# Data visualization with ggplot2 - Answers

This lesson is adapted from the Data Carpentry Ecology Lesson 4.

This is an R Markdown Notebook. When you execute code within the notebook, the results appear beneath the code.

Execute code chunks by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing *Cmd+Shift+Enter* or *Ctrl+Shift+Enter*.

Add a new chunk by clicking the *Insert Chunk* button on the toolbar or by pressing *Cmd+Option+I*.

When you save the notebook, a HTML file containing the code and output will be saved alongside it (click the *Preview* button or press *Cmd+Shift+K* or *Ctrl+Shift+K* to preview the HTML file).

The preview shows you a rendered HTML copy of the contents of the editor. Consequently, unlike *Knit*, *Preview* does not run any R code chunks. Instead, the output of the chunk when it was last run in the editor is displayed.

## Plotting with ggplot2

We start by loading the required packages. **ggplot2** is included in the **tidyverse** package.

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.2     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.2     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

Next, we load the data that we prepared in the previous lesson.

```
surveys_complete <- read_csv("data/surveys_complete.csv")
```

```
## Rows: 30463 Columns: 13
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (6): species_id, sex, genus, species, taxa, plot_type
## dbl (7): record_id, month, day, year, plot_id, hindfoot_length, weight
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

**ggplot2** is a plotting package that makes it simple to create complex plots from data in a data frame. It provides a more programmatic interface for specifying what variables to plot, how they are displayed, and general visual properties. Therefore, we only need minimal changes if the underlying data change or if we

decide to change from a bar plot to a scatter plot. This helps in creating publication quality plots with minimal amounts of adjustments and tweaking.

**ggplot2** functions like data in the 'long' format, i.e., a column for every dimension, and a row for every observation. Well-structured data will save you lots of time when making figures with **ggplot2**

ggplot graphics are built step by step by adding new elements. Adding layers in this fashion allows for extensive flexibility and customisation of plots.

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

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

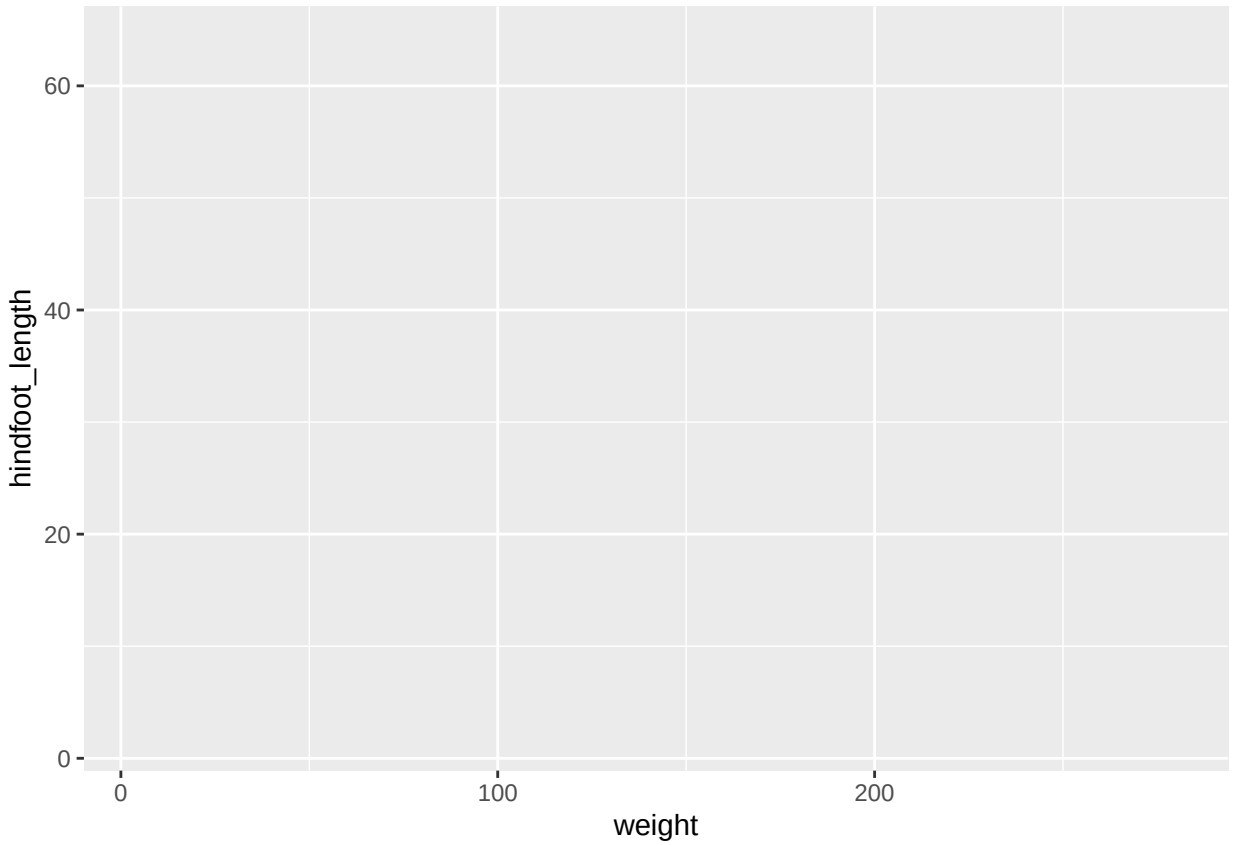
- use the `ggplot()` function and bind the plot to a specific data frame using the `data` argument.

