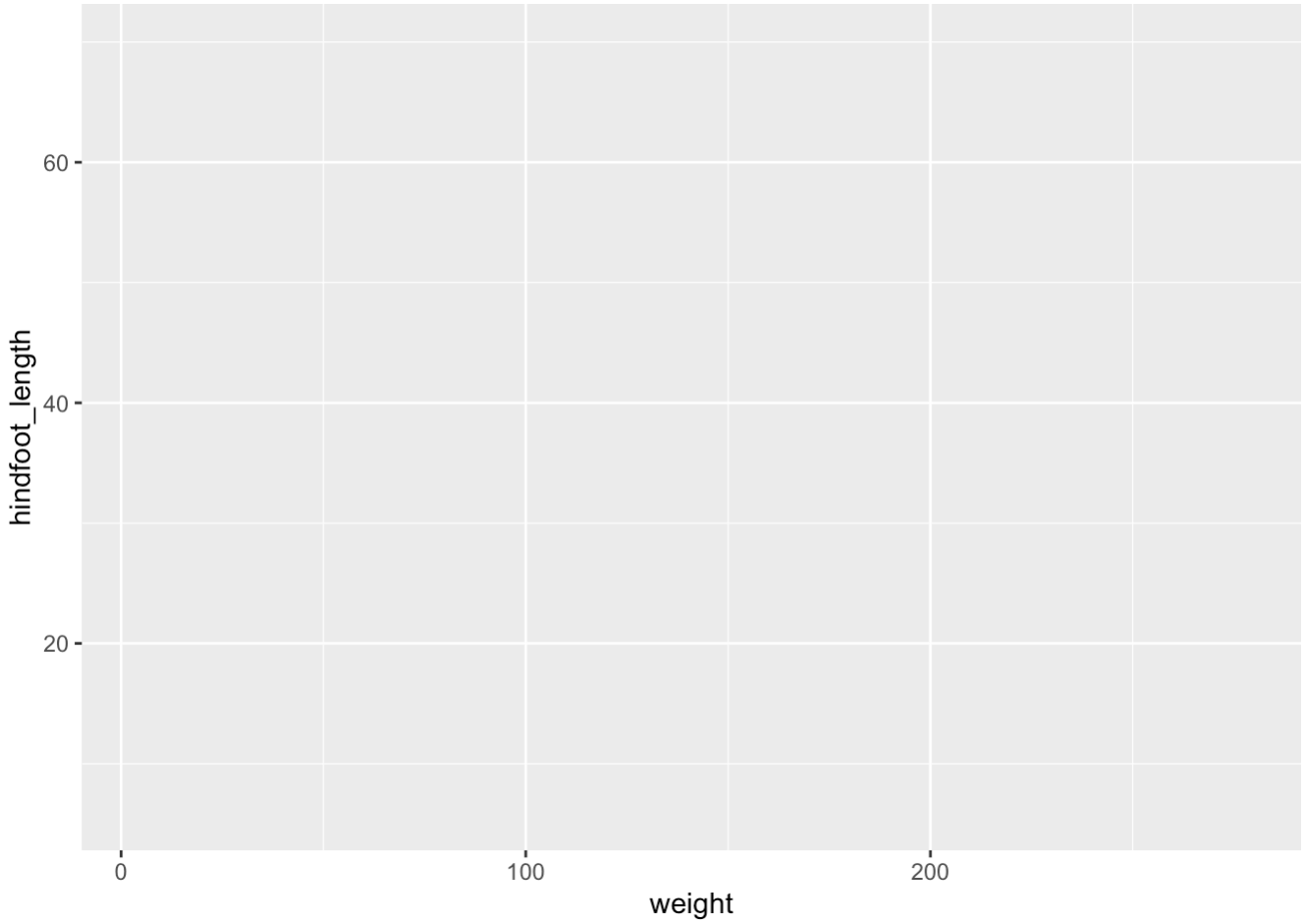
We use the `ggplot()` function to create a plot. In order to tell the function what data to use, we need to specify the `data` argument. This linking of the data to the function is called 'binding'. An 'argument' is an input that a function takes, and you set arguments using the = sign.

```
# ggplot with data

ggplot(data = complete_old)
```

2.3 Define an `aes` mapping

We get a blank plot because we haven't told `ggplot()` which variables from our data that we want to correspond to parts of the plot. We can specify the 'mapping' of variables to plot elements, such as the plot's x/y coordinates, size, or shape, by using the `aes()` function. We'll also add comments, which are any lines starting with a `#`.

```
# adding a mapping to x and y axes

ggplot(data = complete_old, mapping = aes(x = weight, y = hindfoot_length))
```



Tip - Commenting blocks of code

'Comment' or 'uncomment' in RStudio makes it easy to comment or uncomment a paragraph of commands so that they are not run. After selecting the lines you want to comment, press all the following keys at the same time on your keyboard `Ctrl + Shift + C` or OSX: `Cmd + Shift + C` . If you only want to comment out one line, you can put the cursor at any location of that line (i.e. no need to select the whole line), then press `Ctrl + Shift + C` or OSX: `Cmd + Shift + C` .

2.4 Add a 'geom'

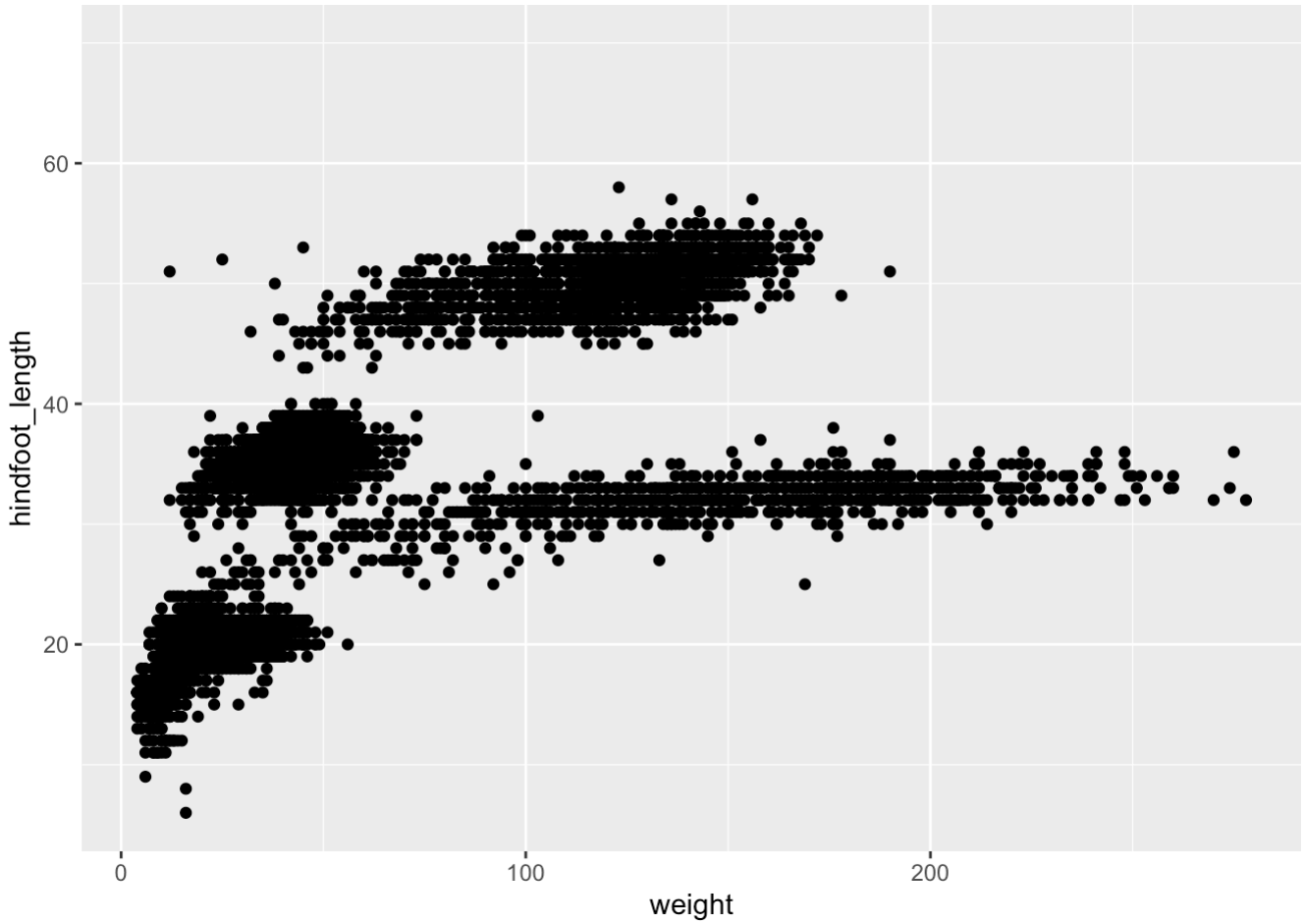
Now we've got a plot with x and y axes corresponding to variables from `complete_old` . However, we haven't specified how we want the data to be displayed. We do this using `geom_` functions, which specify the type of 'geometry' we want, such as points, lines, or bars. We can add a `geom_point()` layer to our plot by using the + sign. We indent onto a new line to make it easier to read, and we have to end the first line with the `+` sign.

- add 'geoms' – graphical representations of the data in the plot (eg points, lines, bars). **`ggplot2`** offers many different geoms; we will use some common ones today, including:
    - `geom_point()` for scatter plots, dot plots, etc.
    - `geom_boxplot()` for, well, boxplots!
    - `geom_line()` for trend lines, time series, etc.

Because we have two continuous variables, let's use `geom_point()` first:

```
# adding a geom_point

ggplot(data = complete_old, mapping = aes(x = weight, y = hindfoot_length)) +
  geom_point()
```

```
## Warning: Removed 3081 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



You may notice a warning that missing values were removed. If a variable necessary to make the plot is missing from a given row of data (in this case, `hindfoot_length` or `weight`), it can't be plotted. **`ggplot2`** just uses a warning message to let us know that some rows couldn't be plotted.

Tip - Warnings

Warning messages are one of a few ways R will communicate with you. Warnings can be thought of as a "heads up". Nothing necessarily went wrong, but the author of that function wanted to draw your attention to something. In the above case, it's worth knowing that some of the rows of your data were not plotted because they had missing data.

A more serious type of message is an error. Here's an example:

```
# example of error

# ggplot(data = complete_old, mapping = aes(x = weight, y = hindfoot_length)) +
#   geom_poit()
```

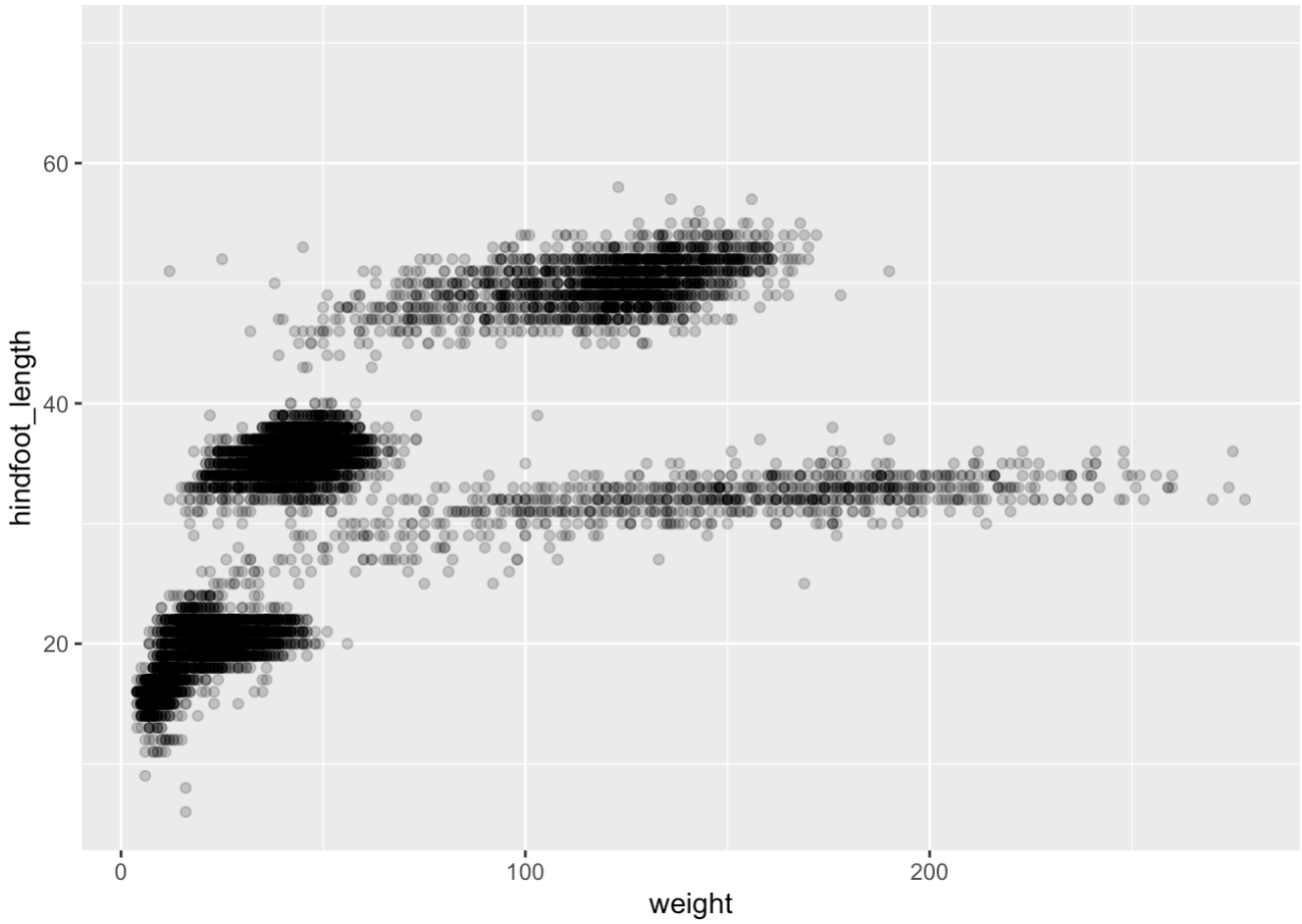
As you can see, we only get the error message, with no plot, because something has actually gone wrong. This particular error message is fairly common, and it happened because we misspelled 'point' as 'poit'. Because there is no function named geom_poit(), R tells us it can't find a function with that name.

2.5 Changing aesthetics

Building `ggplot` plots is often an iterative process, so we'll continue developing the scatter plot we just made. You may have noticed that parts of our scatter plot have many overlapping points, making it difficult to see all the data. We can adjust the transparency of the points using the `alpha` argument, which takes a value between 0 and 1:

```
# adding transparency

ggplot(data = complete_old, mapping = aes(x = weight, y = hindfoot_length)) +
  geom_point(alpha = 0.2)
```

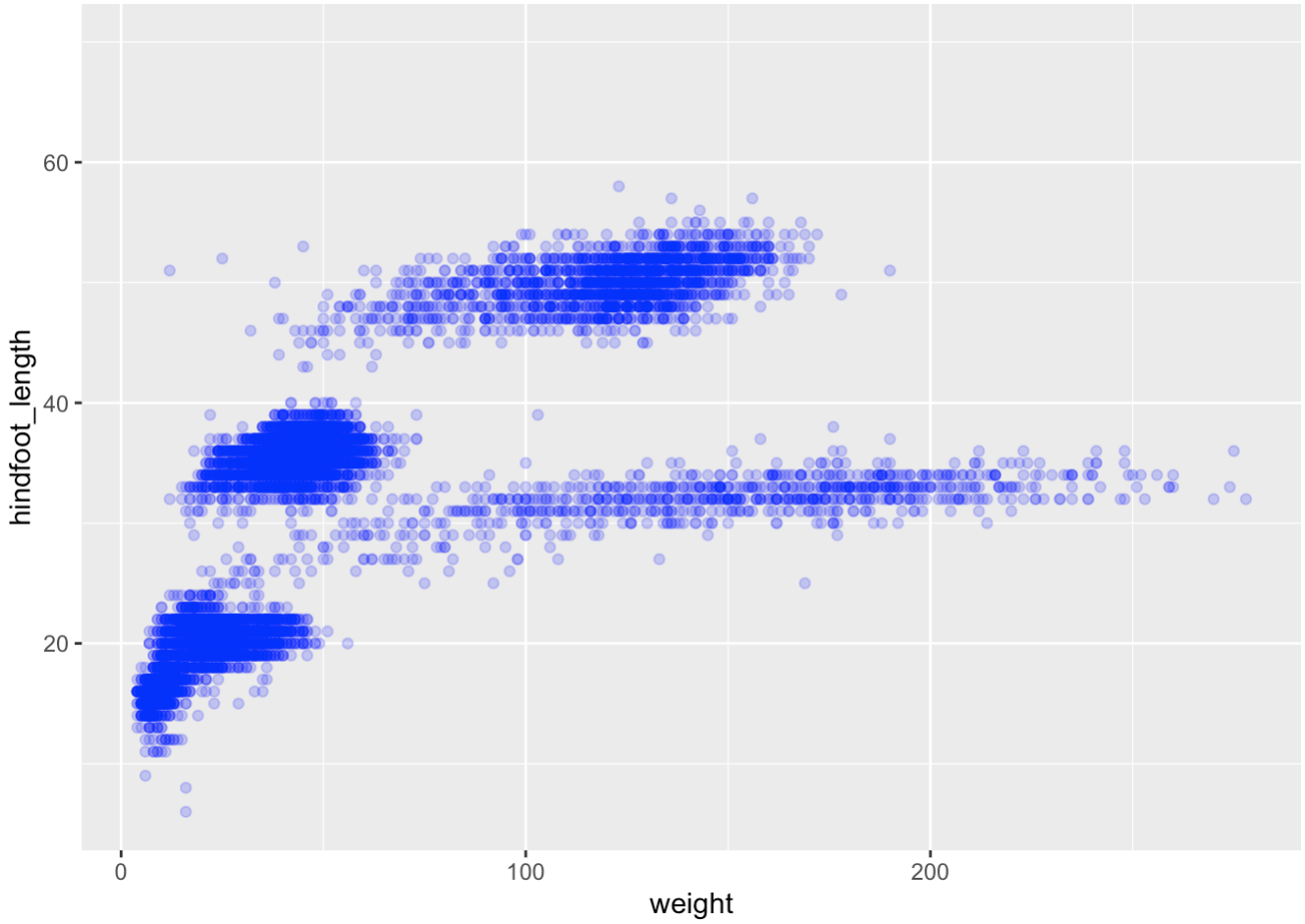
```
## Warning: Removed 3081 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