```
# ggplot with data
ggplot(data = surveys_complete)
```



- define a mapping (using the aesthetic (`aes`) function), by selecting the variables to be plotted and specifying how to present them in the graph, e.g. as x/y positions or characteristics such as size, shape, colour, etc.

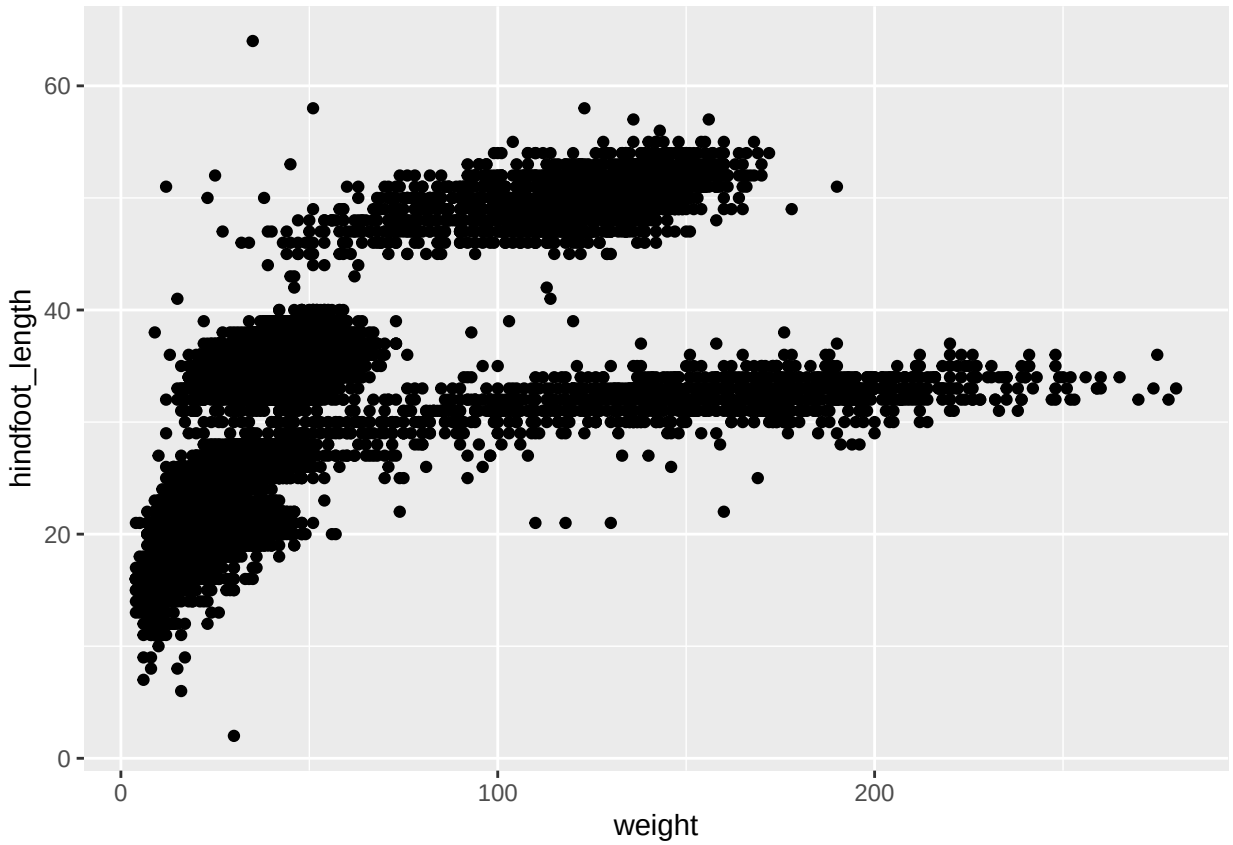
```
# ggplot with data and aes
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length))
```




- add 'geoms' – graphical representations of the data in the plot (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.

To add a geom to the plot use the `+` operator. Because we have two continuous variables, let's use `geom_point()` first:

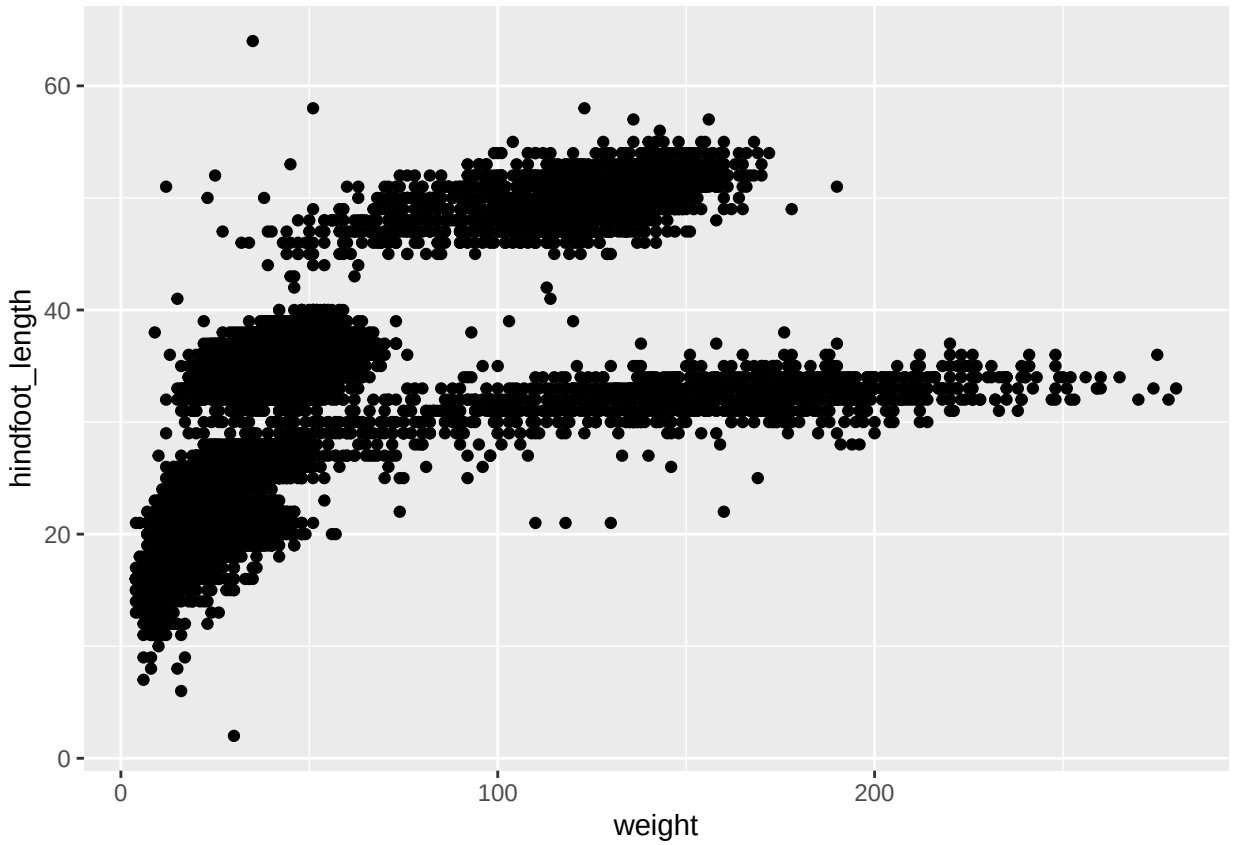
```
# ggplot with data and aes and geom_point
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length)) +
  geom_point()
```



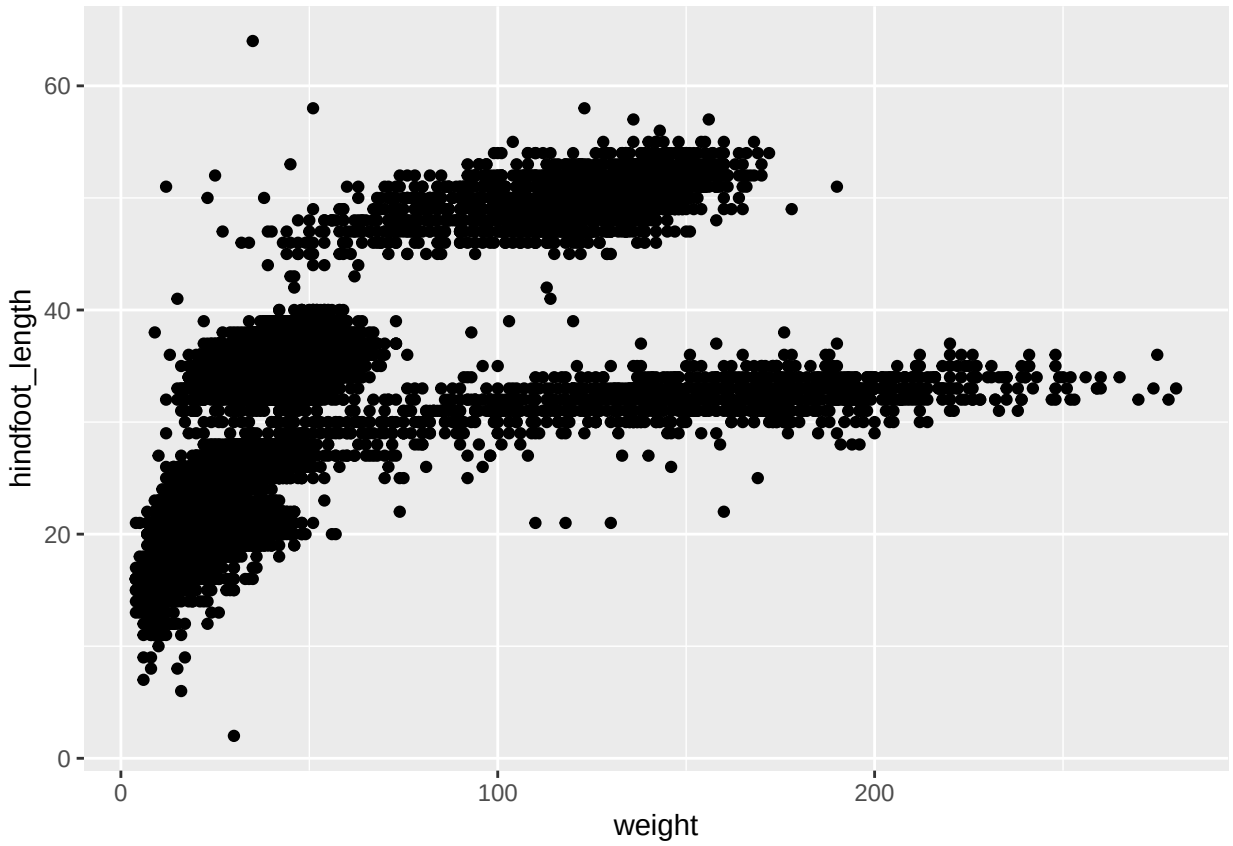

Note: You'll see we have started naming some of our our 'code chunks', like 'first-ggplot'. The name is optional but if included, each code chunk needs a distinct name. The advantage of giving each chunk a name is that it will be easier to understand where to look for errors, should they occur. Also,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

The `+` in the **`ggplot2`** package is particularly useful because it allows you to modify existing `ggplot` objects. This means you can easily set up plot templates and conveniently explore different types of plots, so the above plot can also be generated with code like this:

```
# Assign plot to a variable
surveys_plot <- ggplot(data = surveys_complete,
                       mapping = aes(x = weight, y = hindfoot_length))

# Draw the plot
surveys_plot +
    geom_point()
```



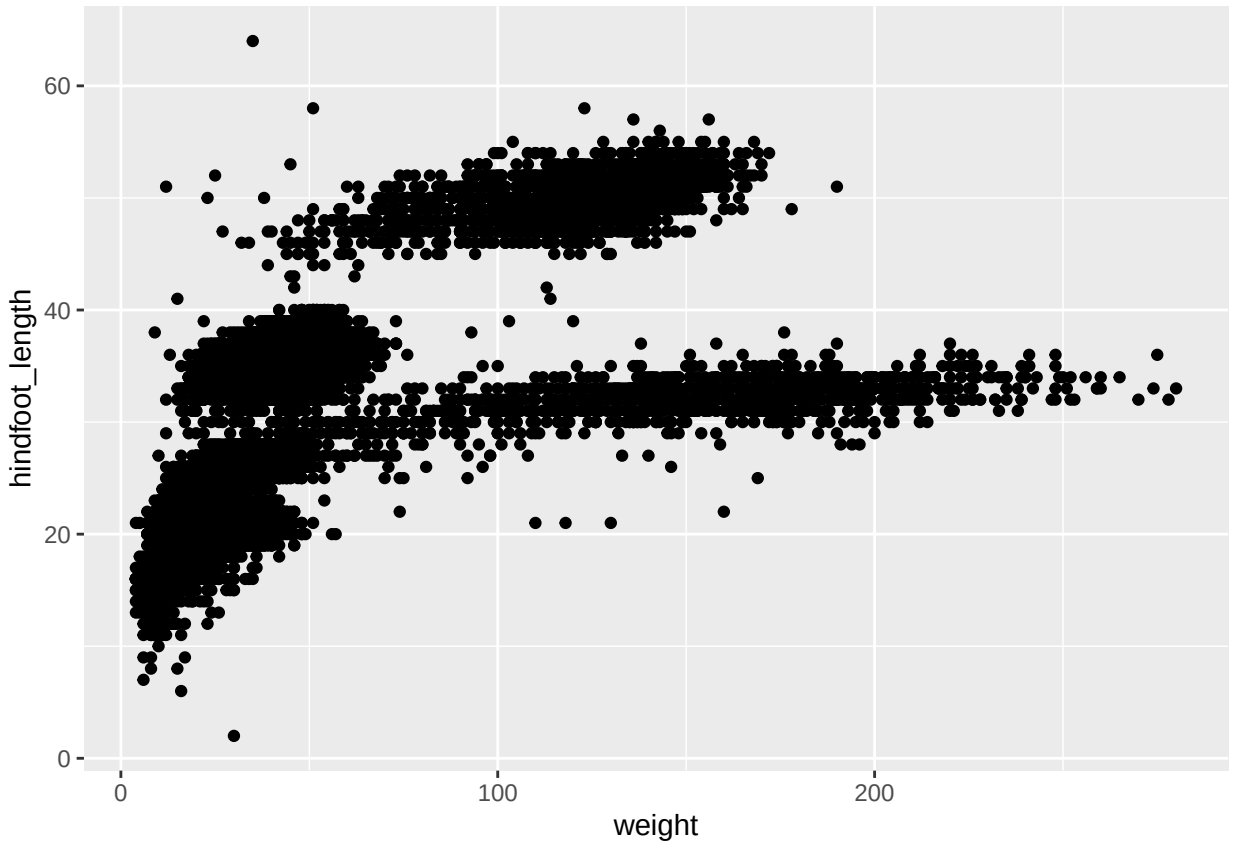

```
# Create a ggplot and draw it
surveys_plot <- ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length))
surveys_plot +
    geom_point()
```




**Notes**

- Anything you put in the `ggplot()` function can be seen by any geom layers that you add (i.e., these are universal plot settings). This includes the x- and y-axis mapping you set up in `aes()`.
- You can also specify mappings for a given geom independently of the mappings defined globally in the `ggplot()` function.
- The `+` sign used to add new layers must be placed at the end of the line containing the *previous* layer. If, instead, the `+` sign is added at the beginning of the line containing the new layer, **`ggplot2`** will not add the new layer and will return an error message.