We can also change the colour of the points:

```
# change colour

ggplot(data = complete_old, mapping = aes(x = weight, y = hindfoot_length)) +
  geom_point(alpha = 0.2, colour = "blue")
```

```
## Warning: Removed 3081 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



Tip - Errors

Two common issues you might run into when working in R are forgetting a closing bracket or a closing quote. Let's take a look at what each one does.

Try running the following code:

```
# demonstrate an error

# ggplot(data = complete_old, mapping = aes(x = weight, y = hindfoot_length)) +
#   geom_point(colour = "blue", alpha = 0.2
```

You will see a `+` appear in your console. This is R telling you that it expects more input in order to finish running the code. It is missing a closing bracket to end the geom_point function call. You can hit `Esc` in the console to reset it.

You will also see a red cross in the code cell. If you hover over it, a helpful message will appear.

Something similar will happen if you run the following code:

```
# demonstrate an error

# ggplot(data = complete_old, mapping = aes(x = weight, y = hindfoot_length)) +
#   geom_point(colour = "blue, alpha = 0.2)
```

A missing quote at the end of blue means that the rest of the code is treated as part of the quote, which is a bit easier to see since RStudio displays character strings in a different colour.

You will get a different error message if you run the following code:

```
# demonstrate an error

# ggplot(data = complete_old, mapping = aes(x = weight, y = hindfoot_length)) +
#   geom_point(colour = "blue", alpha = 0.2))
```

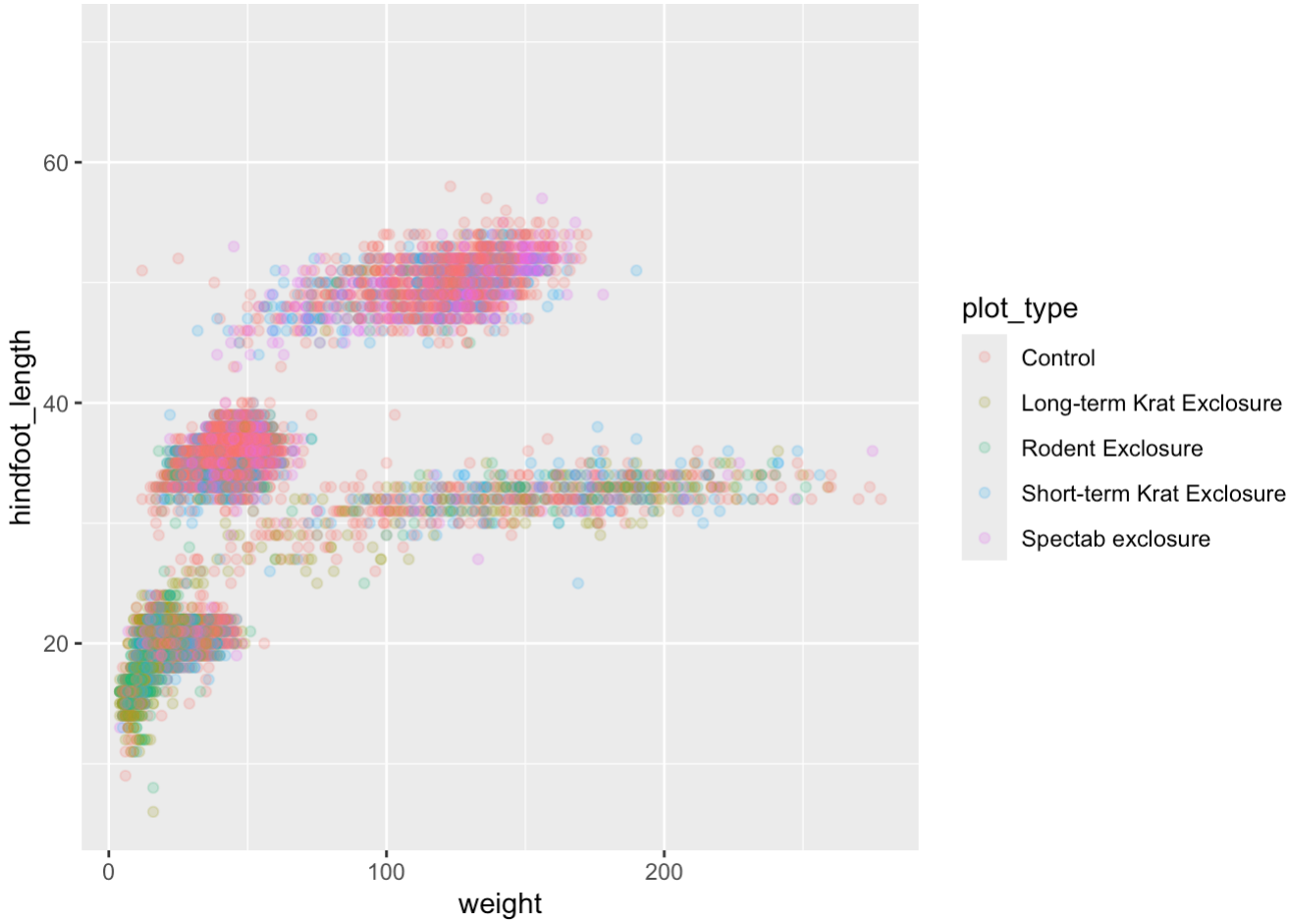
This time we have an extra closing), which R doesn't know what to do with. It tells you there is an unexpected), but it doesn't pinpoint exactly where (though the hover message on the red cross in the code cell does a better job!). With enough time working in R, you will get better at spotting mismatched brackets.

2.6 Adding another variable

Let's try colouring our points according to the type of sampling plot (plot here refers to the physical area where rodents were sampled and has nothing to do with making graphs). Since we're now mapping a variable (`plot_type`) to a component of the ggplot2 plot (`colour`), we need to put the argument inside `aes()` :

```
# add plot_type

ggplot(data = complete_old, mapping = aes(x = weight, y = hindfoot_length, colour = p
lot_type)) +
  geom_point(alpha = 0.2)
```

```
## Warning: Removed 3081 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



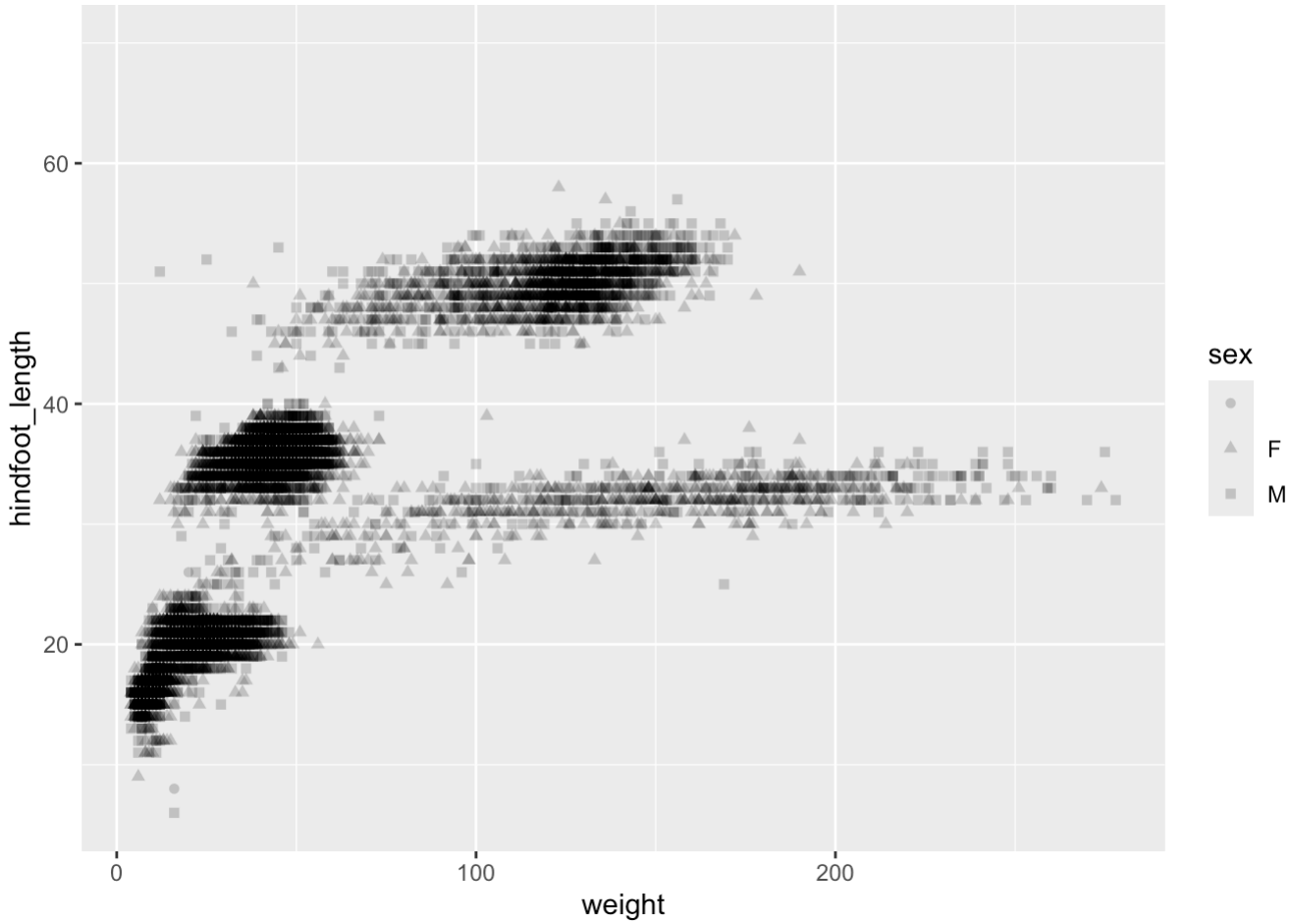
Challenge 1.a: Modifying plots

Try modifying the plot so that the shape of the point varies by sex. You will set the shape the same way you set the colour.

Do you think this is a good way to represent sex with these data?

```
ggplot(data = complete_old,
       mapping = aes(x = weight, y = hindfoot_length, shape = sex)) +
  geom_point(alpha = 0.2)
```

```
## Warning: Removed 3081 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```

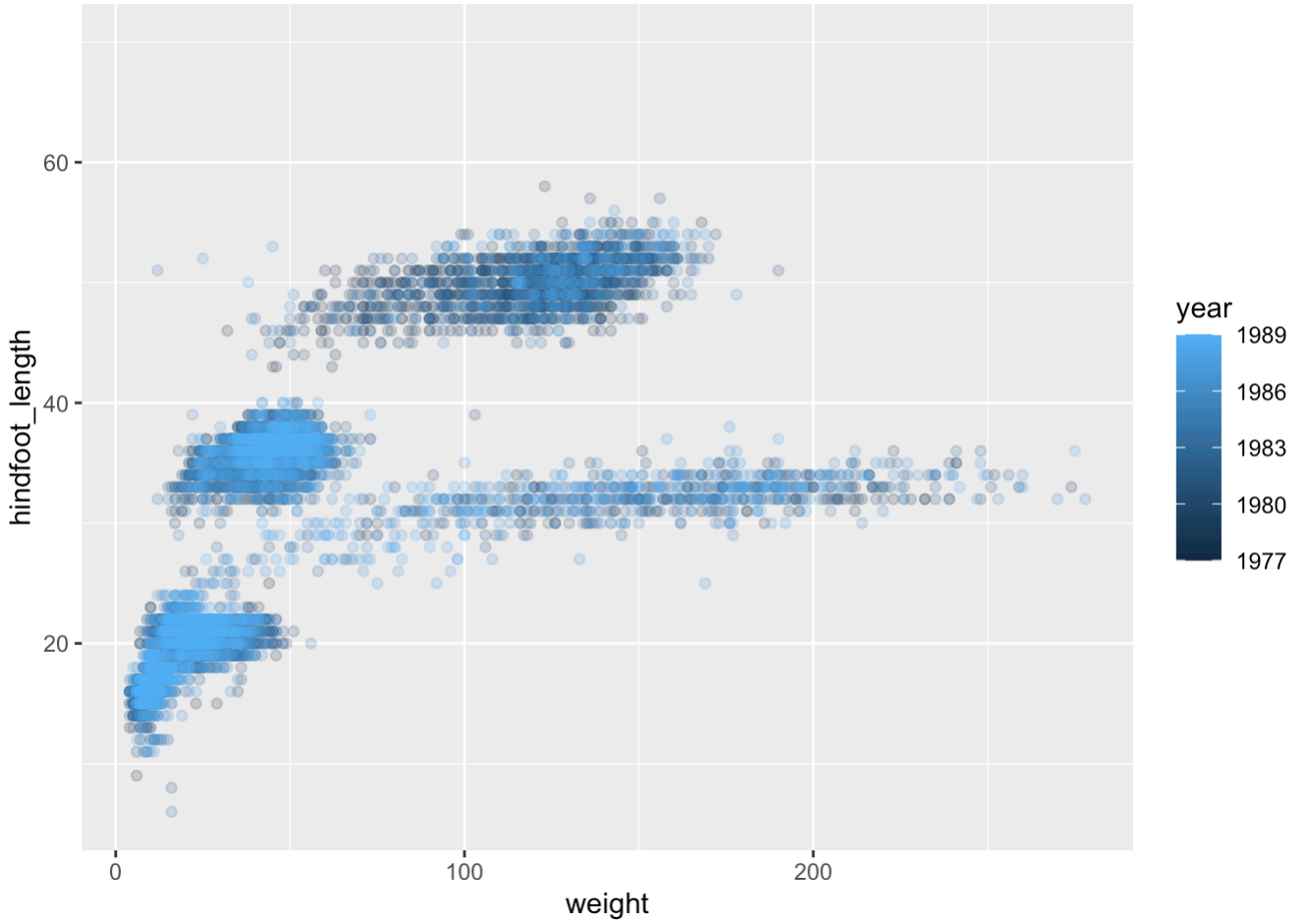


Challenge 1.b: Modifying plots

Now try changing the plot so that the colour of the points vary by year. Do you notice a difference in the colour scale on the legend compared to displaying colour by plot type? Why do you think this happened?

```
ggplot(data = complete_old,
       mapping = aes(x = weight, y = hindfoot_length, colour = year)) +
  geom_point(alpha = 0.2)
```

```
## Warning: Removed 3081 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



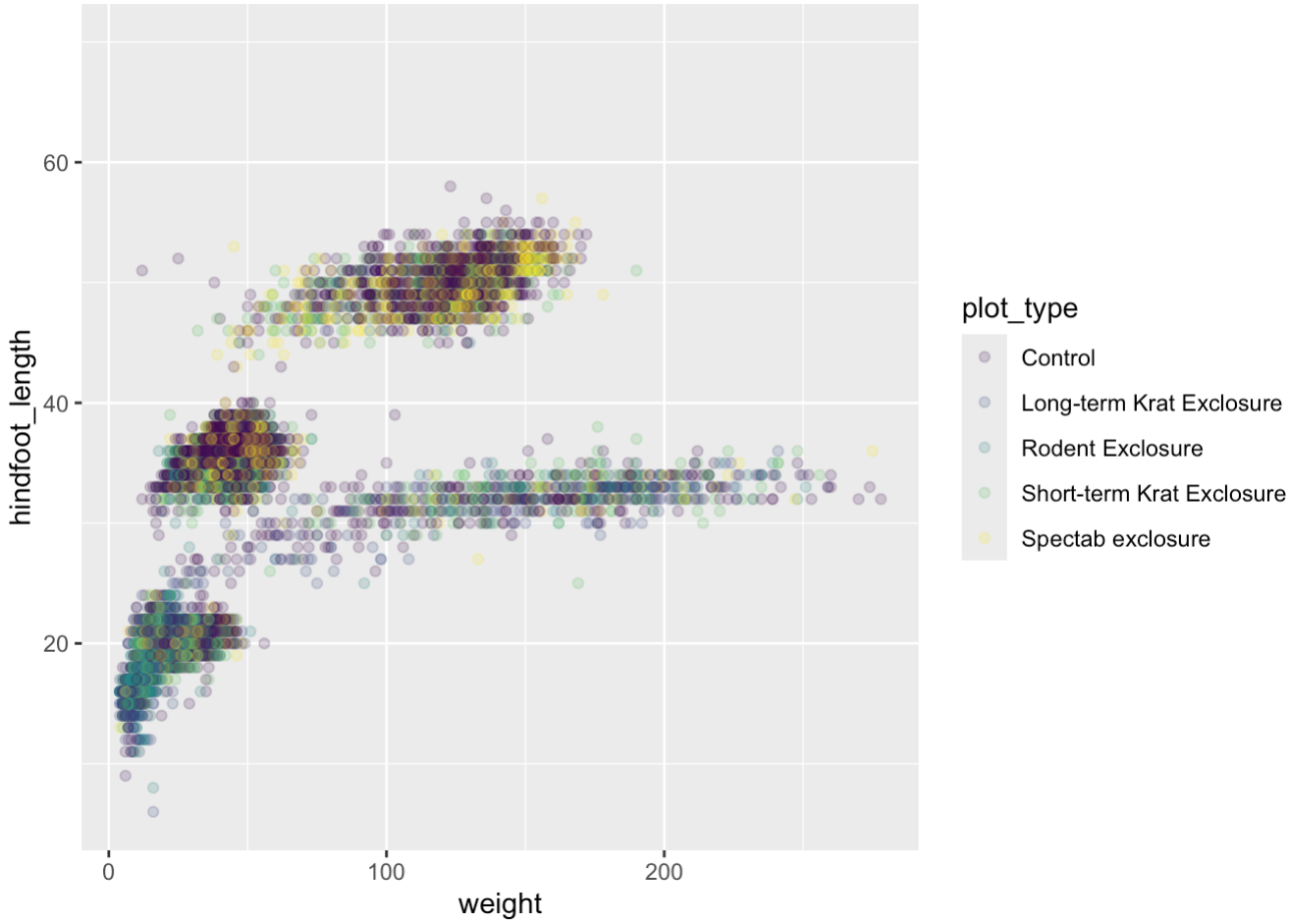
For Challenge 1.b, the colour scale is different compared to using colour = plot_type because plot_type and year are different variable *types*. plot_type is a categorical variable, so **`ggplot2`** defaults to use a **discrete** colour scale, whereas year is a numeric variable, so ggplot2 uses a **continuous** colour scale.

2.7 Changing colour scales

The default discrete colour scale isn't always ideal: it isn't friendly to viewers with colour-blindness and it doesn't translate well to grayscale. However, **`ggplot2`** comes with quite a few other colour scales, including the fantastic `viridis` colour scales, which are designed to be colourblind and grayscale friendly. We can change colour scales by adding `scale_` functions to our plots:

```
# change scale

ggplot(data = complete_old, mapping = aes(x = weight, y = hindfoot_length, colour = p
lot_type)) +
  geom_point(alpha = 0.2) +
  scale_colour_viridis_d()
```

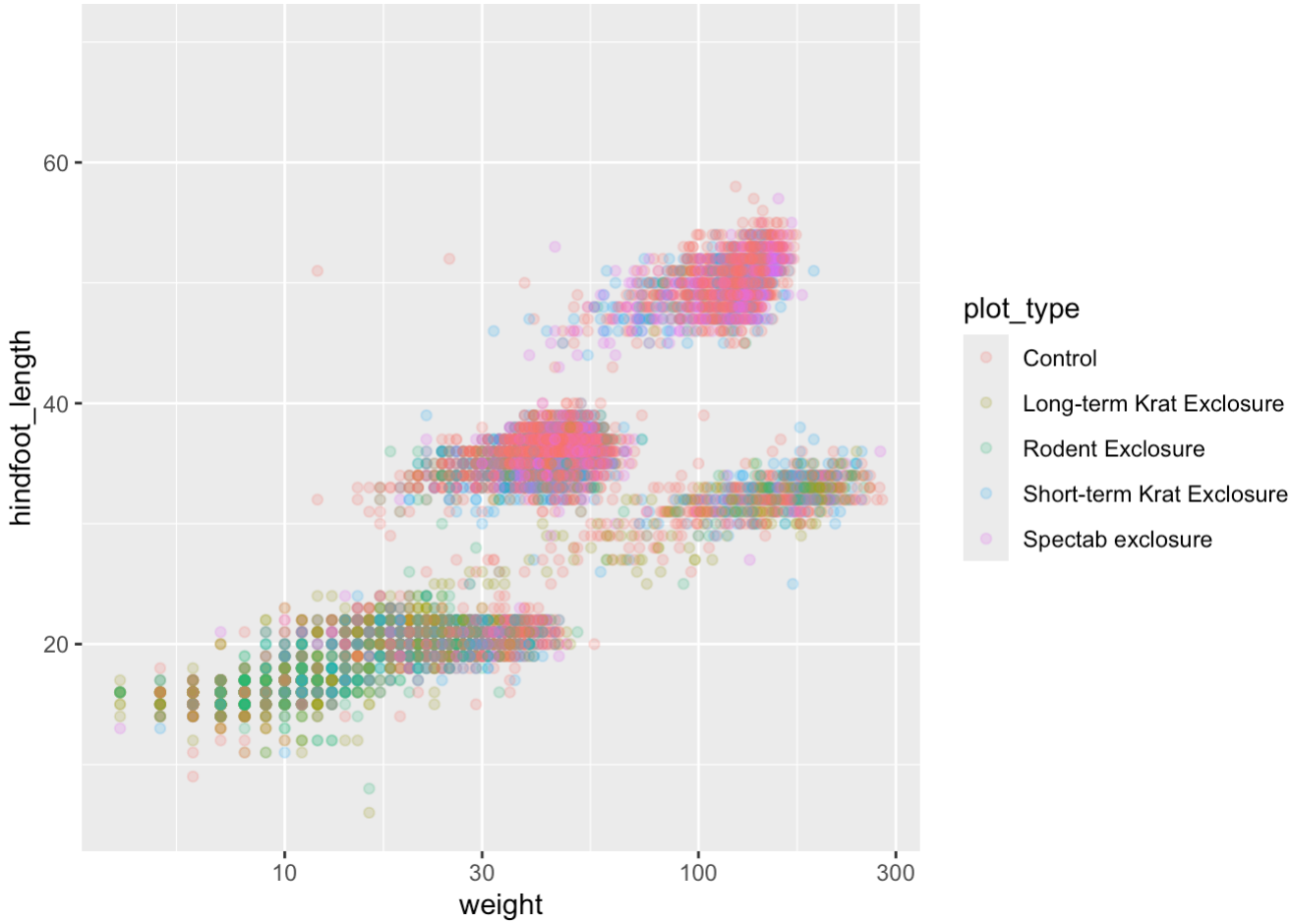
```
## Warning: Removed 3081 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



Scales don't just apply to colours- any plot component that you put inside `aes()` can be modified with `scale_` functions. Just as we modified the scale used to map `plot_type` to `colour`, we can modify the way that the data variable `weight` is mapped to the x axis by using the `scale_x_log10()` function:

```
# use log10 scale

ggplot(data = complete_old, mapping = aes(x = weight, y = hindfoot_length, colour = p
lot_type)) +
  geom_point(alpha = 0.2) +
  scale_x_log10()
```

```
## Warning: Removed 3081 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```

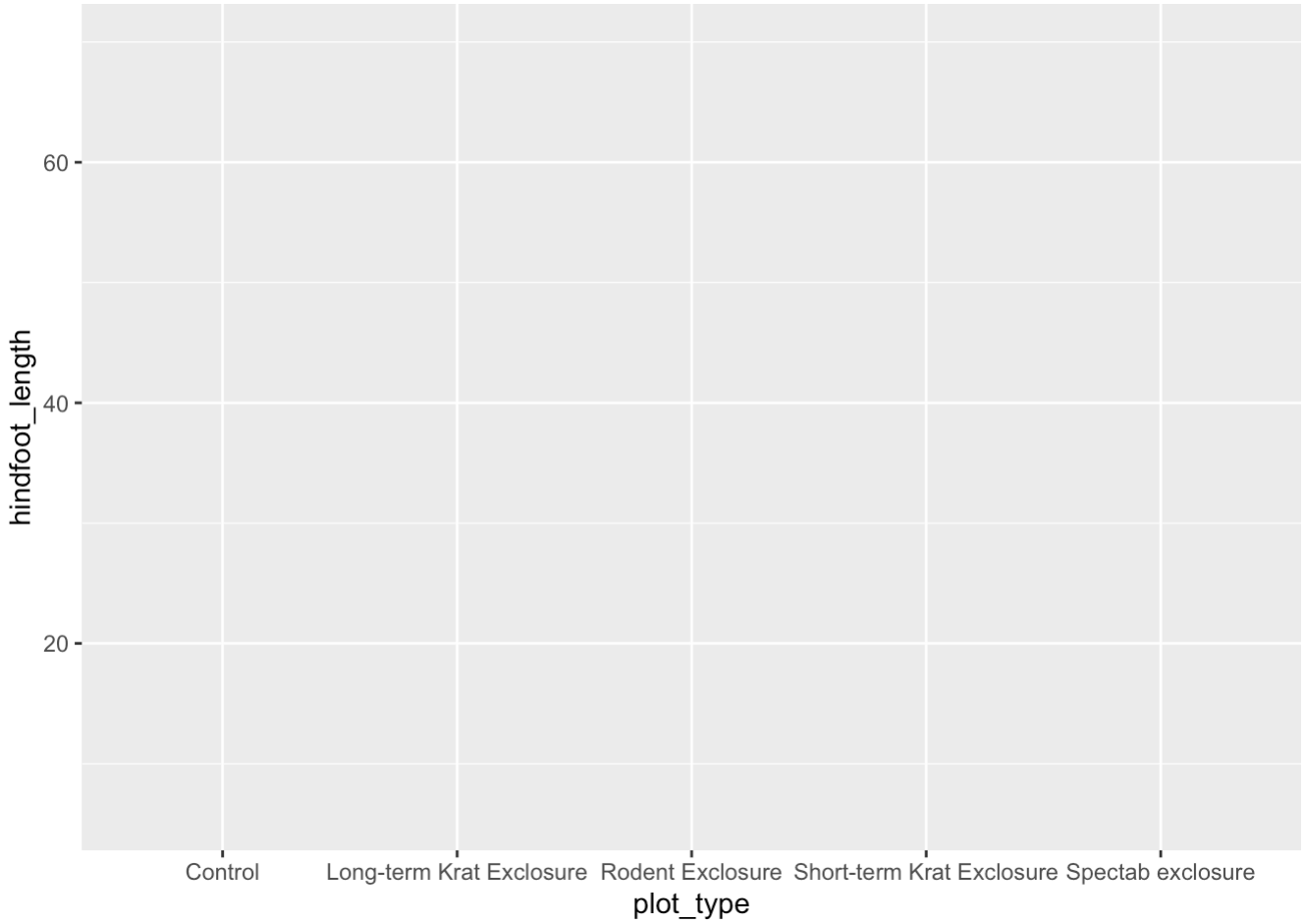


One nice thing about ggplot and the tidyverse in general is that groups of functions that do similar things are given similar names. Any function that modifies a ggplot scale starts with `scale_`, making it easier to search for the correct function.

2.8 Boxplot

Let's try making a different type of plot altogether. We'll start off with our same basic building blocks using `ggplot()` and `aes()`.

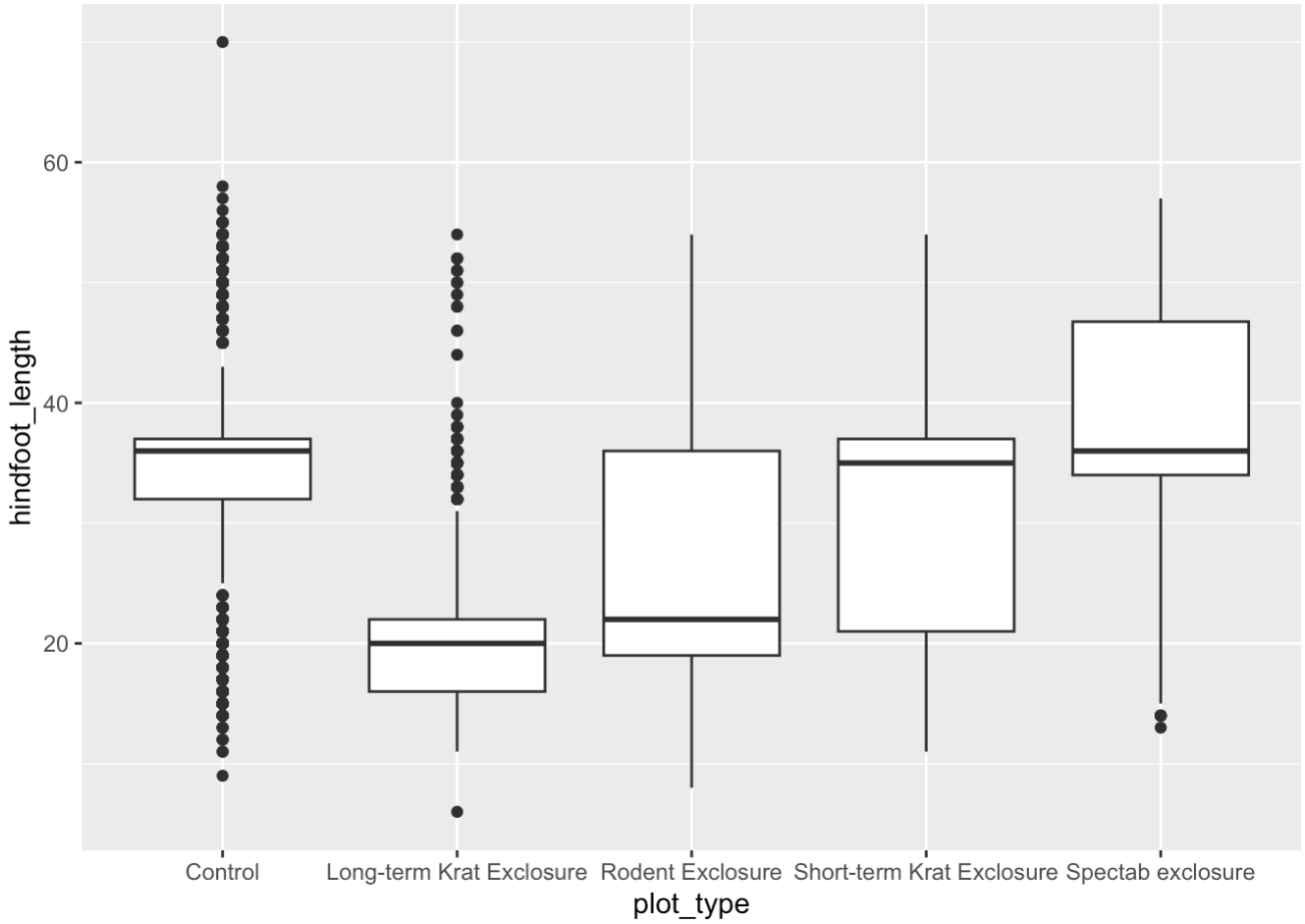
```
# create ggplot

ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length))
```



This time, let's try making a boxplot, which will have `plot_type` on the x axis and hindfoot_length on the y axis. We can do this by adding `geom_boxplot()` to our `ggplot()`:

```
# add geom

ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length)) +
  geom_boxplot()
```

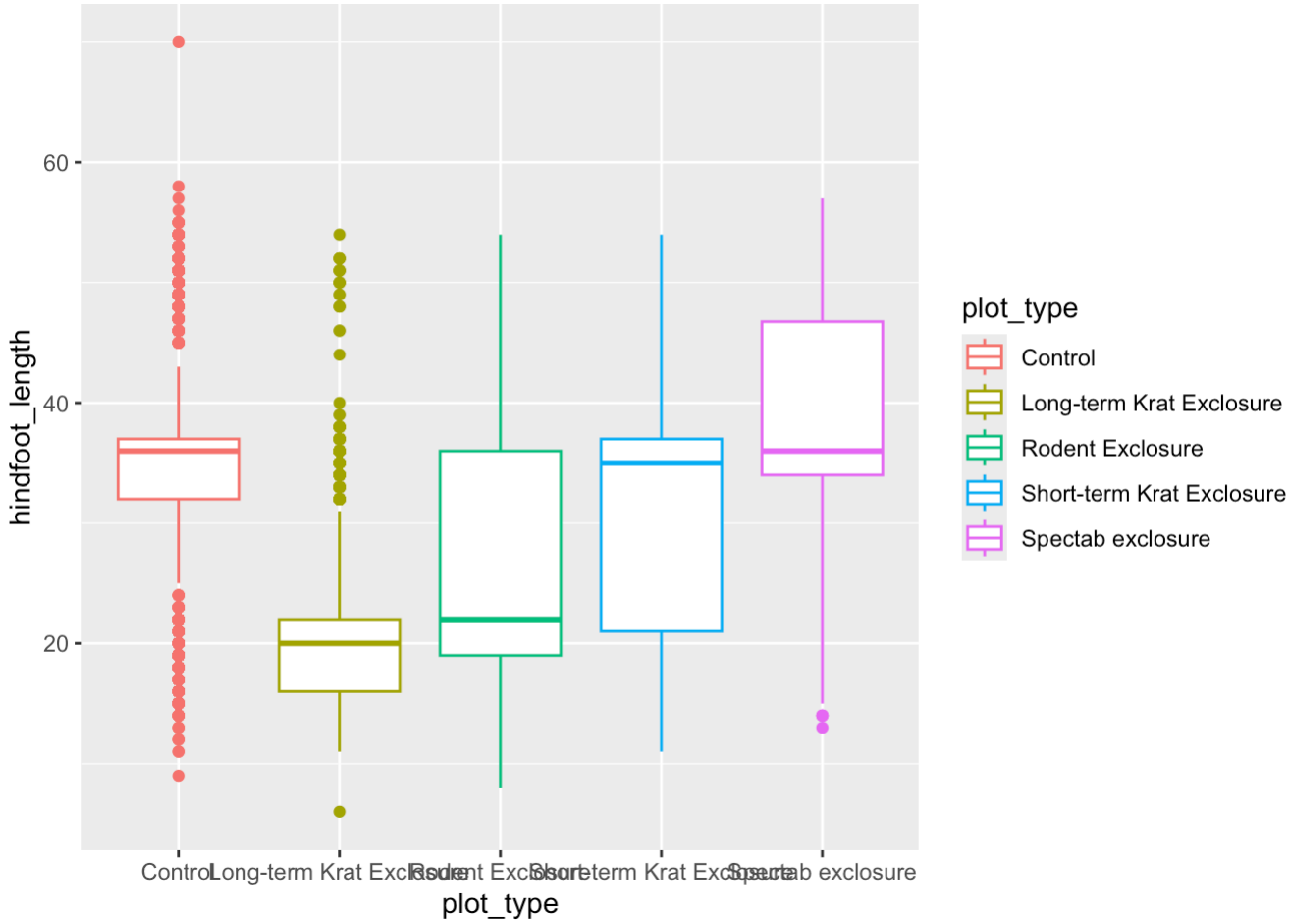
```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```



Just as we coloured the points before, we can colour our boxplot by `plot_type` as well:

```
# add colour

ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length, colour
= plot_type)) +
  geom_boxplot()
```

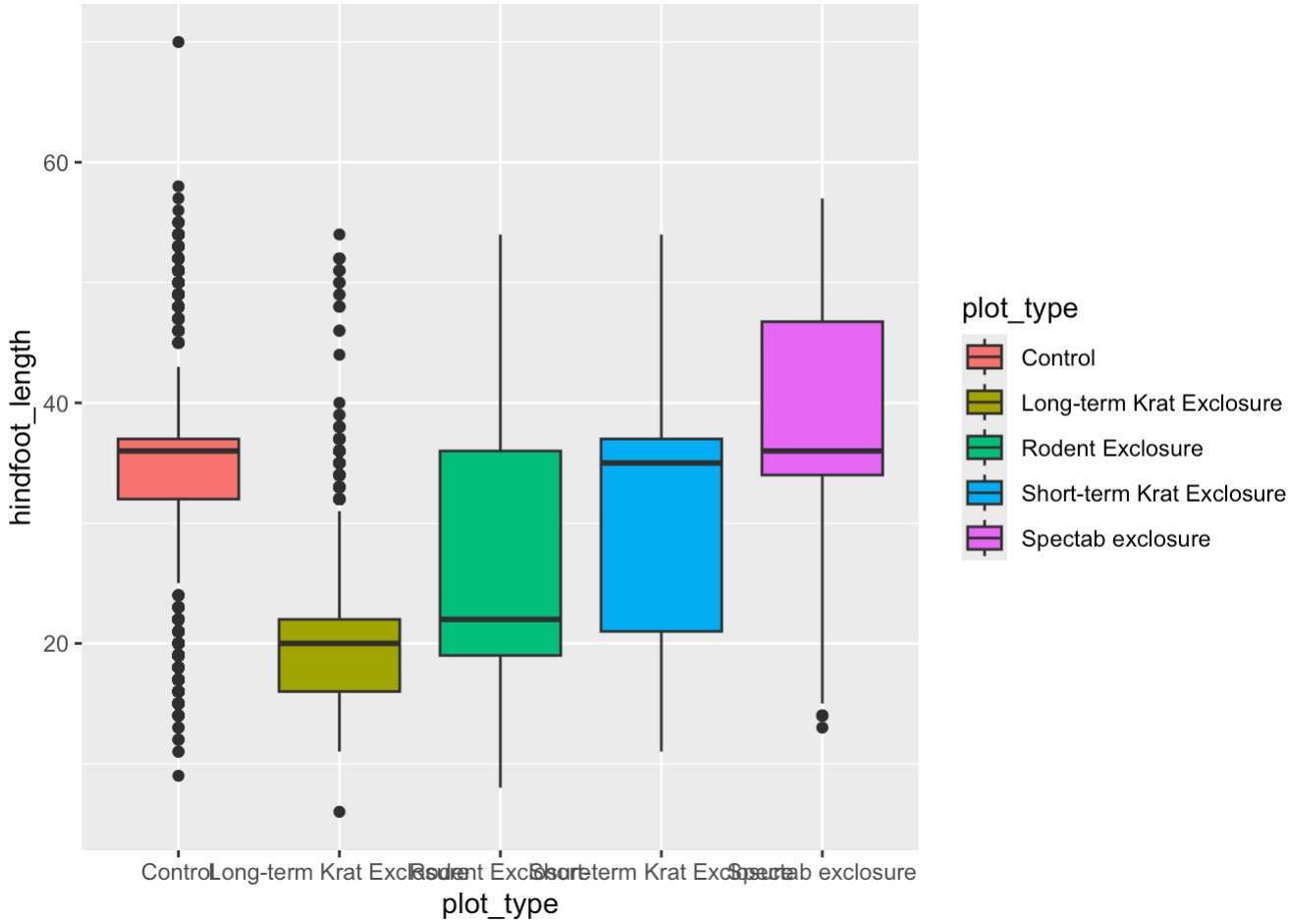
```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```



It looks like `colour` has only affected the outlines of the boxplot, not the rectangular/interior portions. This is because the `colour` only impacts 1-dimensional parts of a `ggplot` : points and lines. To change the colour of 2-dimensional parts of a plot, we use `fill` :

```
# add fill

ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length, fill =
plot_type)) +
  geom_boxplot()
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```



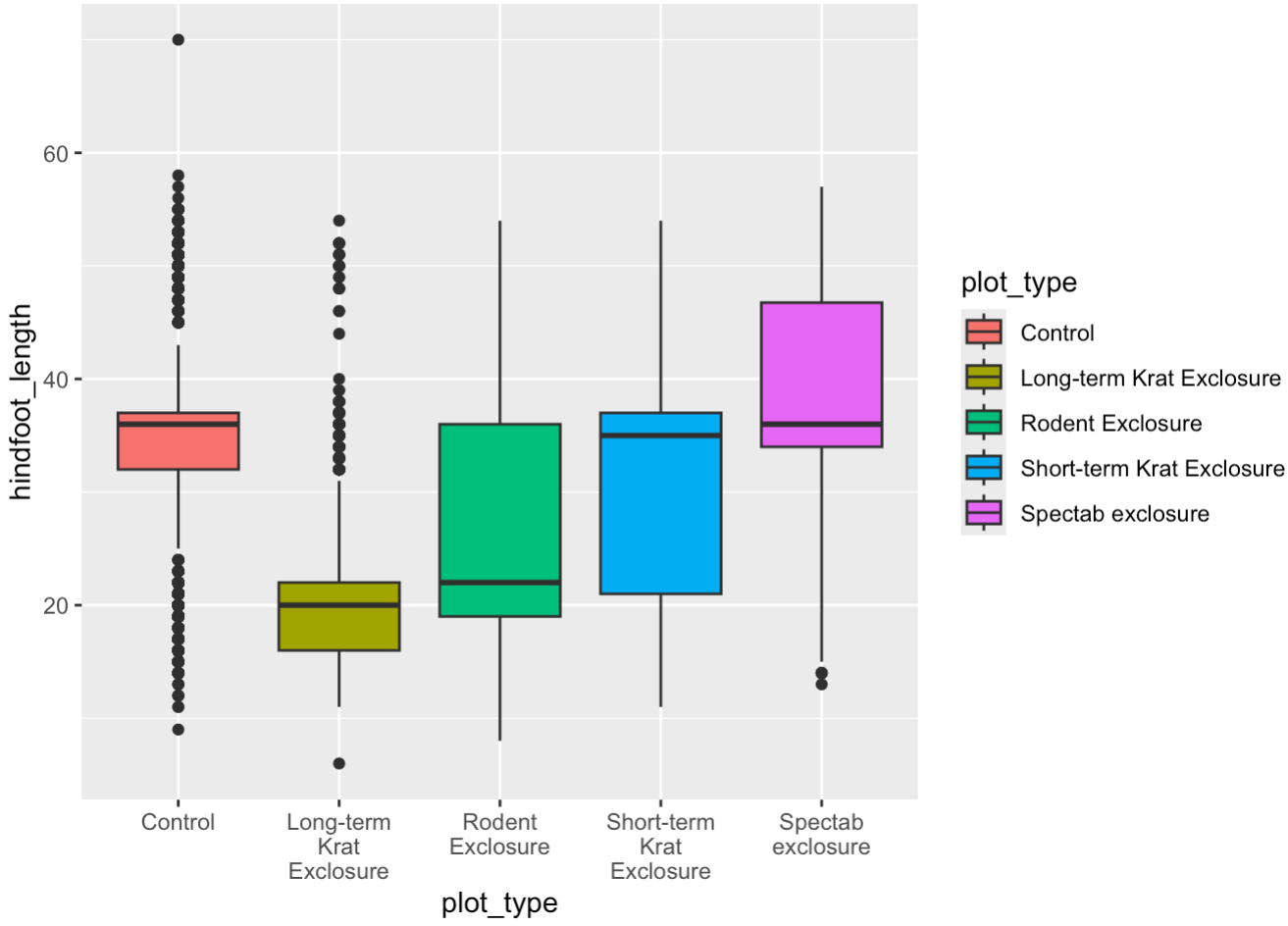
Tip - Plot labels

One thing you may notice on this plot is that the axis labels are overlapping each other, depending on how wide your plot viewer is. One way to help make the axis labels more legible is to wrap the text. We can do that by modifying the `labels` for the x axis `scale`.

We do this using the `scale_x_discrete()` function because we have a discrete variable on the axis, and we modify the `labels` argument. The function `label_wrap_gen()` will wrap the text of the labels to make them more legible.

```
# add labels

ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length, fill =
plot_type)) +
  geom_boxplot() +
  scale_x_discrete(labels = label_wrap_gen(width = 10))
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```



2.9 Adding geoms

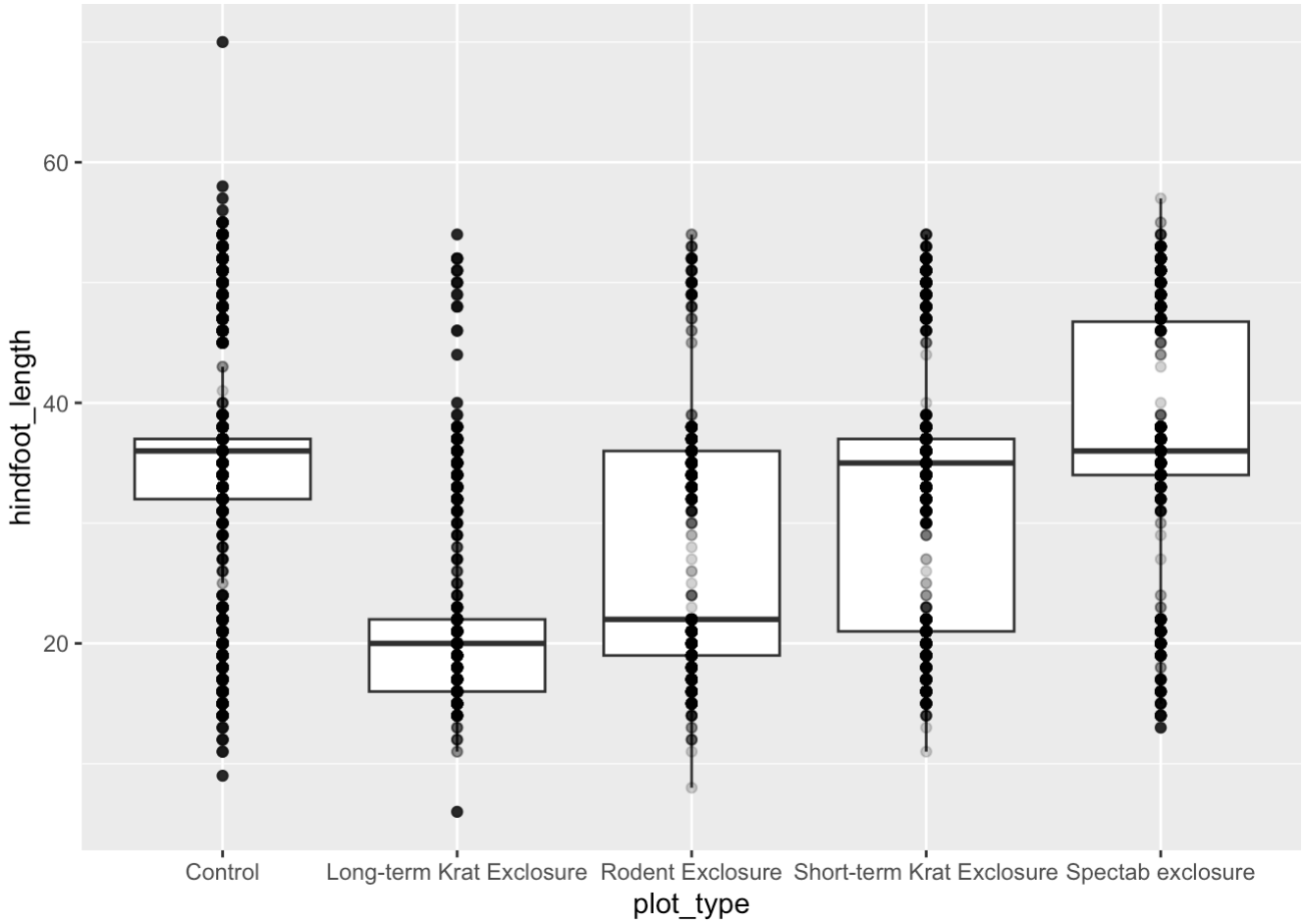
One of the most powerful aspects of `ggplot` is the way we can add components to a plot in successive layers. While boxplots can be very useful for summarizing data, it is often helpful to show the raw data as well. With `ggplot`, we can easily add another `geom_` to our plot to show the raw data too.

Let's add `geom_point()` to visualise the raw data on our plot. We will modify the 'alpha' argument to help with over-plotting.

```
# add geom_point

ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length)) +
  geom_boxplot() +
  geom_point(alpha = 0.2)
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```

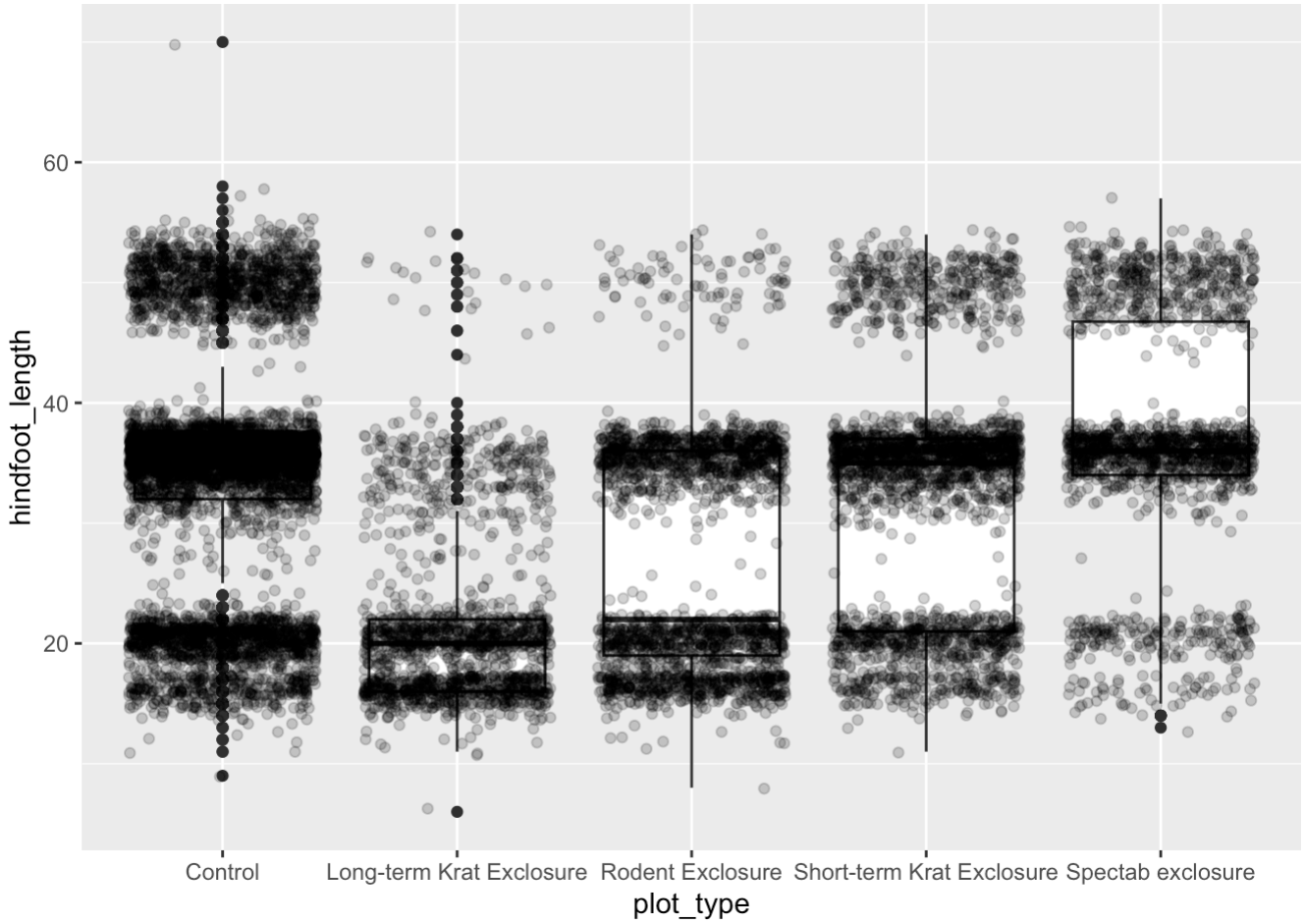


Uh oh… all our points for a given x axis category fall exactly on a line, which isn't very useful. We can shift to using `geom_jitter()`, which will add points to the plot with a bit of random noise added to the positions to prevent this from happening.