```
# This is the correct syntax for adding layers
surveys_plot +
  geom_point()
```

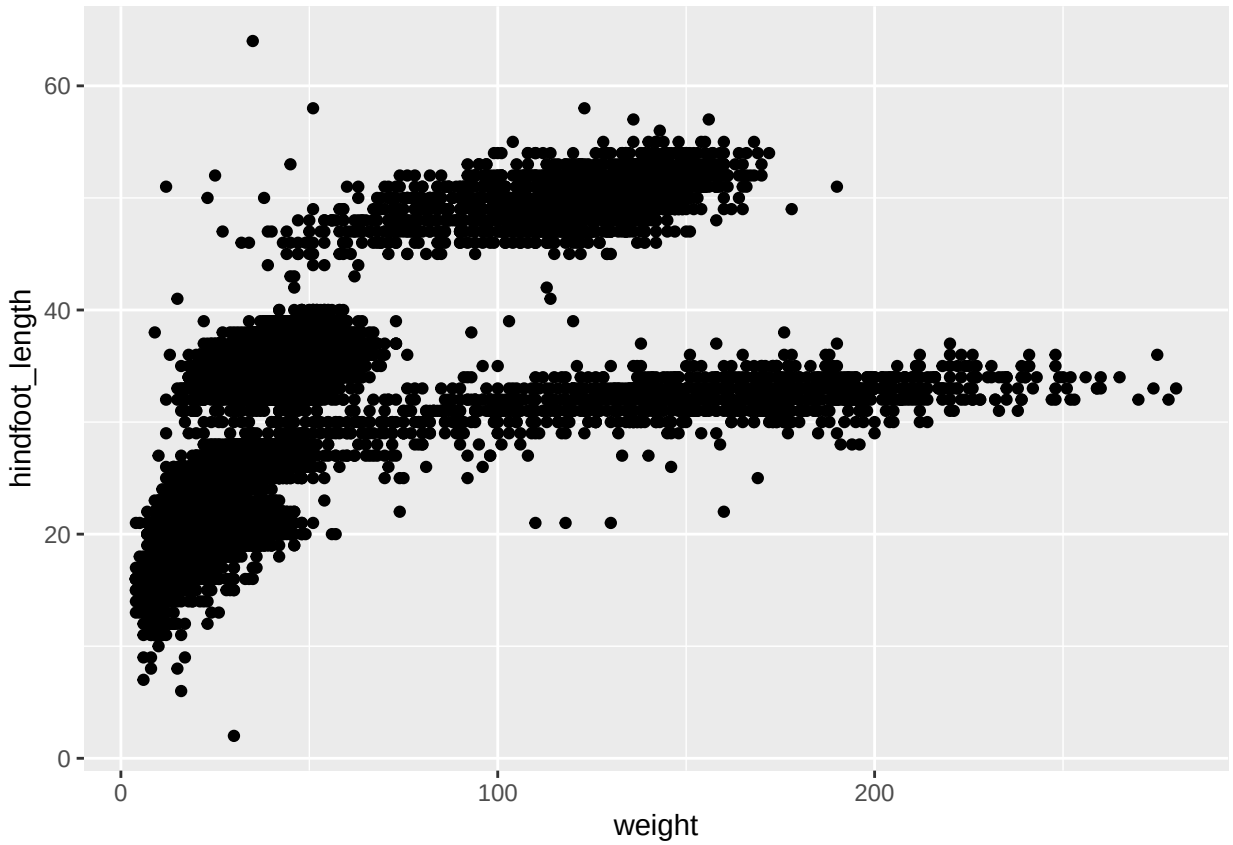



```
# This will not add the new layer and will return an error message
#surveys_plot
#  + geom_point()
```

### Building your plots iteratively

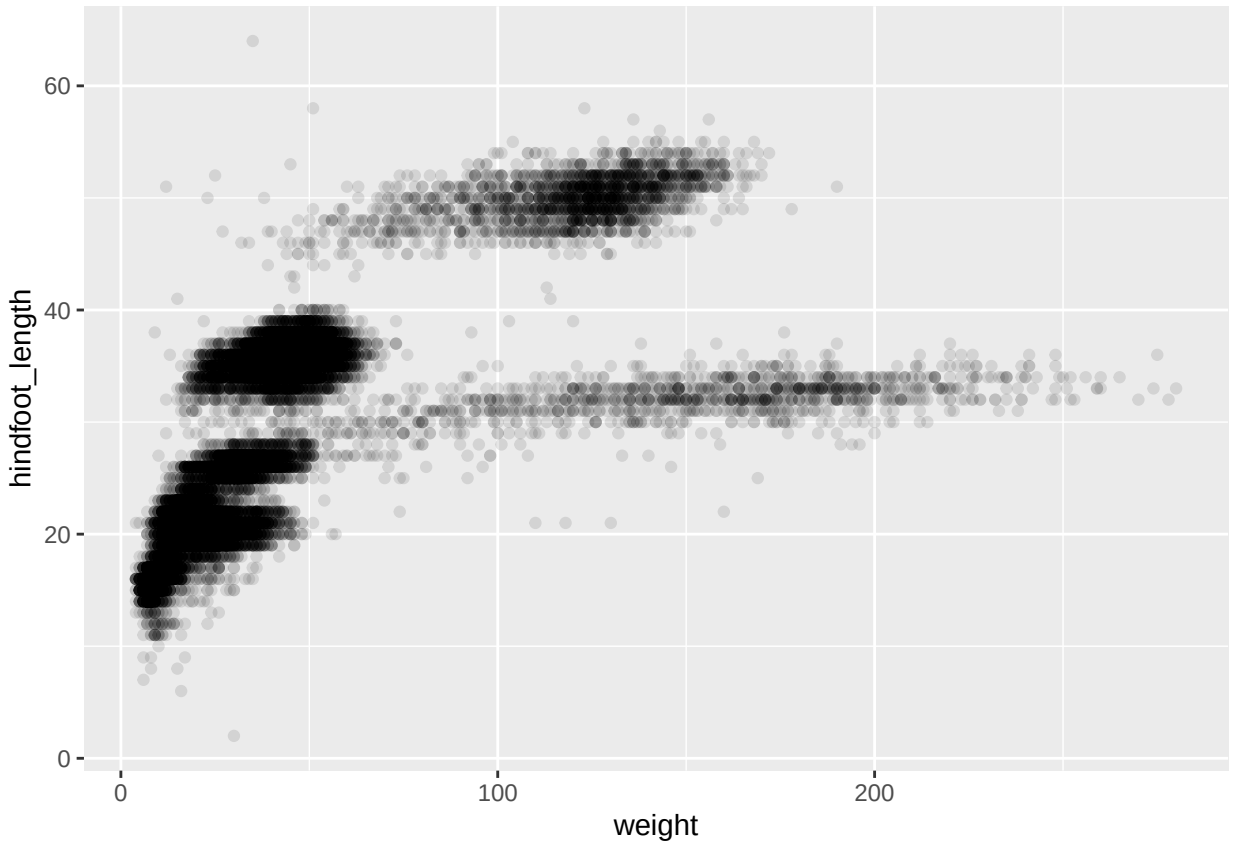
Building plots with **ggplot2** is typically an iterative process. We start by defining the dataset we'll use, lay out the axes, and choose a geom:

```
# create ggplot
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length)) +
    geom_point()
```



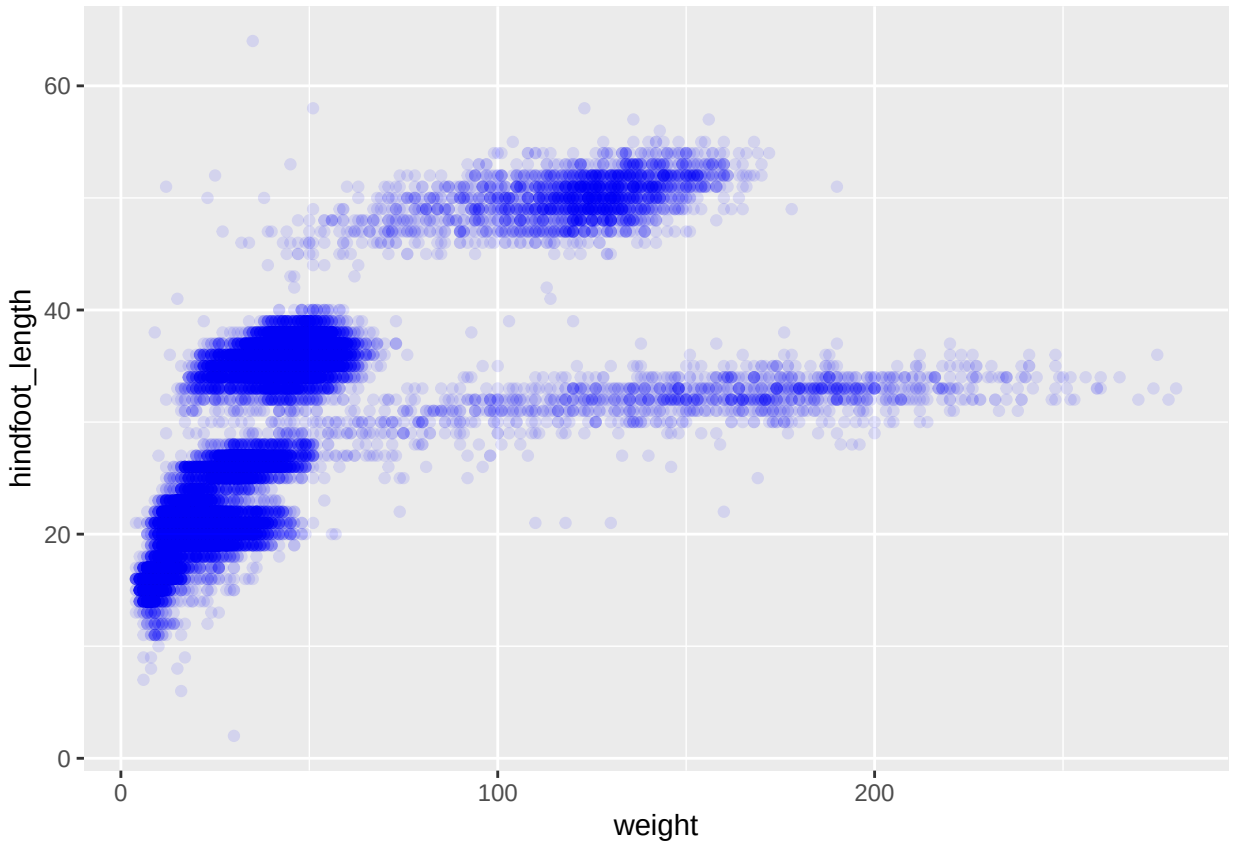

Then, we start modifying this plot to extract more information from it. For instance, we can add transparency (`alpha`) to avoid overplotting:

```
# add transparency
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length)) +
    geom_point(alpha = 0.1)
```



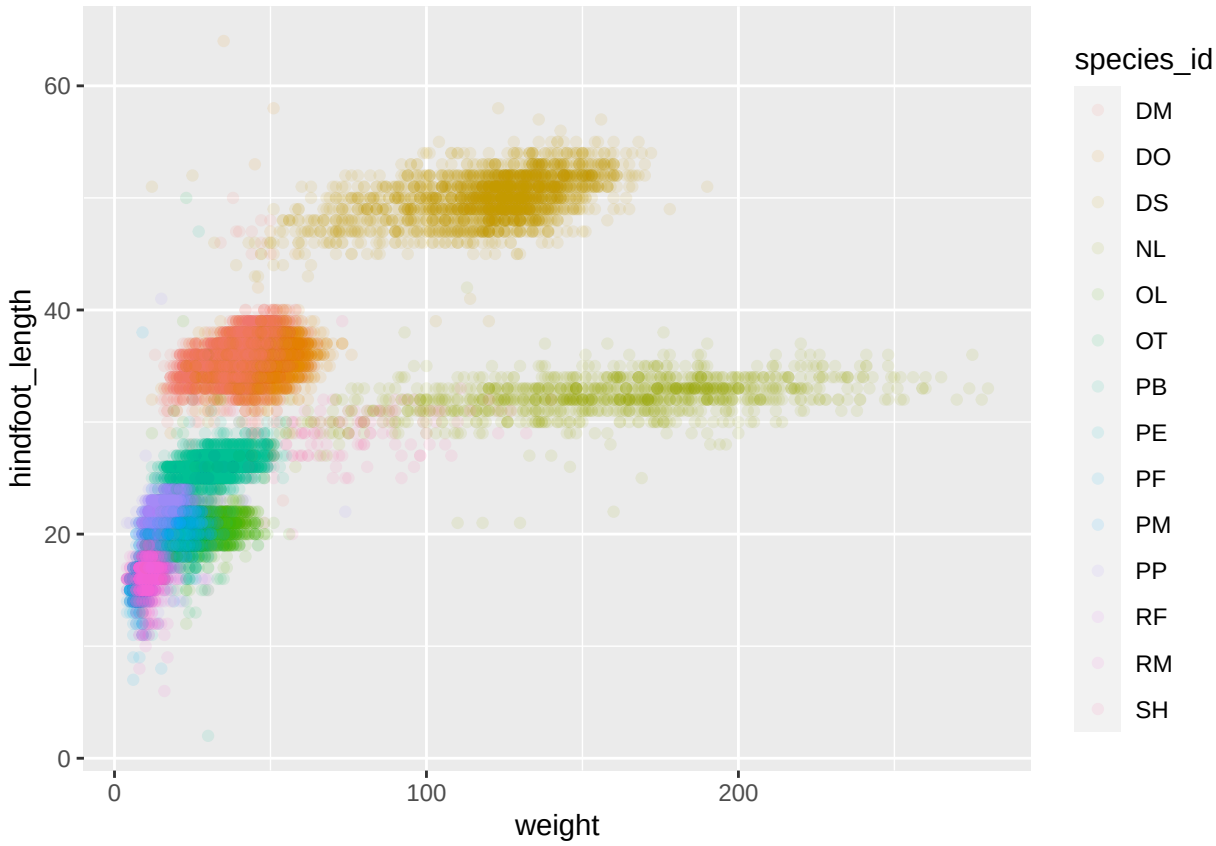

We can also add colours for all the points:

```
# add colours
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length)) +
    geom_point(alpha = 0.1, colour = "blue")
```



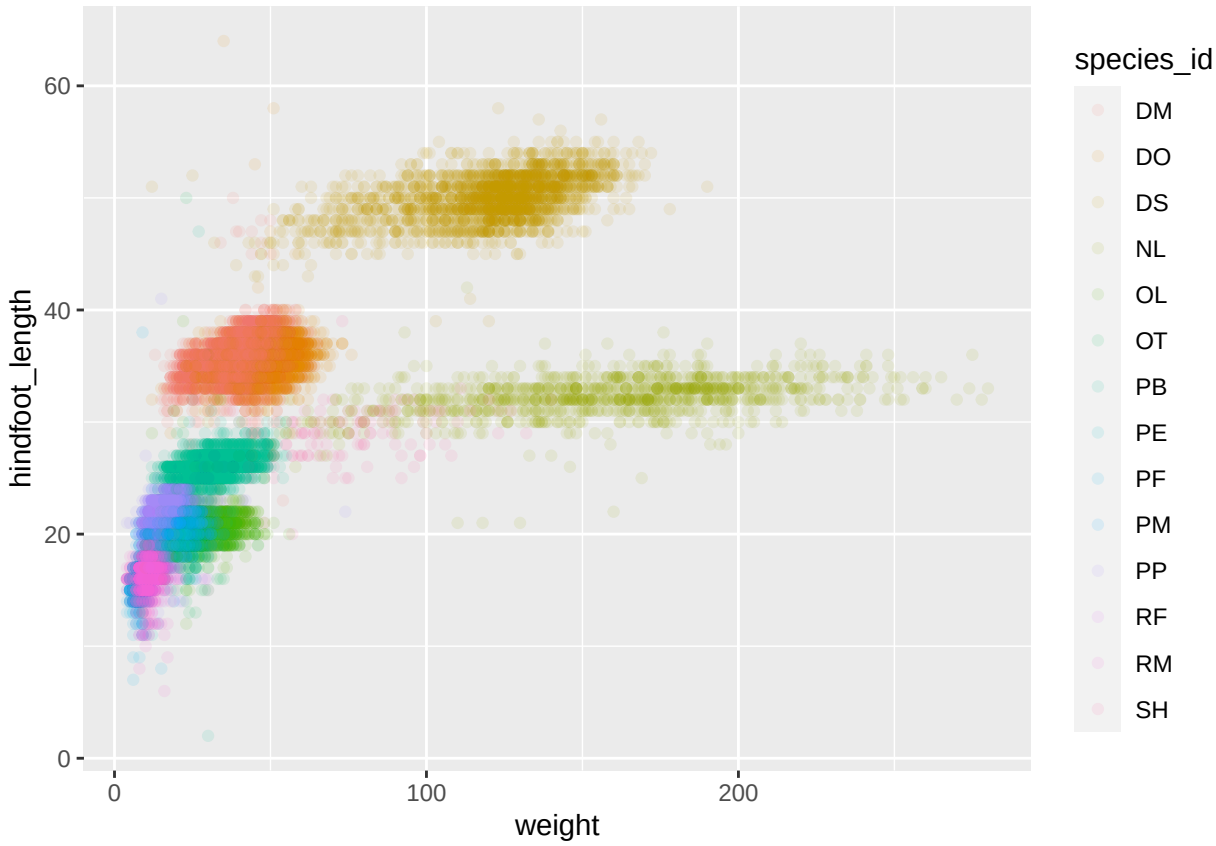

Or to colour each species in the plot differently, you could use a vector as an input to the argument **colour**. **ggplot2** will provide a different colour corresponding to different values in the vector. Here is an example where we colour with **species_id**:

```
# colour by species
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length)) +
    geom_point(alpha = 0.1, aes(colour = species_id))
```



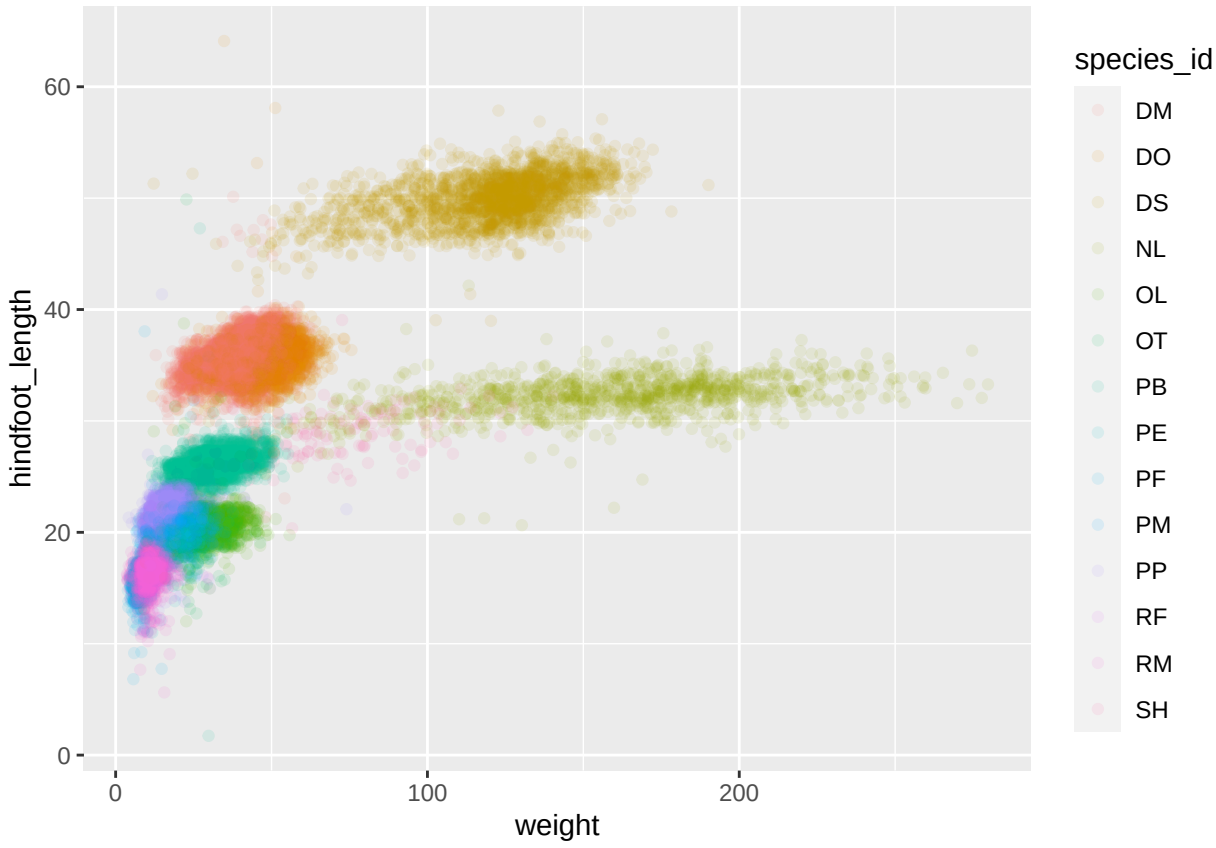

We can also specify the colours directly inside the mapping provided in the `ggplot()` function. This will be seen by any geom layers and the mapping will be determined by the x- and y-axis set up in `aes()`.