```
# ad geom_jitter

ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length)) +
  geom_boxplot() +
  geom_jitter(alpha = 0.2)
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```

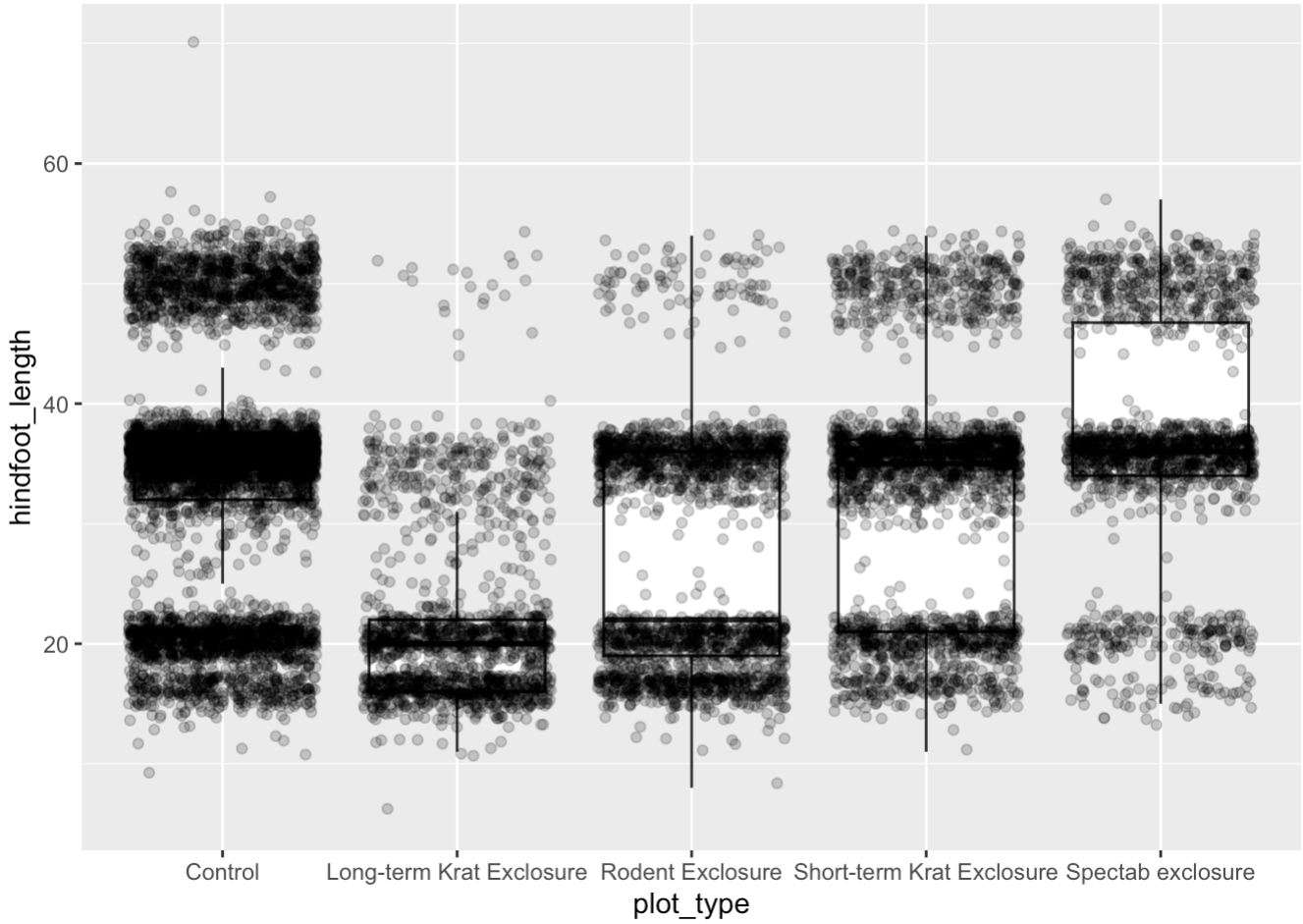


You may have noticed that some of our data points are now appearing on our plot twice: the outliers are plotted from `geom_boxplot()` as black points, but they are also plotted with `geom_jitter()`. Since we don't want to represent these data multiple times in the same form (points), we can stop `geom_boxplot()` from plotting them. We do this by setting the `outlier.shape` argument to `NA`, which means the outliers don't have a shape to be plotted.

```
# remove outliers

ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length)) +
  geom_boxplot(outlier.shape = NA) +
  geom_jitter(alpha = 0.2)
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```

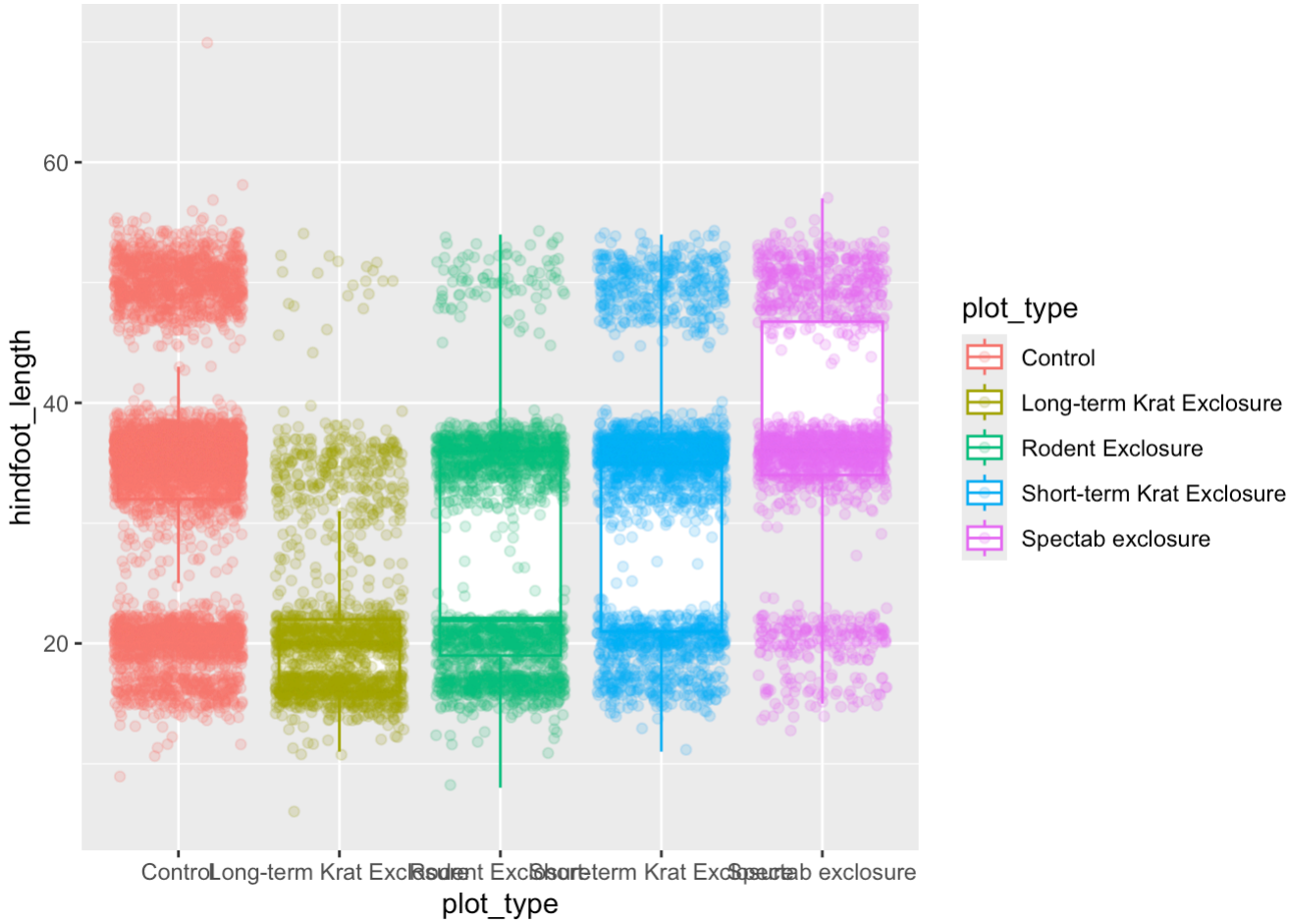


Just as before, we can map `plot_type` to colour by putting it inside `aes()`.

```
# add plot_type

ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length, colour
= plot_type)) +
  geom_boxplot(outlier.shape = NA) +
  geom_jitter(alpha = 0.2)
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



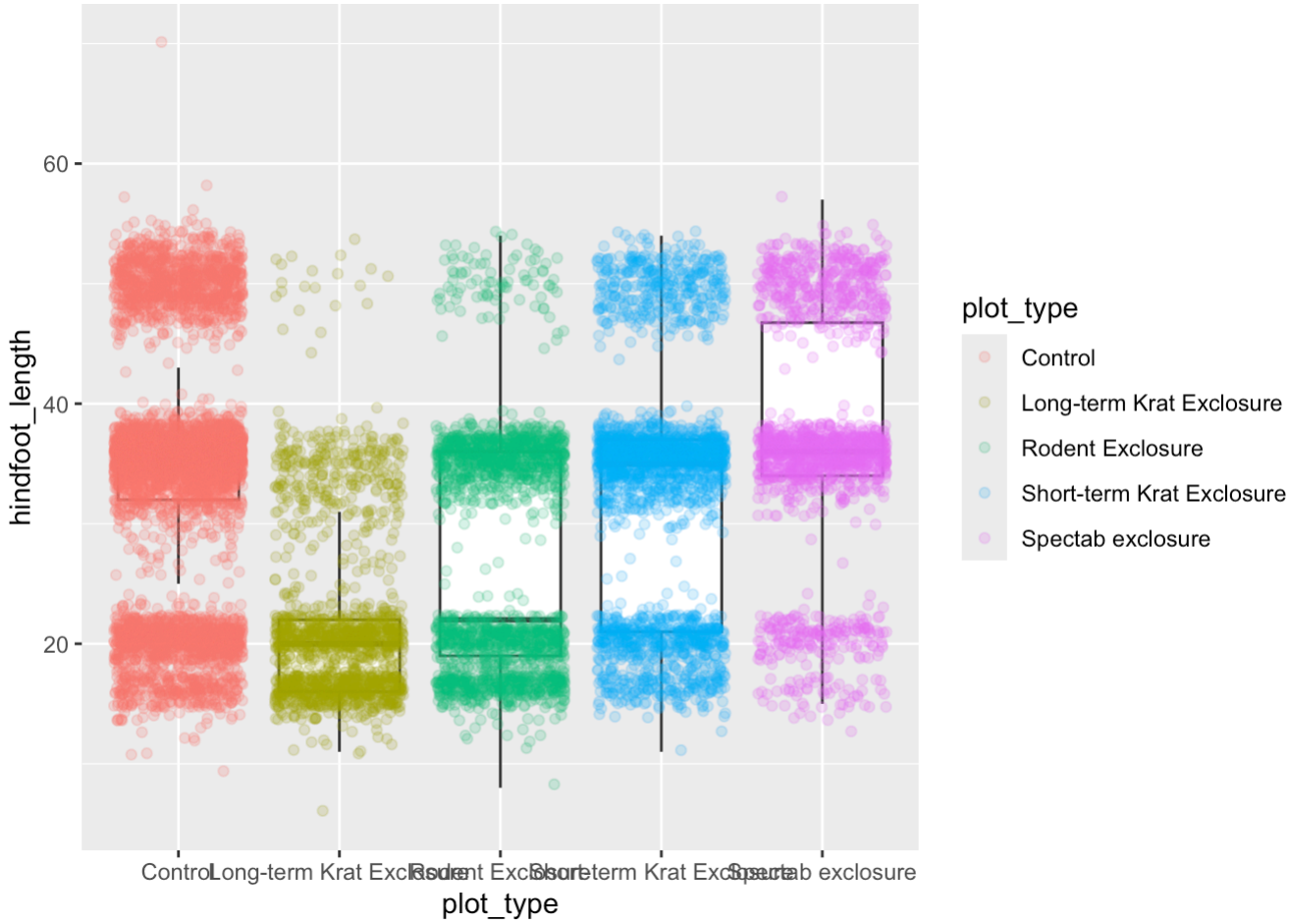
Notice that both the colour of the points and the colour of the boxplot lines changed. Any time we specify an `aes()` mapping inside our initial `ggplot()` function, that mapping will apply to all our geoms.

If we want to limit the `aes()` mapping to a single `geom`, we can put the mapping into the specific `geom_` function, like this:

```
# limit colour to geom_jitter

ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length)) +
  geom_boxplot(outlier.shape = NA) +
  geom_jitter(aes(colour = plot_type), alpha = 0.2)
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```

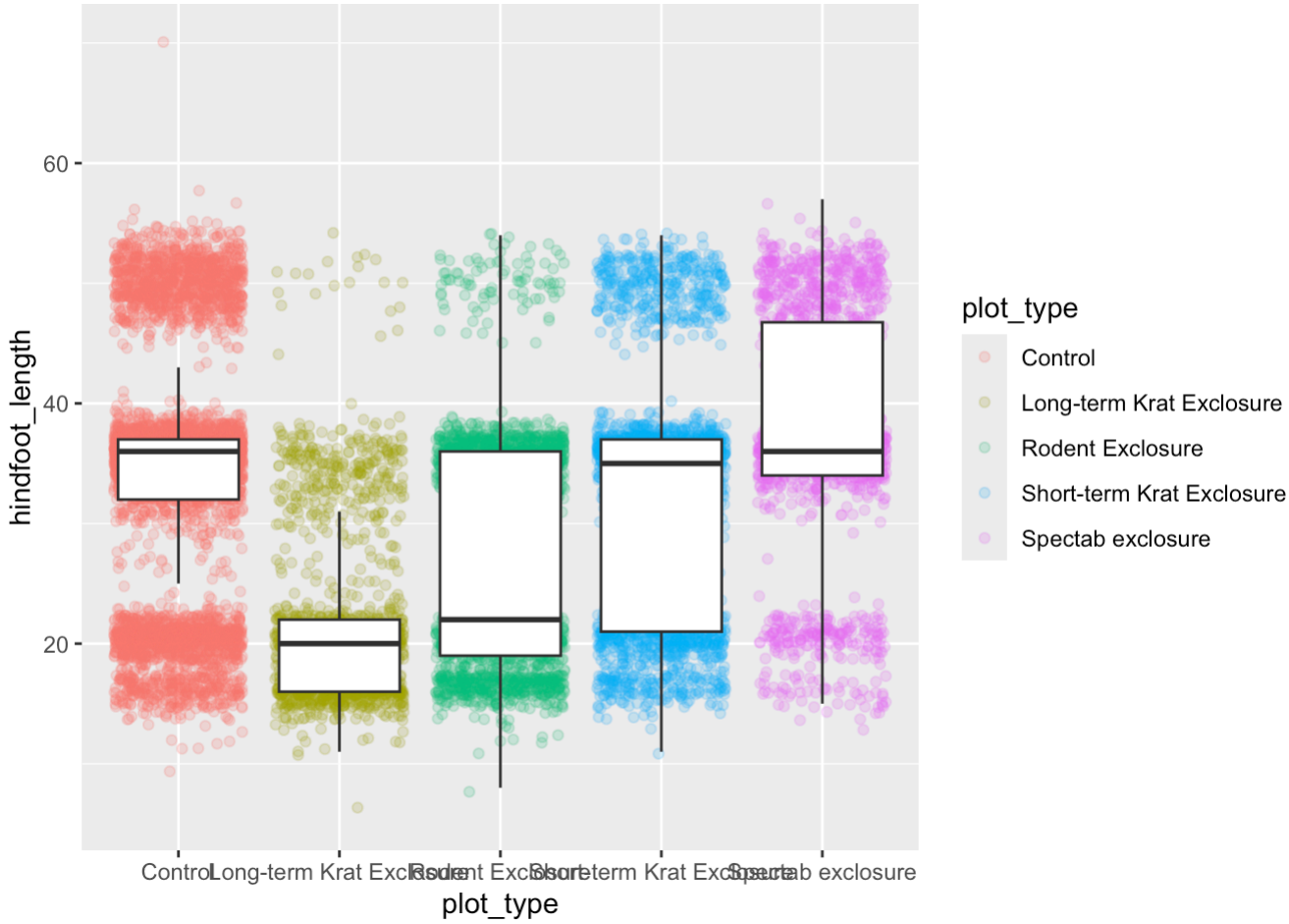


Now our points are coloured according to `plot_type`, but the boxplots are all the same colour. One thing you might notice is that even with `alpha = 0.2`, the points obscure parts of the boxplot. This is because the `geom_point()` layer comes after the `geom_boxplot()` layer, which means the points are plotted on top of the boxes. To put the boxplots on top, we switch the order of the layers:

```
# switch order of layers

ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length)) +
  geom_jitter(aes(colour = plot_type), alpha = 0.2) +
  geom_boxplot(outlier.shape = NA)
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```

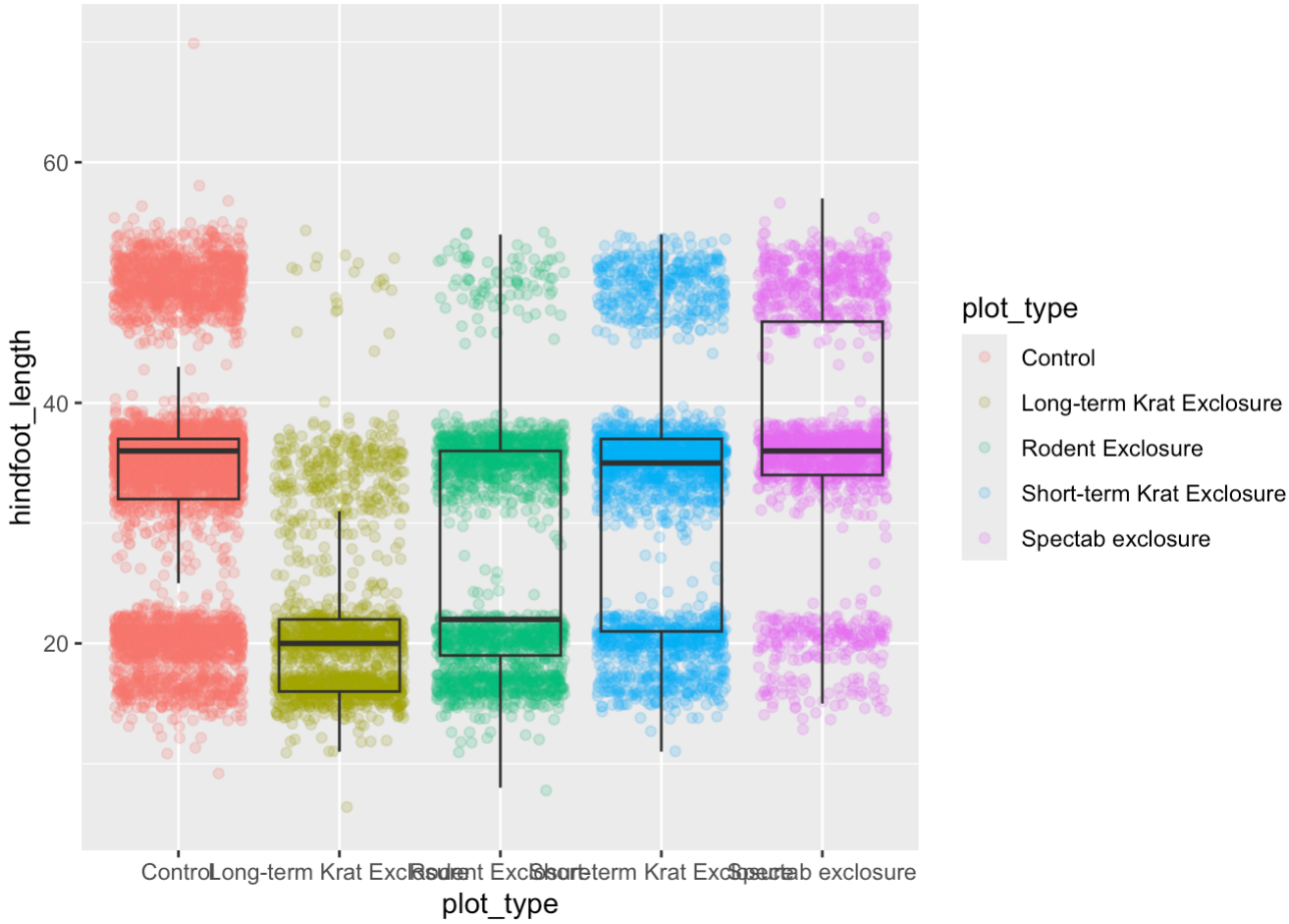


Now we have the opposite problem! The white `fill` of the boxplots completely obscures some of the points. To address this problem, we can remove the `fill` from the boxplots altogether, leaving only the black lines. To do this, we set `fill` to `NA`:

```
# remove fill

ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_length)) +
  geom_jitter(aes(colour = plot_type), alpha = 0.2) +
  geom_boxplot(outlier.shape = NA, fill = NA)
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



Now we can see all the raw data and our boxplots on top!

Challenge 2a: Change geoms

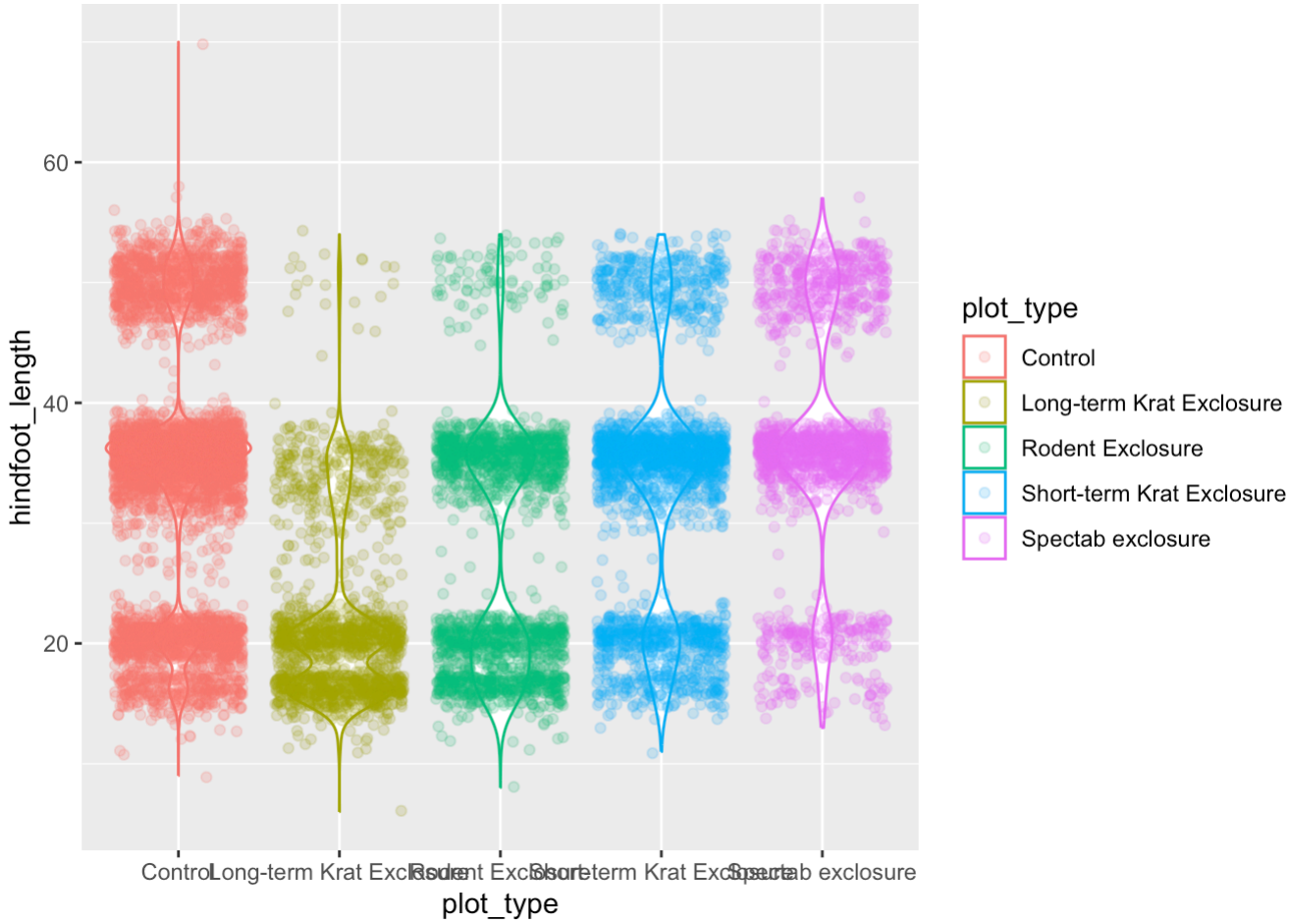
Violin plots are similar to boxplots- try making one using `plot_type` and `hindfoot_length` as the x and y variables. Remember that all geom functions start with `geom_` , followed by the type of geom.

This might also be a place to test your search engine skills. It is often useful to search for **R package_name stuff you want to search**. So for this example we might search for **R ggplot2 violin plot**.

```
ggplot(data = complete_old,
       mapping = aes(x = plot_type,
                     y = hindfoot_length,
                     color = plot_type)) +
  geom_violin(fill = "white") +
  geom_jitter(alpha = 0.2)
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_ydensity()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



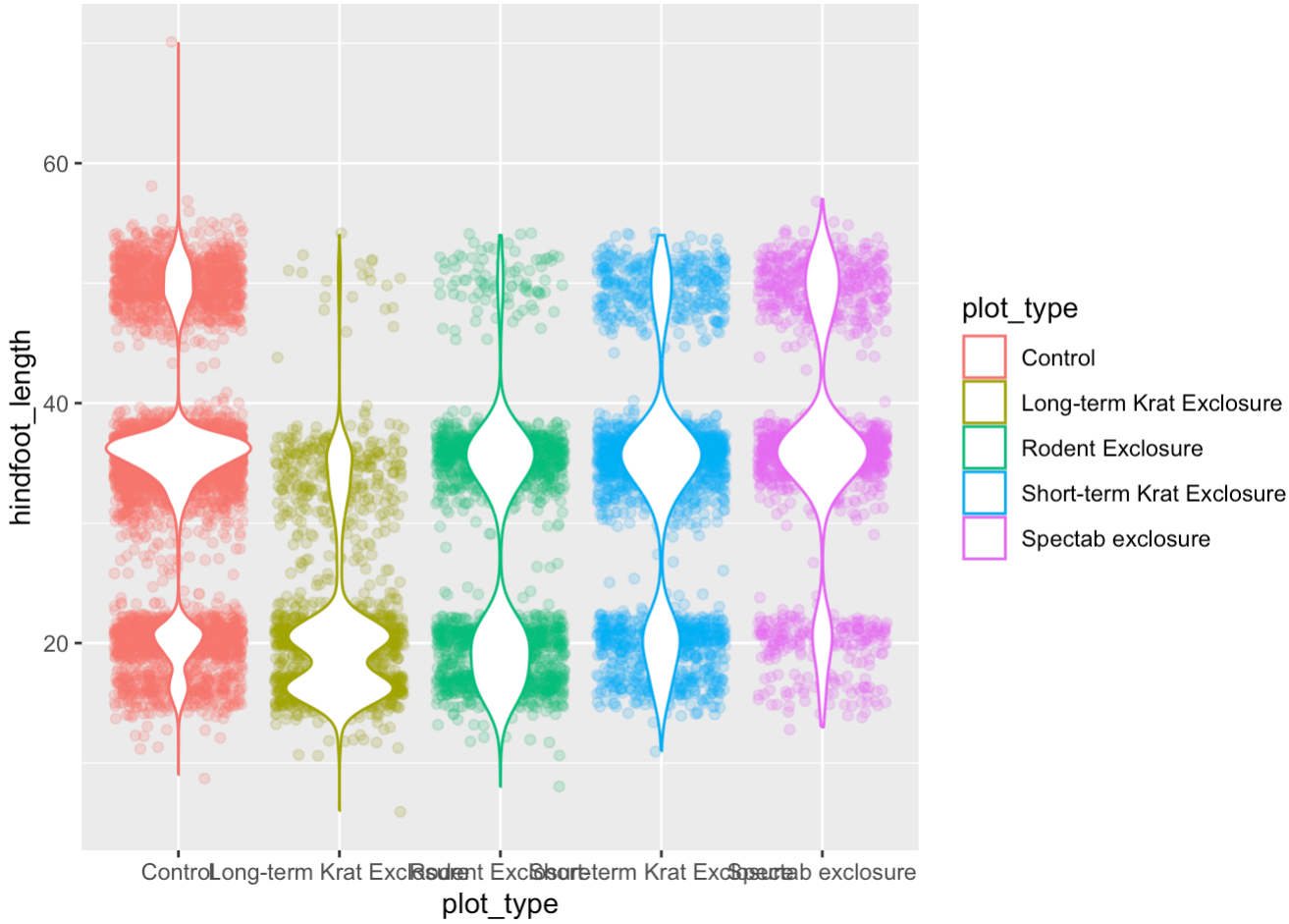
Challenge 2b: Change geoms (continued)

Make the colour of the points and outlines of the violins vary by `plot_type`, and set the `fill` of the violins to white. Try playing with the order of the layers to see what looks best.

```
ggplot(data = complete_old,
       mapping = aes(x = plot_type,
                     y = hindfoot_length,
                     color = plot_type)) +
  geom_jitter(alpha = 0.2) +
  geom_violin(fill = "white")
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_ydensity()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



2.10 Assigning the plot to an object

So far we've been changing the appearance of parts of our plot related to our data and the `geom_` functions, but we can also change many of the non-data components of our plot.

At this point, we are pretty happy with the basic layout of our plot, so we can assign the plot to a named `object`. We do this using the `assignment arrow <-`. What we are doing here is taking the result of the code on the right side of the arrow, and assigning it to an object whose name is on the left side of the arrow.

Tip - Assignment operator

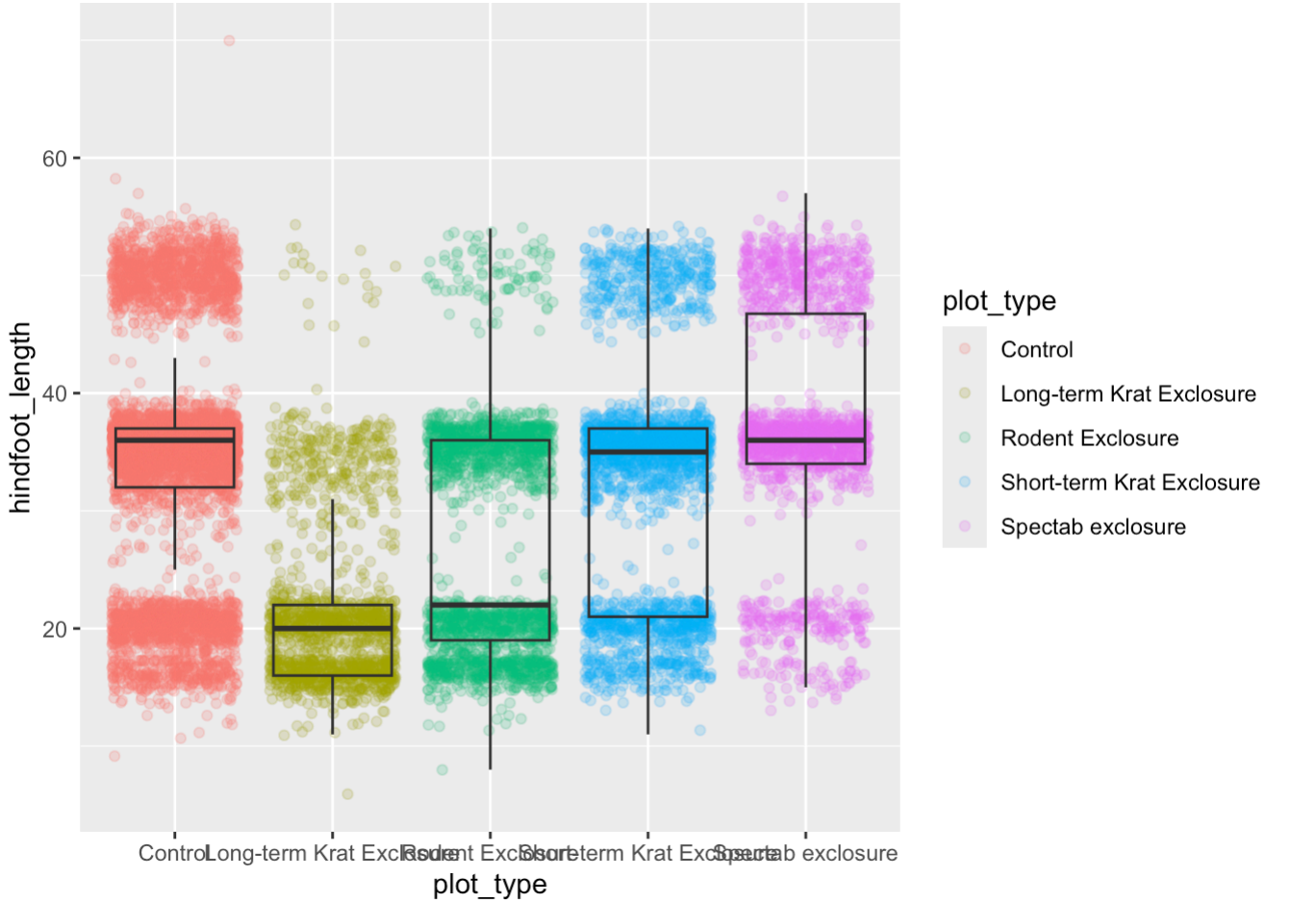
In RStudio, typing `Alt + -` or OS X: `Option + -` will write `<-` in a single keystroke

We will create an object called `myplot` for our plot. If you run the name of the **`ggplot2`** object, it will show the plot, just like if you ran the code itself.

```
# create myplot

myplot <- ggplot(data = complete_old, mapping = aes(x = plot_type, y = hindfoot_lengt
h)) +
  geom_jitter(aes(colour = plot_type), alpha = 0.2) +
  geom_boxplot(outlier.shape = NA, fill = NA)

myplot
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



This process of assigning something to an `object` is not specific to **`ggplot2`**, but rather a general feature of R. We will be using it a lot in the rest of this lesson. We can now work with the `myplot` object as if it was a block of **`ggplot2`** code, which means we can use `+` to add new components to it.

Tip - Environment tab

In the *Environment* tab you can see the current values for variables in memory.