```
# colour by species
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length, colour = species_id)) +
    geom_point(alpha = 0.1)
```




Notice that we can change the geom layer and colours will be still determined by **species_id**

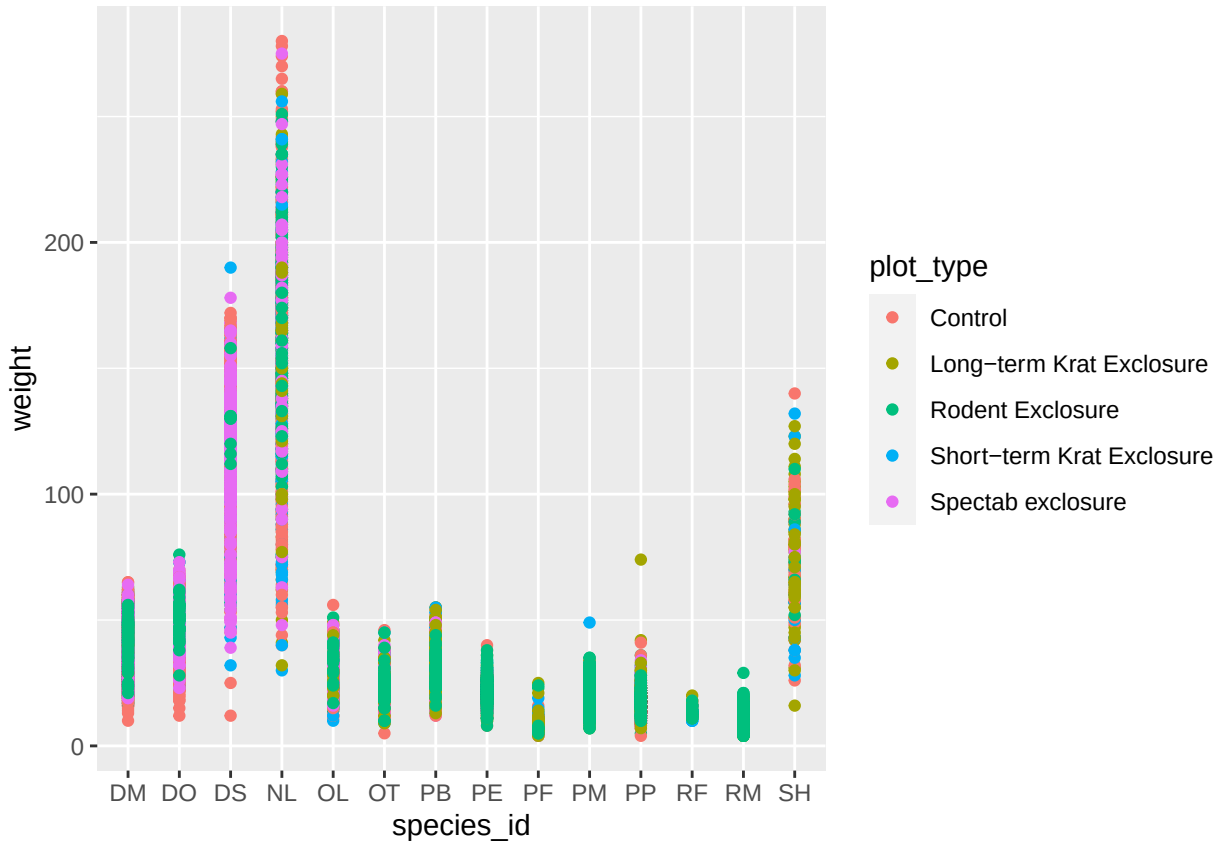
```
# change layers
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length, colour = species_id)) +
    geom_jitter(alpha = 0.1)
```




**Individual Challenge**

Use what you just learned to create a scatter plot of `weight` over `species_id` with the plot types shown in different colours. Is this a good way to show this type of data? If not, what type of plot would you suggest?

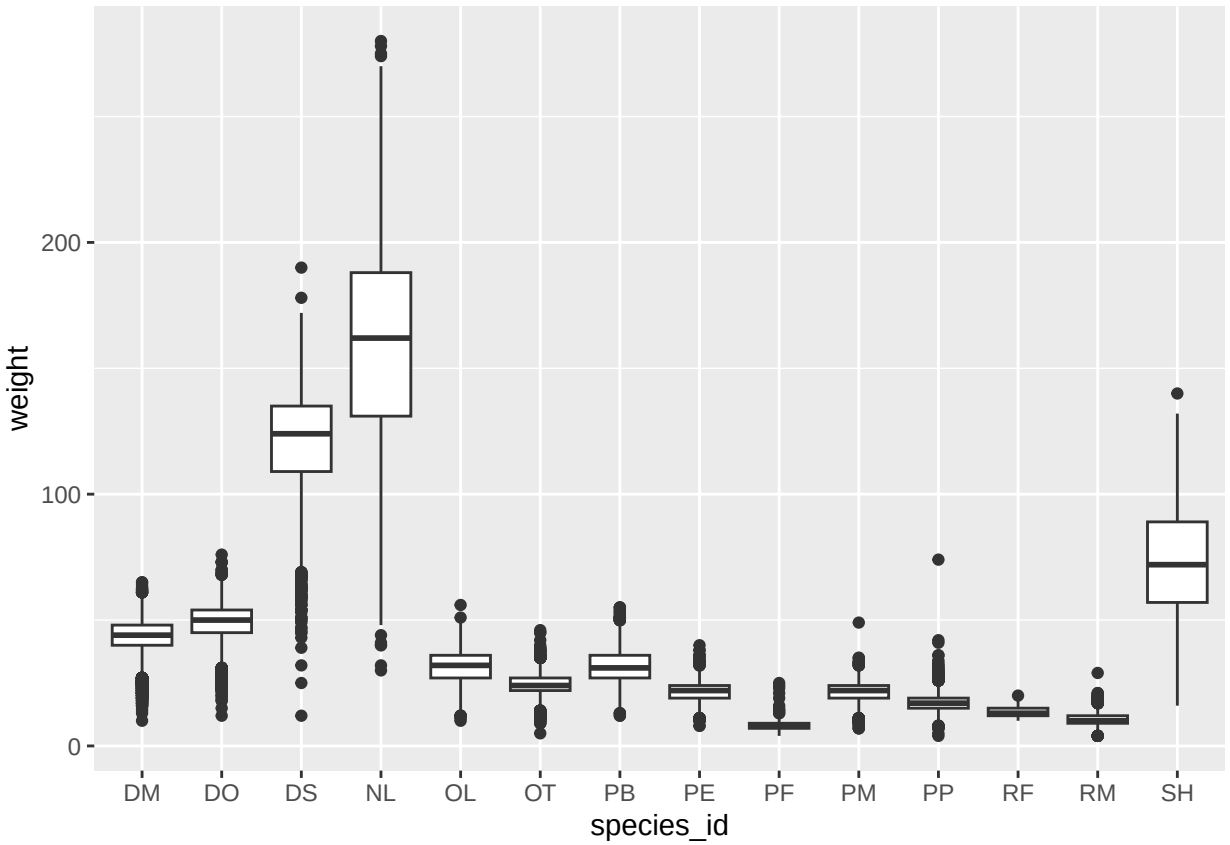
```
# Challenge
ggplot(data = surveys_complete, mapping = aes(x = species_id, y = weight)) +
  geom_point(aes(colour = plot_type))
```




## Boxplots

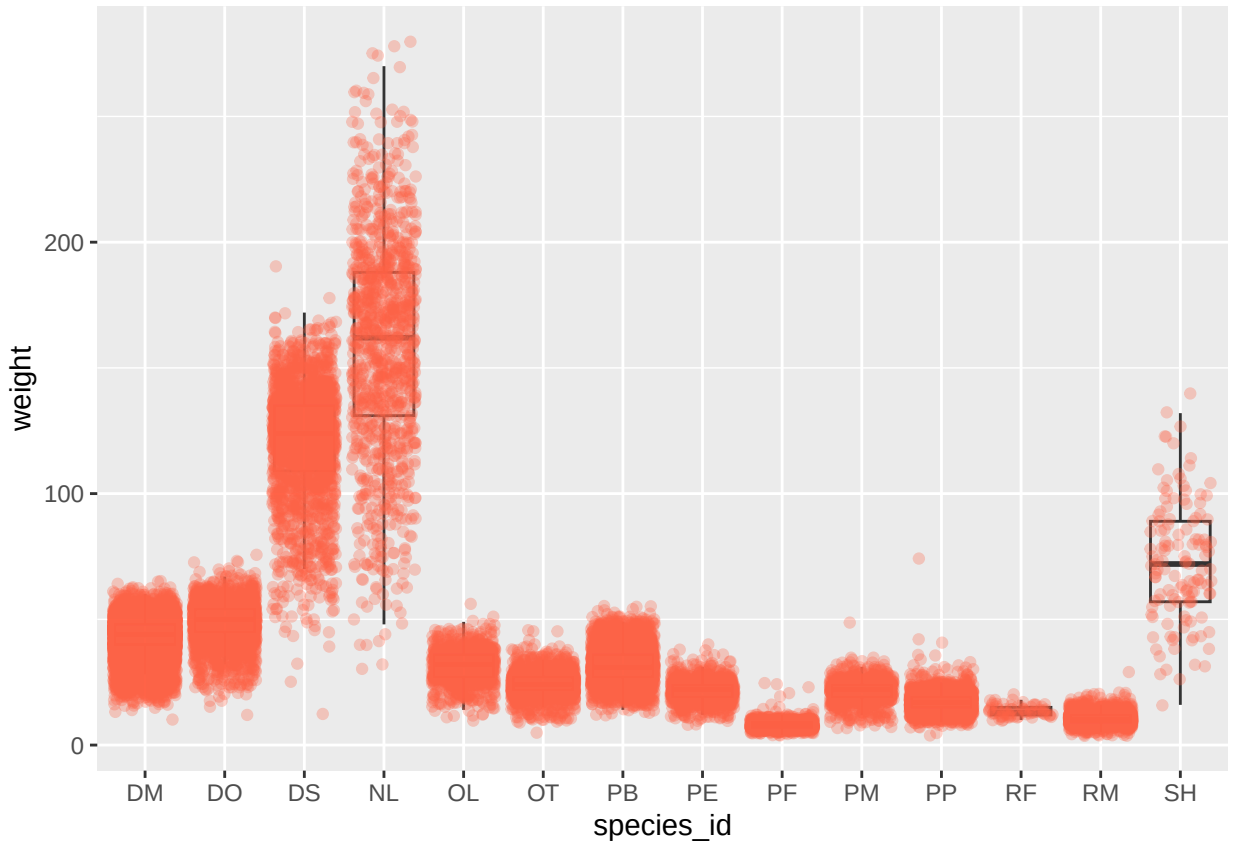
We can use boxplots to visualize the distribution of weight within each species:

```
# boxplot
ggplot(data = surveys_complete, mapping = aes(x = species_id, y = weight)) +
    geom_boxplot()
```



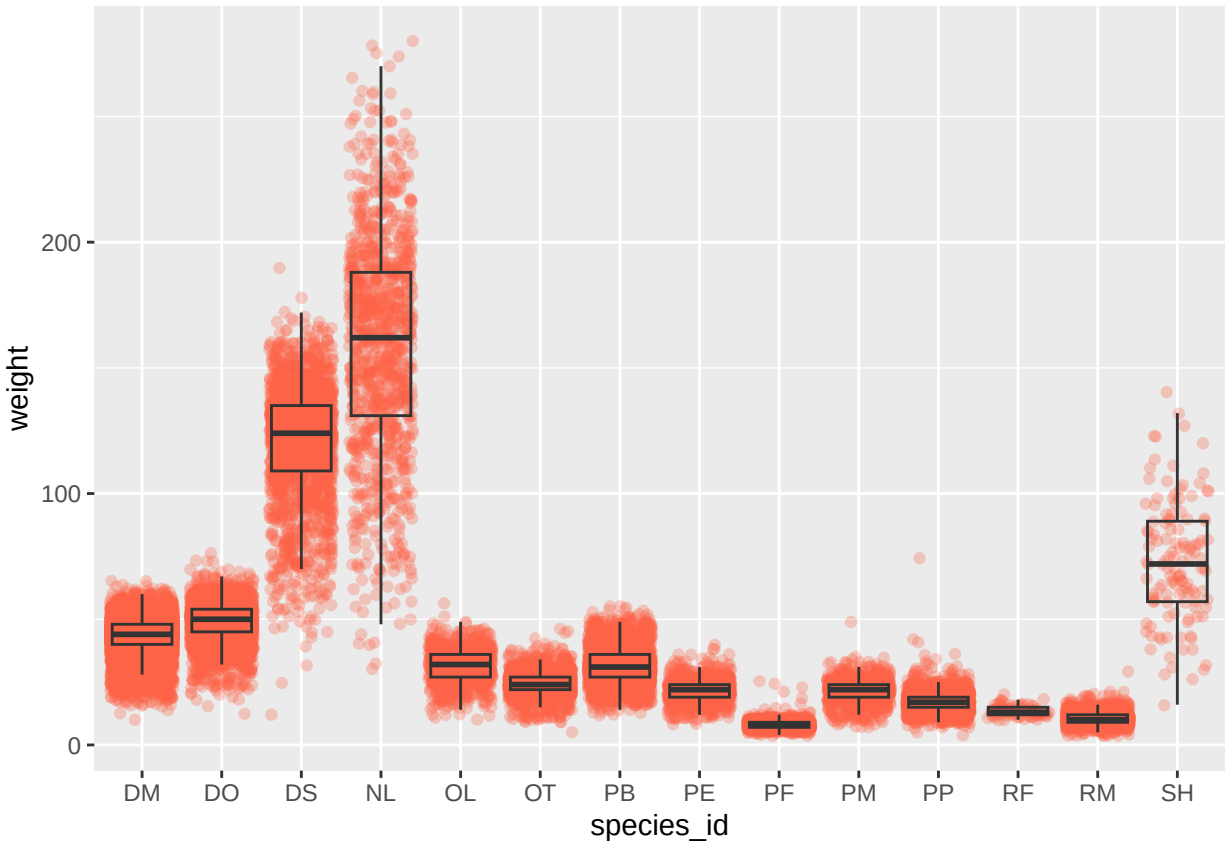

By adding points to the boxplot, we can have a better idea of the number of measurements and of their distribution:

```
# boxplot
ggplot(data = surveys_complete, mapping = aes(x = species_id, y = weight)) +
    geom_boxplot(alpha = 0) +
    geom_jitter(alpha = 0.3, colour = "tomato")
```




Notice how the boxplot layer is behind the jitter layer? What do you need to change in the code to put the boxplot in front of the points such that it's not hidden?

```
# boxplot
ggplot(data = surveys_complete, mapping = aes(x = species_id, y = weight)) +
  geom_jitter(alpha = 0.3, colour = "tomato") +
  geom_boxplot(alpha = 0)
```




**Group Challenges**

Boxplots are useful summaries, but hide the *shape* of the distribution. For example, if the distribution is bimodal, we would not see it in a boxplot. An alternative to the boxplot is the violin plot, where the shape (of the density of points) is drawn.

- Replace the box plot with a violin plot; see `geom_violin()`.

In many types of data, it is important to consider the *scale* of the observations. For example, it may be worth changing the scale of the axis to better distribute the observations in the space of the plot. Changing the scale of the axes is done similarly to adding/modifying other components (i.e., by incrementally adding commands). Try making these modifications:

- Represent weight on the $\log_{10}$ scale; see `scale_y_log10()`.

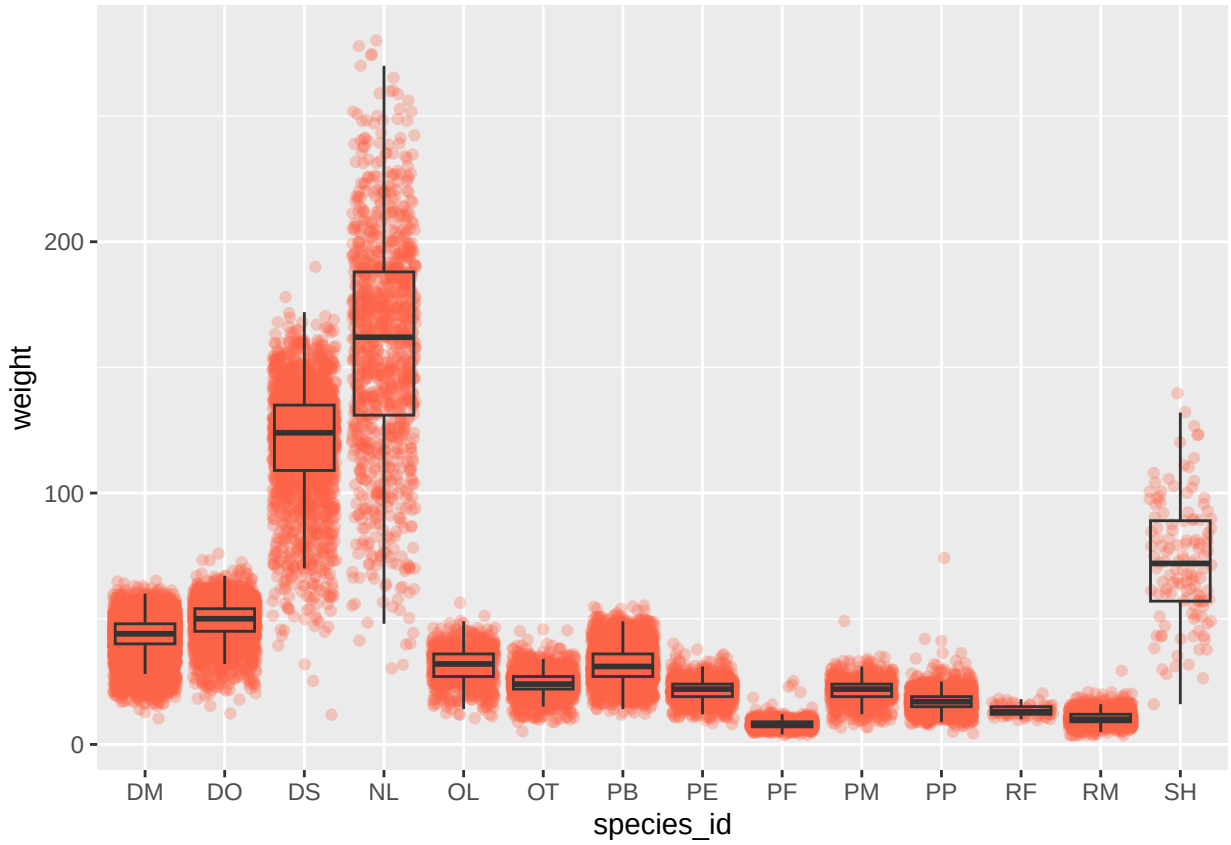
So far, we've looked at the distribution of weight within species. Try making a new plot to explore the distribution of another variable within each species.

- Create a boxplot for `hindfoot_length`. Overlay the boxplot layer on a jitter layer to show actual measurements.
- Add colour to the data points on your boxplot according to the plot from which the sample was taken (`plot_id`).

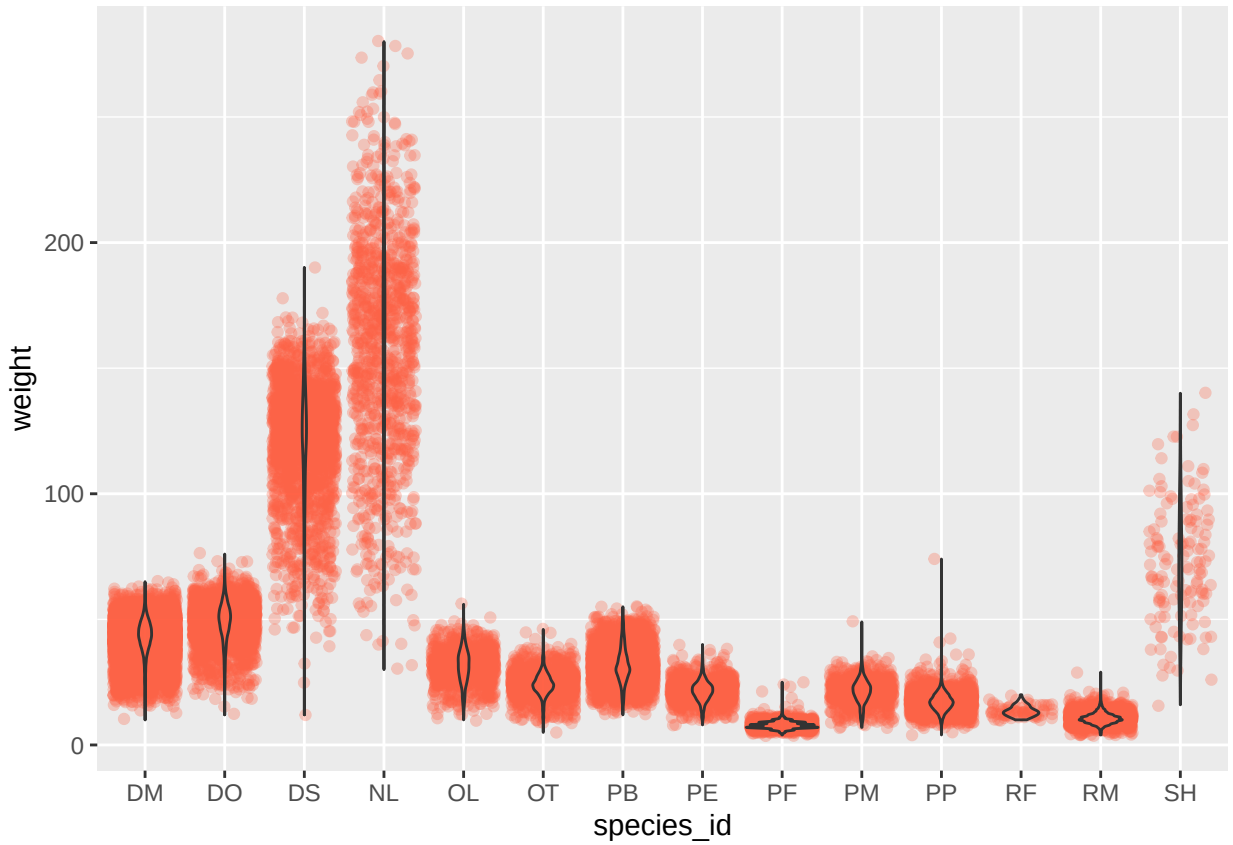
*Hint:* Check the class for `plot_id`. Consider changing the class of `plot_id` from integer to factor. Why does this change how R makes the graph?