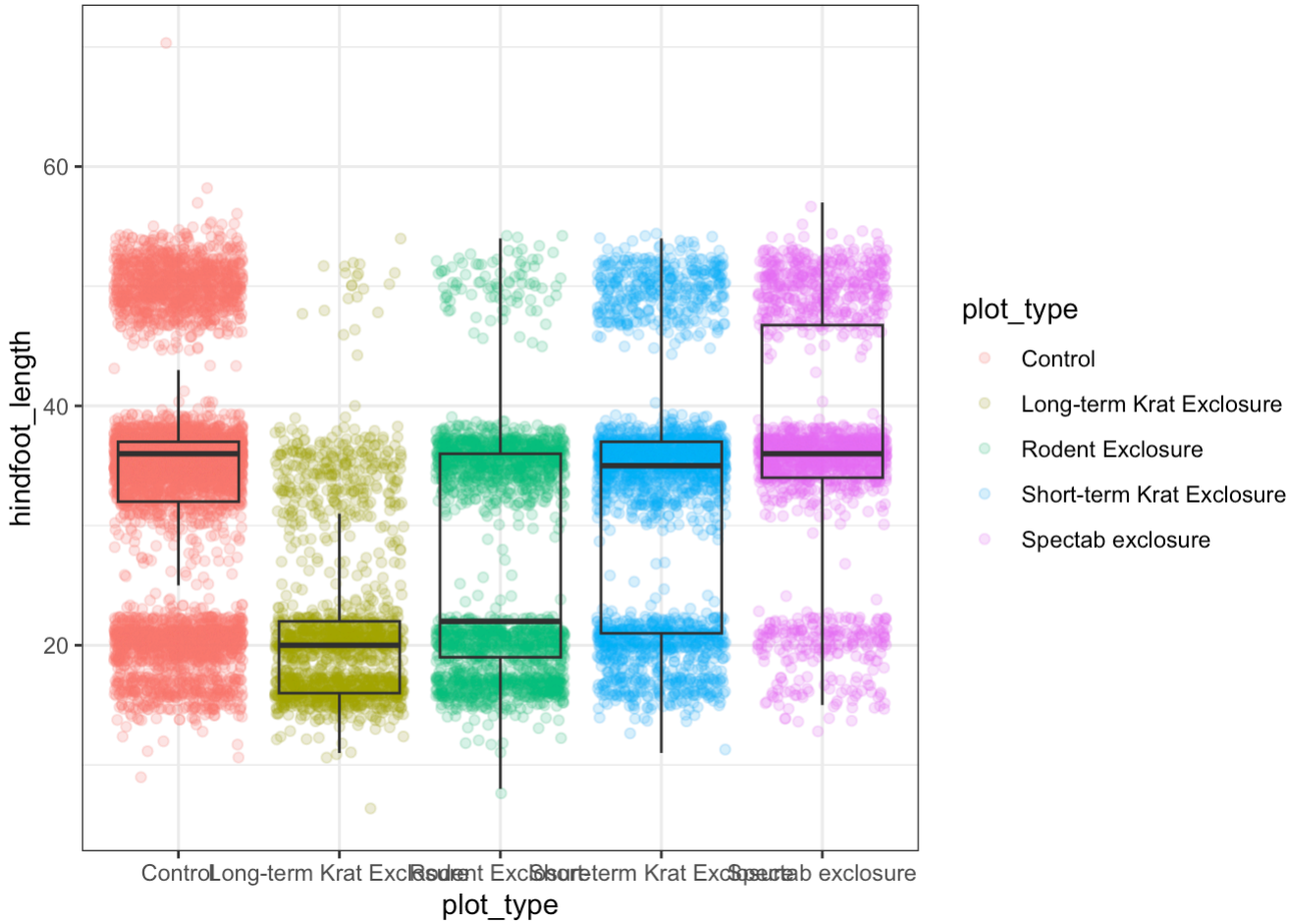
2.11 Changing themes

So far we've been changing the appearance of parts of our plot related to our data and the `geom_` functions, but we can also change many of the non-data components of our plot. We can change the overall appearance of our plot using `theme_` functions. Let's try a black-and-white theme by adding `theme_bw()` to our plot:

```
# add theme

myplot + theme_bw()
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```

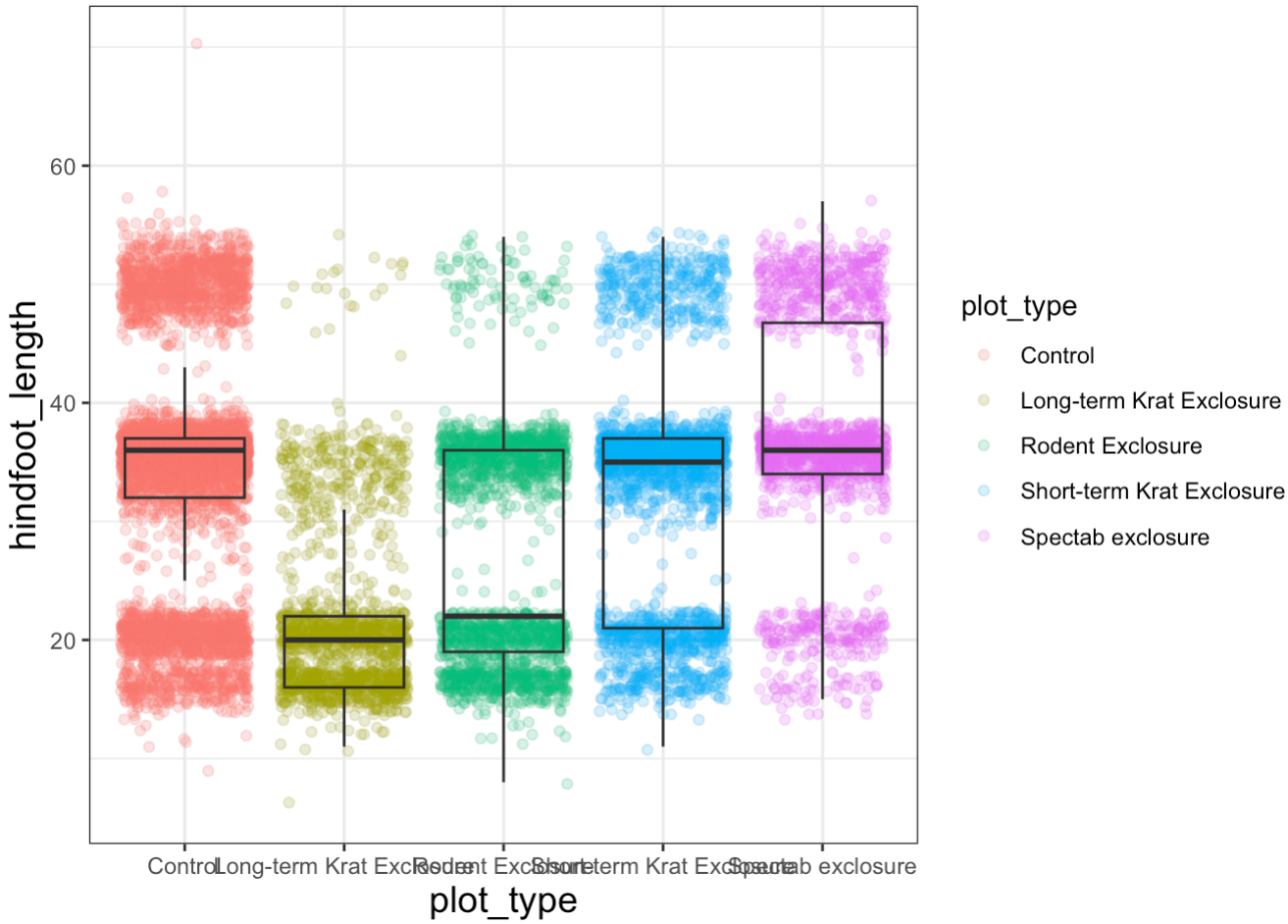


As you can see, a number of parts of the plot have changed. `theme_` functions usually control many aspects of a plot's appearance all at once, for the sake of convenience. To individually change parts of a plot, we can use the `theme()` function, which can take many different arguments to change things about the text, grid lines, background colour, and more. Let's try changing the size of the text on our axis titles. We can do this by specifying that the `axis.title` should be an `element_text()` with `size` set to 14.

```
# customise theme

myplot +
  theme_bw() +
  theme(axis.title = element_text(size = 14))
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```

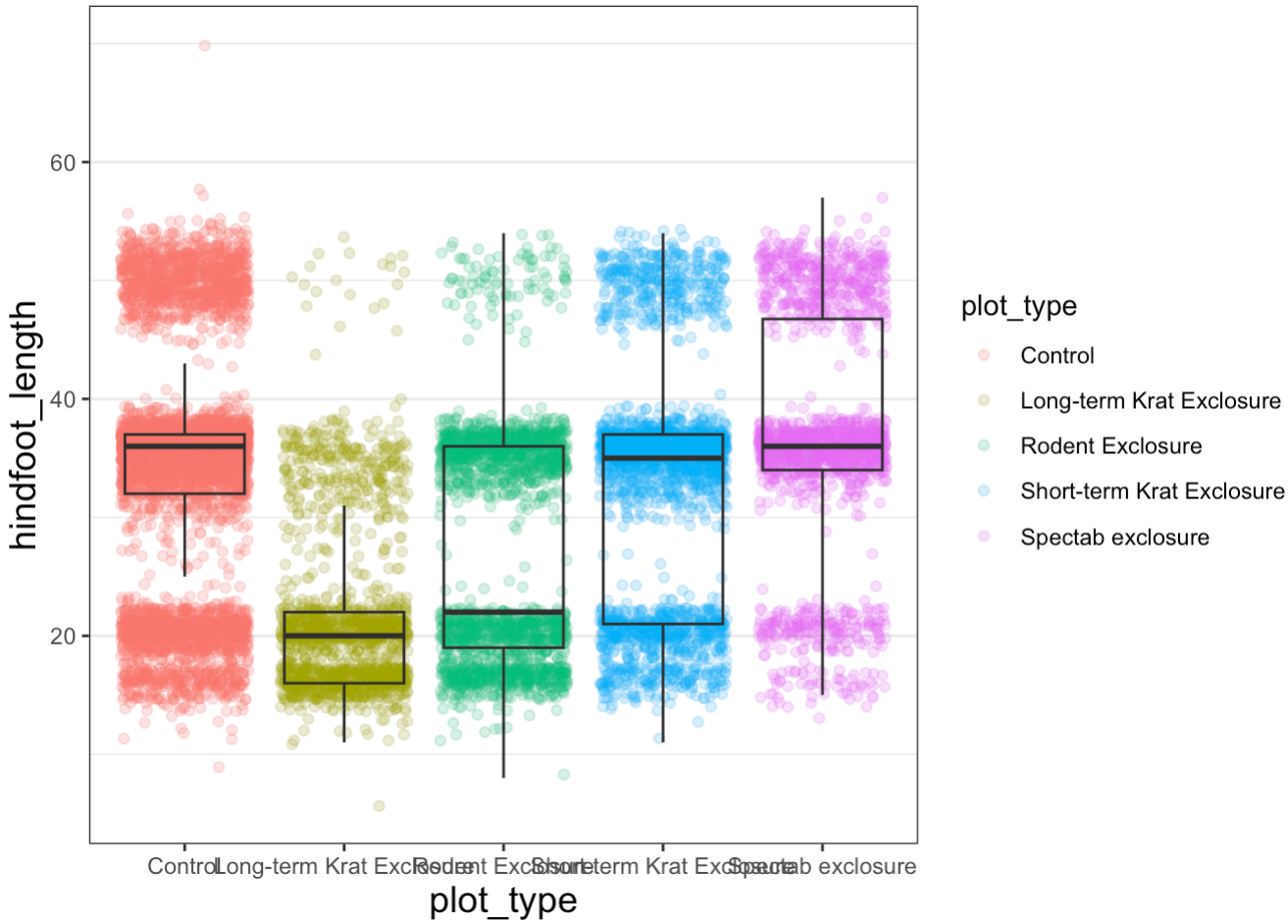


Another change we might want to make is to remove the vertical grid lines. Since our x axis is categorical, those grid lines aren't useful. To do this, inside `theme()`, we will change the `panel.grid.major.x` to an `element_blank()`.

```
# remove grid lines

myplot +
  theme_bw() +
  theme(axis.title = element_text(size = 14),
        panel.grid.major.x = element_blank())
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```

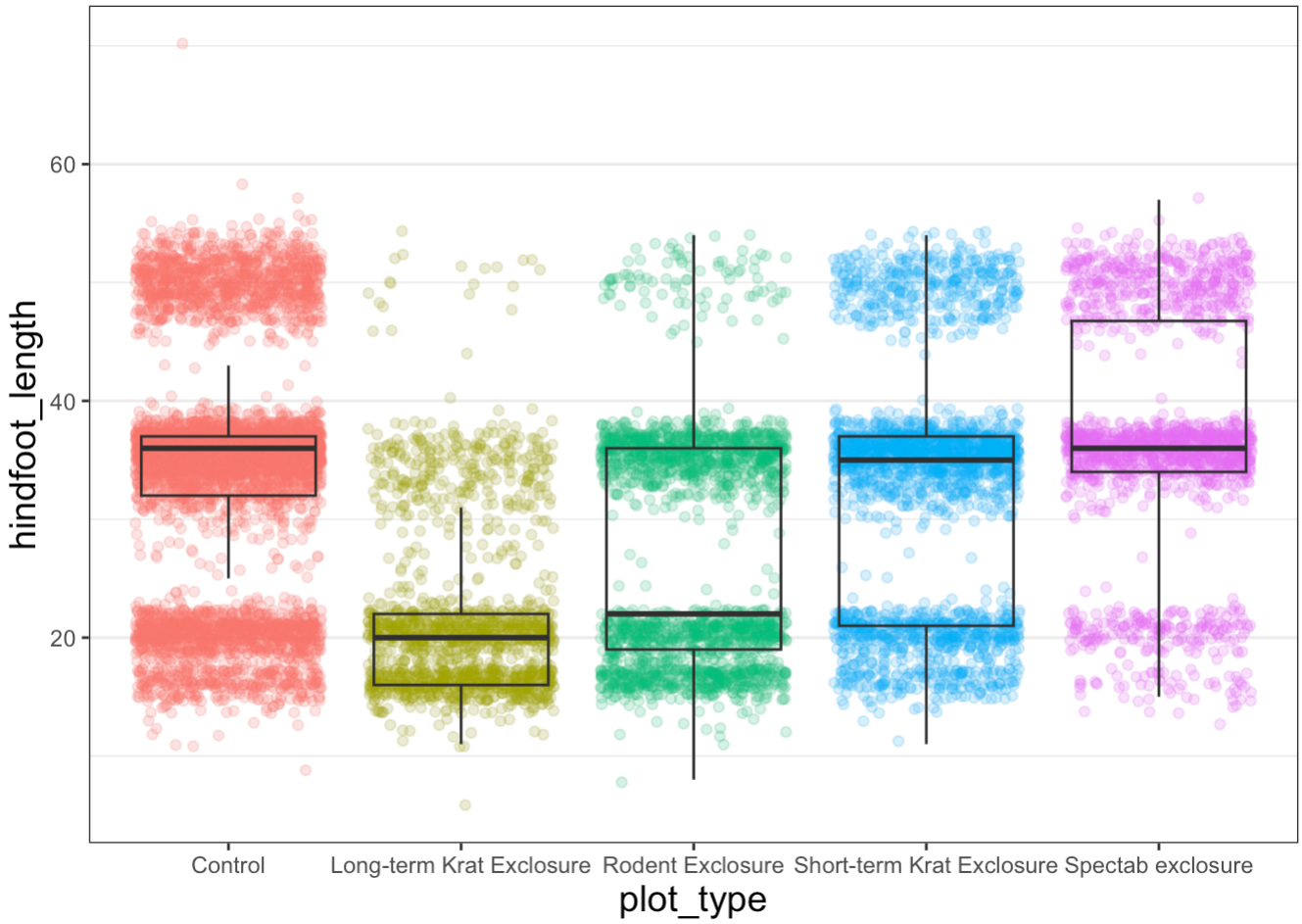


Another useful change might be to remove the colour legend, since that information is already on our x axis. For this one, we will set `legend.position` to "none".

```
# remove legend

myplot +
  theme_bw() +
  theme(axis.title = element_text(size = 14),
        panel.grid.major.x = element_blank(),
        legend.position = "none")
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



Tip - Themes

Because there are so many possible arguments to the `theme()` function, it can sometimes be hard to find the right one. Here are some tips for figuring out how to modify a plot element:

- type out `theme()`, put your cursor between the parentheses, and hit `Tab` to bring up a list of arguments
    - you can scroll through the arguments, or start typing, which will shorten the list of potential matches
    - like many things in the `tidyverse`, similar argument start with similar names
        - there are `axis`, `legend`, `panel`, `plot`, and `strip` arguments
    - arguments have hierarchy
        - `text` controls all text in the whole plot
        - `axis.title` controls the text for the axis titles
        - `axis.title.x` controls the text for the x axis title

Tip - Many ways to do the same thing

You may have noticed that we have used 3 different approaches to getting rid of something in `ggplot`:

- `outlier.shape = NA` to remove the outliers from our boxplot
- `panel.grid.major.x = element_blank()` to remove the x grid lines
- `legend.position = "none"` to remove our legend

Why are there so many ways to do what seems like the same thing?? This is a common frustration when working with R, or with any programming language. There are a couple reasons for it:

1. Different people contribute to different packages and functions, and they may choose to do things differently.

2. Code may appear to be doing the same thing, when the details are actually quite different. The inner workings of ggplot2 are actually quite complex, since it turns out making plots is a very complicated process! Because of this, things that seem the same (removing parts of a plot), may actually be operating on very different components or stages of the final plot.
3. Developing packages is a highly iterative process, and sometimes things change. However, changing too much stuff can make old code break. Let's say removing the legend was introduced as a feature of ggplot2, and then a lot of time passed before someone added the feature letting you remove outliers from geom_boxplot(). Changing the way you remove the legend, so that it's the same as the boxplot approach, could break all of the code written in the meantime, so developers may opt to keep the old approach in place.