```
## Challenge with boxplots:
##  Start with the boxplot we created:
ggplot(data = surveys_complete, mapping = aes(x = species_id, y = weight)) +
```

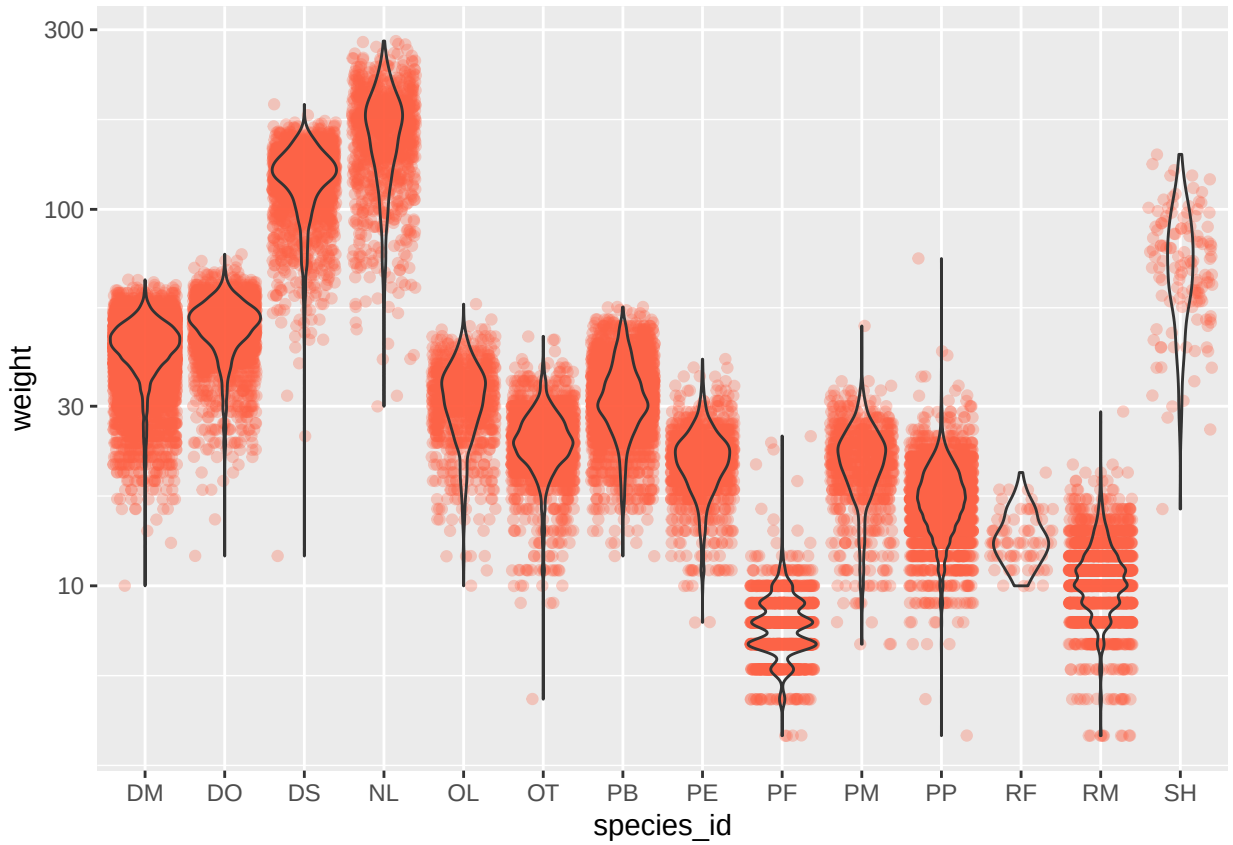
```
  geom_jitter(alpha = 0.3, colour = "tomato") +
  geom_boxplot(alpha = 0)
```



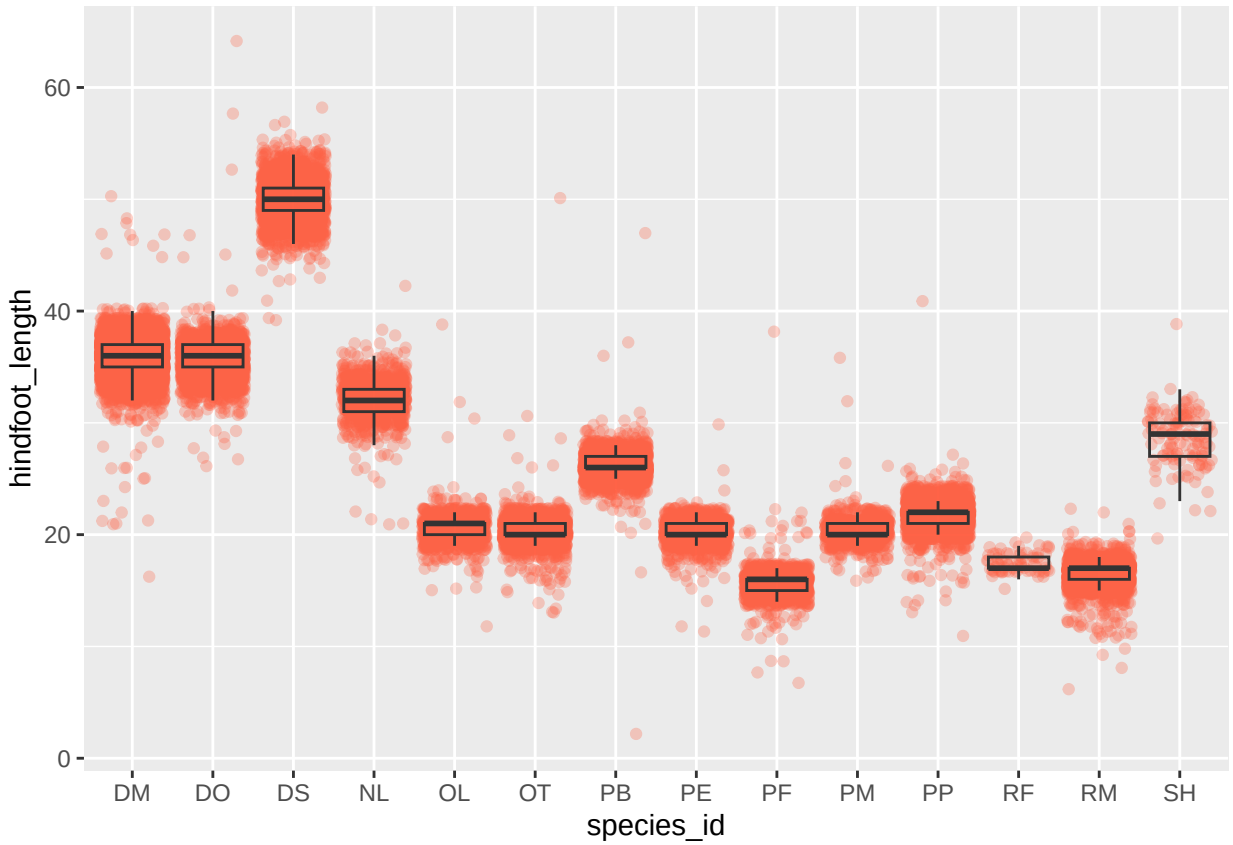