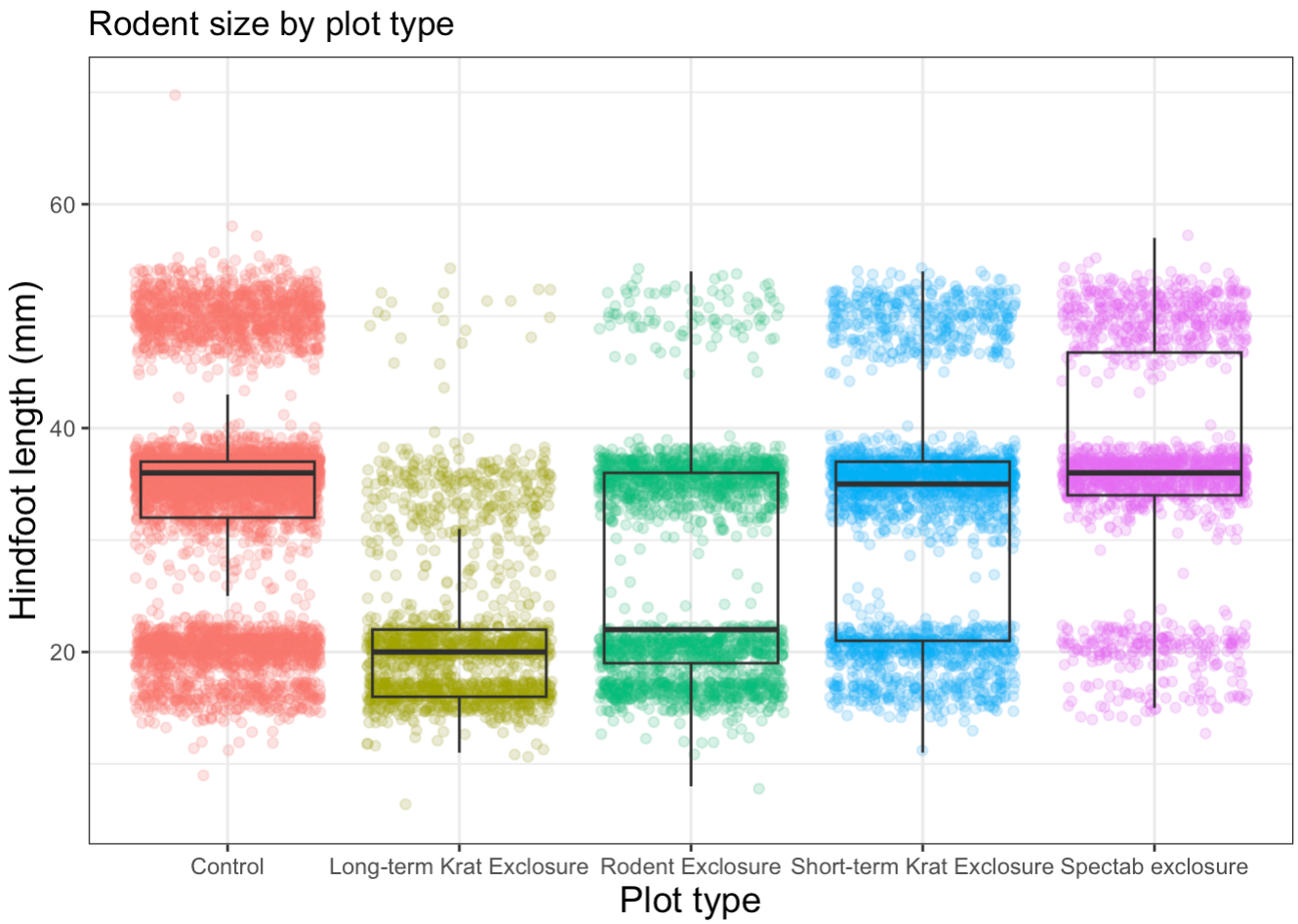
2.12 Changing labels

Our plot is really shaping up now. However, we probably want to make our axis titles nicer, and perhaps add a main title to the plot. We can do this using the `labs()` function:

```
myplot +
  theme_bw() +
  theme(axis.title = element_text(size = 14),
        legend.position = "none") +
  labs(title = "Rodent size by plot type",
       x = "Plot type",
       y = "Hindfoot length (mm)")
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



We removed our legend from this plot, but you can also change the titles of various legends using labs(). For example, `labs(colour = "Plot type")` would change the title of a colour scale legend to “Plot type”.

Challenge 3: Customising a plot

Modify the previous plot by adding a descriptive subtitle. Increase the font size of the plot title and make it bold.

Hint: “bold” is referred to as a font “face”

```
myplot +
  theme_bw() +
  theme(axis.title = element_text(size = 14), legend.position = "none",
        plot.title = element_text(face = "bold", size = 20)) +
  labs(title = "Rodent size by plot type",
       subtitle = "Long-term dataset from Portal, AZ",
       x = "Plot type",
       y = "Hindfoot length (mm)")
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```

2.13 Faceting

One of the most powerful features of `ggplot` is the ability to quickly split a plot into multiple smaller plots based on a categorical variable, which is called **faceting**.

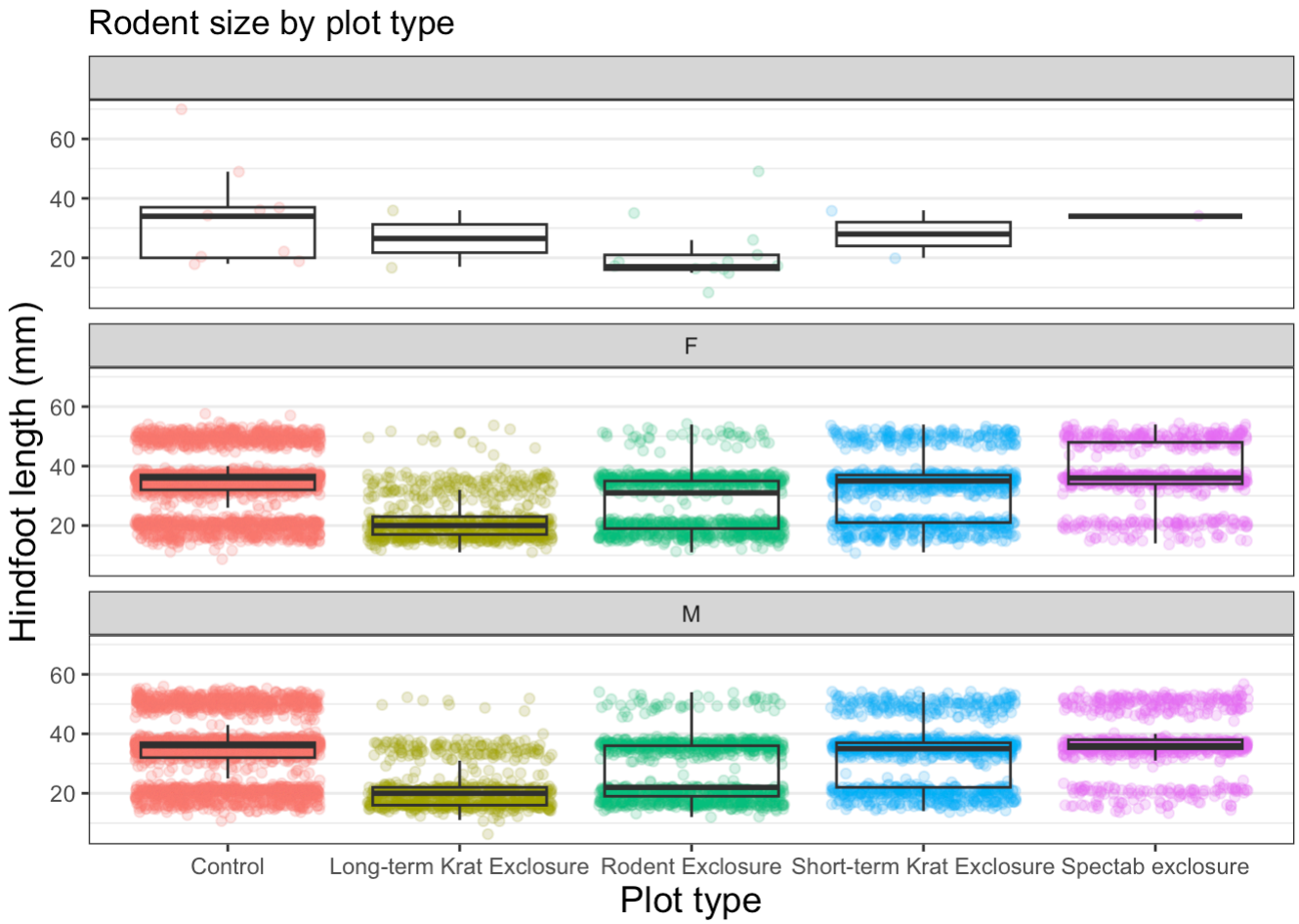
So far we've mapped variables to the x axis, the y axis, and colour, but trying to add a 4th variable becomes difficult. Changing the shape of a point might work, but only for very few categories, and even then, it can be hard to tell the differences between the shapes of small points.

Instead of cramming one more variable into a single plot, we will use the `facet_wrap()` function to generate a series of smaller plots, split out by sex. We also use `ncol` to specify that we want them arranged in a single column:

```
# add facet_wrap

myplot +
  theme_bw() +
  theme(axis.title = element_text(size = 14),
        legend.position = "none",
        panel.grid.major.x = element_blank()) +
  labs(title = "Rodent size by plot type",
       x = "Plot type",
       y = "Hindfoot length (mm)",
       colour = "Plot type") +
  facet_wrap(vars(sex), ncol = 1)
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



Tip - Faceting

Faceting comes in handy in many scenarios. It can be useful when:

- a categorical variable has too many levels to differentiate by colour (such as a dataset with 20 countries)
- your data overlap heavily, obscuring categories
- you want to show more than 3 variables at once
- you want to see each category in isolation while allowing for general comparisons between categories

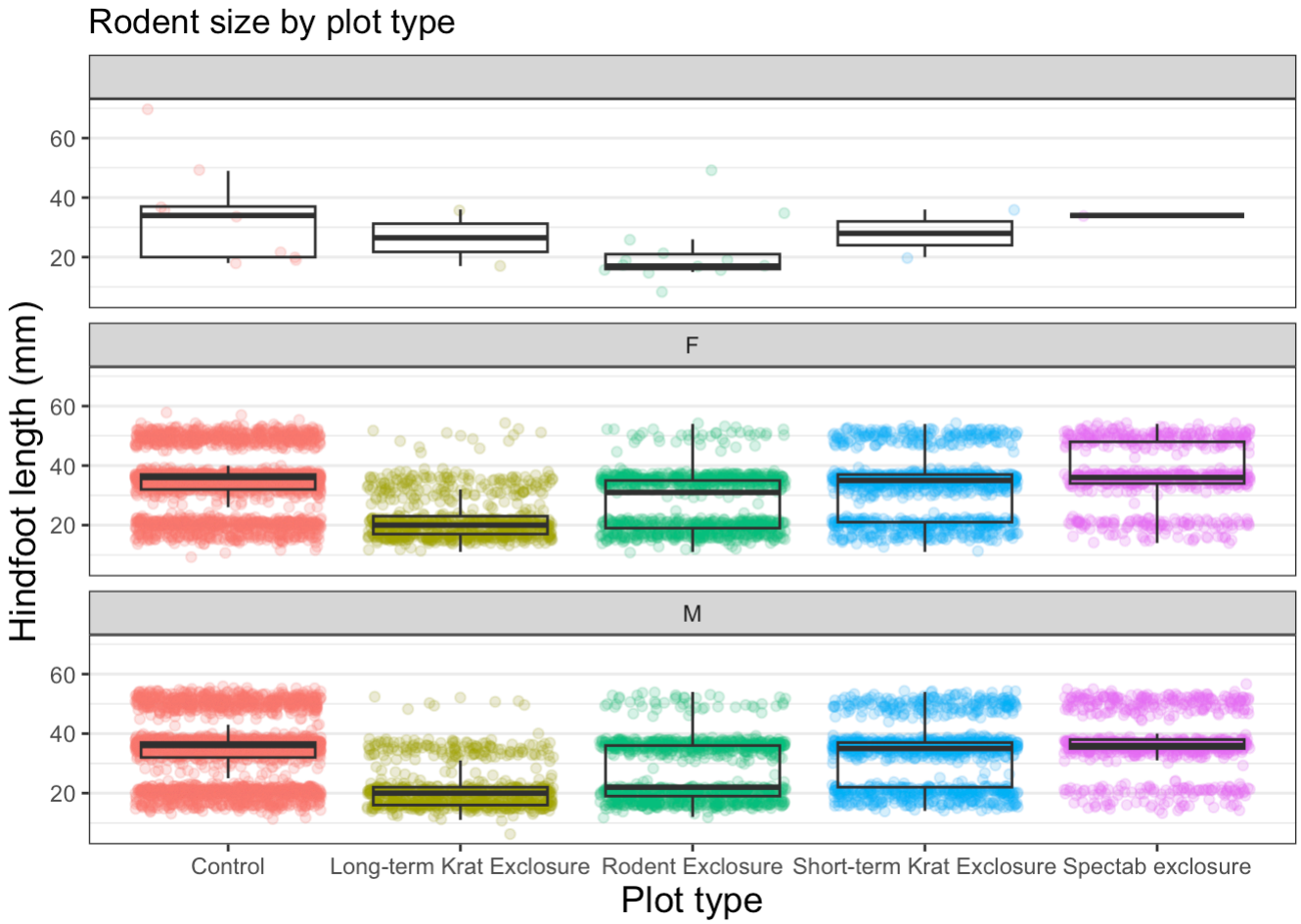
2.14 Exporting plots

Once we are happy with our final plot, we can assign the whole thing to a new object, which we can call `finalplot`.

```
finalplot <- myplot +
  theme_bw() +
  theme(axis.title = element_text(size = 14),
        legend.position = "none",
        panel.grid.major.x = element_blank()) +
  labs(title = "Rodent size by plot type",
       x = "Plot type",
       y = "Hindfoot length (mm)",
       colour = "Plot type") +
  facet_wrap(vars(sex), ncol = 1)

finalplot
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```



After this, we can run `ggsave()` to save our plot. The first argument we give is the path to the file we want to save, including the correct file extension. This code will make an image called `rodent_size_plots.jpg` in the `images/` folder of our current project. We are making a `.jpg`, but you can save `.pdf`, `.tiff`, and other file formats. Next, we tell it the name of the plot object we want to save. We can also specify things like the `width` and `height` of the plot in inches.

If you do not have an images folder, one will be created for you, if you enter '1' in the console.

```
# save plot

ggsave(filename = "images/rodent_size_plots.jpg", plot = finalplot,
       height = 6, width = 8)
```

```
## Warning: Removed 2733 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

```
## Warning: Removed 2733 rows containing missing values or values outside the scale r
ange
## (`geom_point()`).
```

Challenge 4: Make your own plot!

Try making your own plot! You can run `str(complete_old)` or `?complete_old` to explore variables you might use in your new plot. Feel free to use variables we have already seen, or some we haven't explored yet.

Here are a couple ideas to get you started:

- make a histogram of one of the numeric variables
- try using a different colour scale_
- try changing the size of points or thickness of lines in a geom

```
# create your own plot! Go wild!
```

Key points

- The `ggplot()` function initiates a plot, and `geom_` functions add representations of your data
- use `aes()` when mapping a variable from the data to a part of the plot
- use `scale_` functions to modify the scales used to represent variables
- use premade `theme_` functions to broadly change appearance, and the `theme()` function to fine-tune
- start simple and build your plots iteratively

3. Interactive plotting with Plotly

Note, this is extra content not included in the Carpentries lesson.

Plotly is an R package (https://plotly.com/ggplot2/getting-started/) for creating interactive web-based graphs via plotly's JavaScript graphing library, `plotly.js` .

Furthermore, you have the option of manipulating the Plotly object with the `style` function.

Tip - Behind the scenes of Plotly

The plotly R package serializes ggplot2 figures into Plotly's universal graph JSON. `plotly::ggplotly` will crawl the ggplot2 figure, extract and translate all of the attributes of the ggplot2 figure into JSON (the colours, the axes, the chart type, etc), and draw the graph with plotly.js.

```
library(plotly)
```

```
##
## Attaching package: 'plotly'
```

```
## The following object is masked from 'package:ggplot2':
##
##     last_plot
```

```
## The following object is masked from 'package:stats':
##
##     filter
```

```
## The following object is masked from 'package:graphics':
##
##     layout
```

Simply printing the Plotly object will render the chart locally in your web browser or in the RStudio viewer.

Plotly graphs are interactive! You can click on legend entries to toggle traces, click-and-drag on the chart to zoom, double-click to autoscale, and shift-and-drag to pan.

Let's create a new `ggplot` object called myplot_final, and wrap it in the `ggplotly` function:

```
# use ggplotly to convert a ggplot to an interactive visualisation

my_interactive_plot <- ggplot(data = complete_old, mapping = aes(x = weight, y = hind
foot_length, colour = plot_type)) +
   geom_point(alpha = 0.2) +
   theme_bw()

ggplotly(my_interactive_plot)
```