```
## 1. Replace the box plot with a violin plot; see `geom_violin()`.
ggplot(data = surveys_complete, mapping = aes(x = species_id, y = weight)) +
  geom_jitter(alpha = 0.3, colour = "tomato") +
  geom_violin(alpha = 0)
```



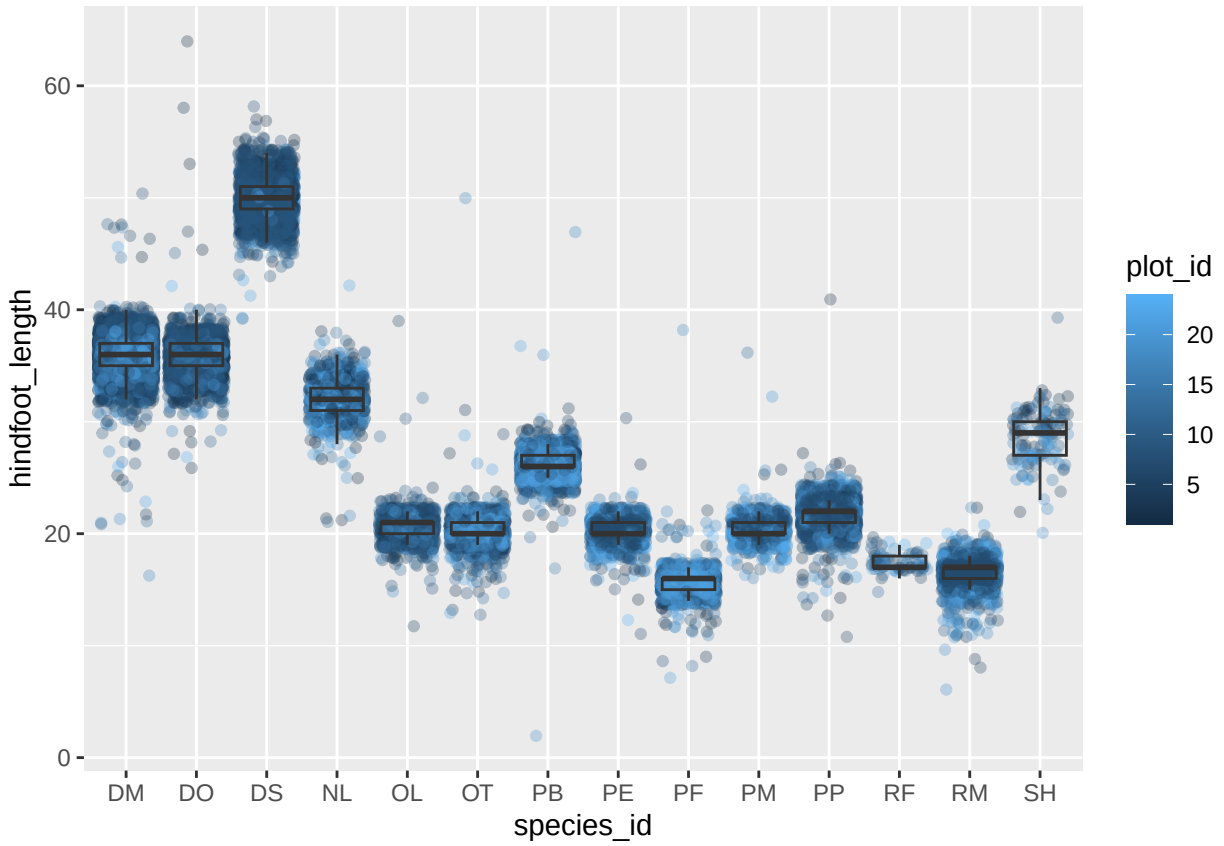

```
##  2. Represent weight on the log10 scale; see `scale_y_log10()`.
ggplot(data = surveys_complete, mapping = aes(x = species_id, y = weight)) +
  geom_jitter(alpha = 0.3, colour = "tomato") +
  geom_violin(alpha = 0) +
  scale_y_log10()
```




```
##  3. Create boxplot for `hindfoot_length` overlaid on a jitter layer.
ggplot(data = surveys_complete, mapping = aes(x = species_id, y = hindfoot_length)) +
  geom_jitter(alpha = 0.3, colour = "tomato") +
  geom_boxplot(alpha = 0)
```



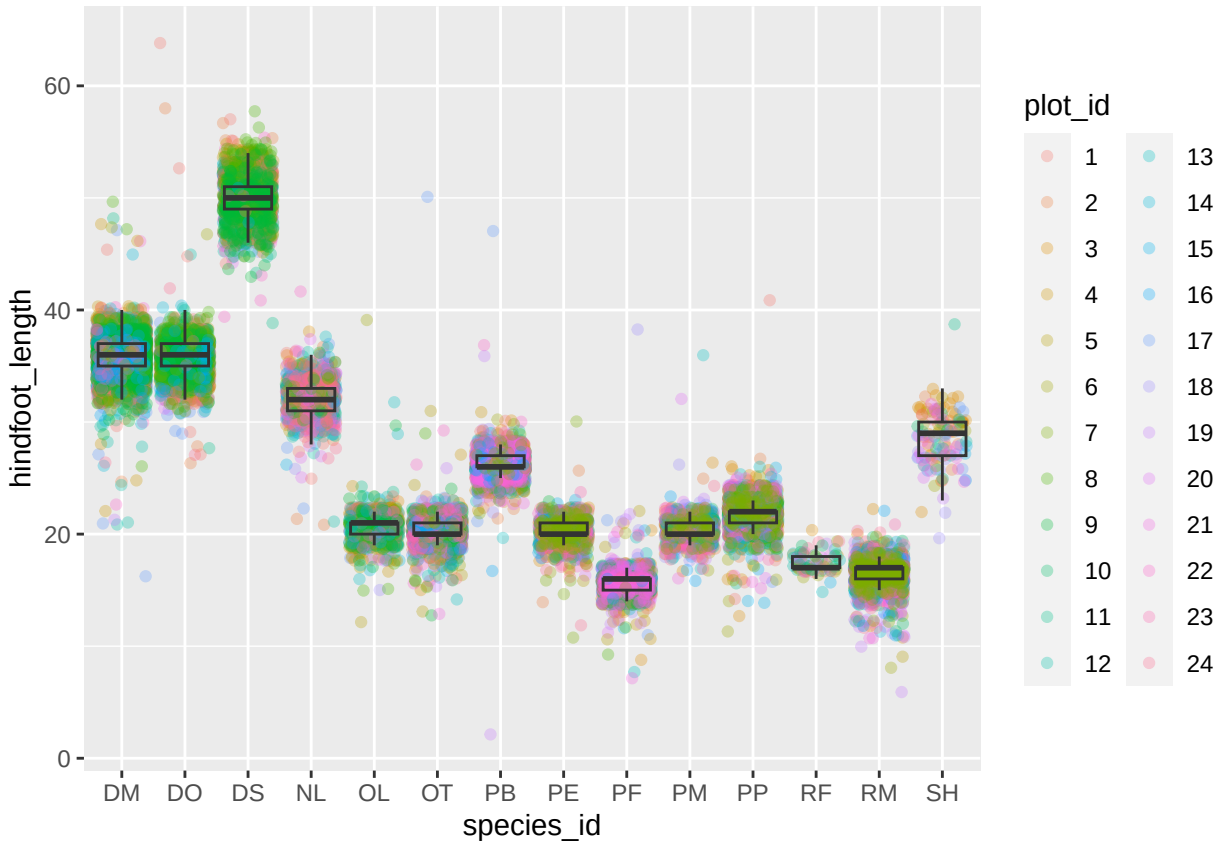

```
##  4. Add colour to the data points on your boxplot according to the
##  plot from which the sample was taken (`plot_id`).
ggplot(data = surveys_complete, mapping = aes(x = species_id, y = hindfoot_length)) +
  geom_jitter(alpha = 0.3, aes(colour = plot_id)) +
  geom_boxplot(alpha = 0)
```




```
##  *Hint:* Check the class for `plot_id`.
class(surveys_complete$plot_id)
```

```
## [1] "numeric"
```

```
## Consider changing the class of `plot_id` from integer to factor. Why does this change how R makes th
surveys_complete$plot_id <- as.factor(surveys_complete$plot_id)

ggplot(data = surveys_complete, mapping = aes(x = species_id, y = hindfoot_length)) +
  geom_jitter(alpha = 0.3, aes(colour = plot_id)) +
  geom_boxplot(alpha = 0)
```



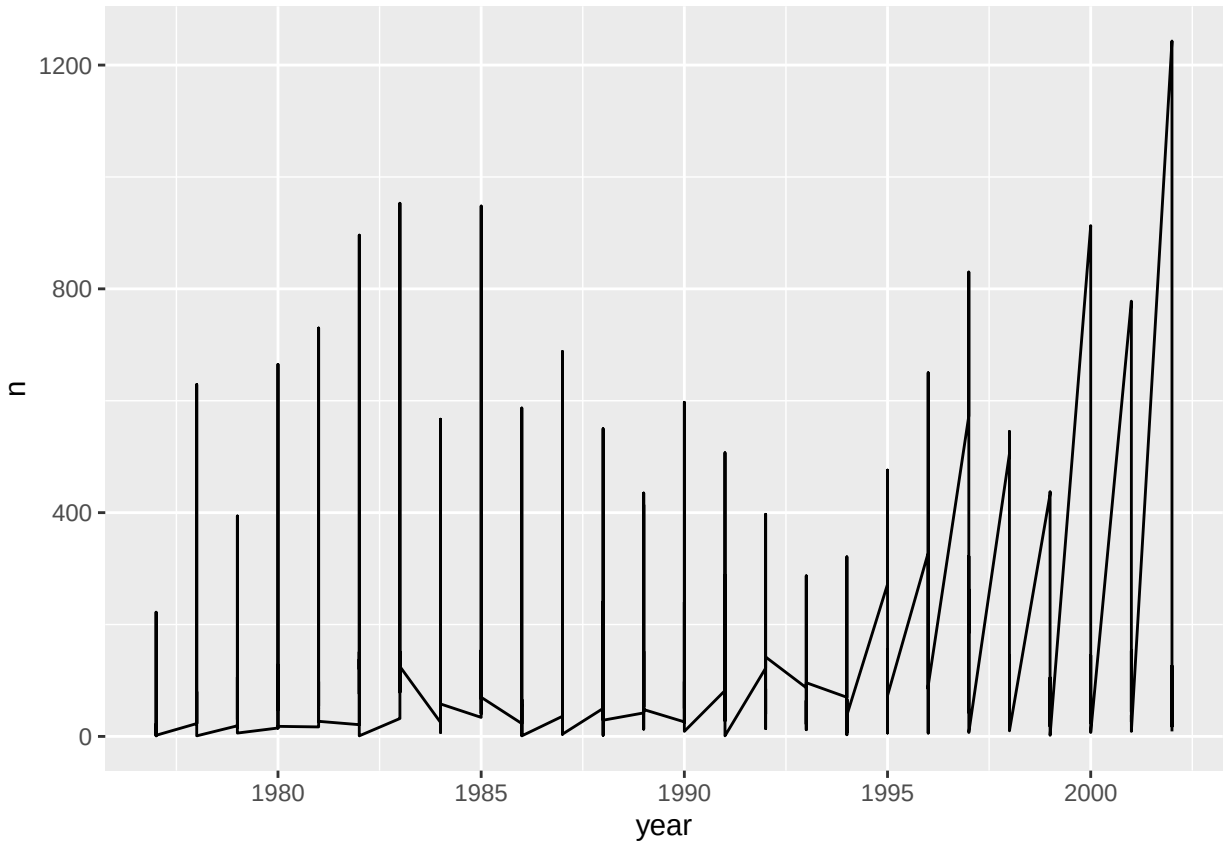

## Plotting time series data

Let's calculate number of counts per year for each genus. First we need to group the data and count records within each group:

```
# number of counts
yearly_counts <- surveys_complete %>%
  count(year, genus)
```

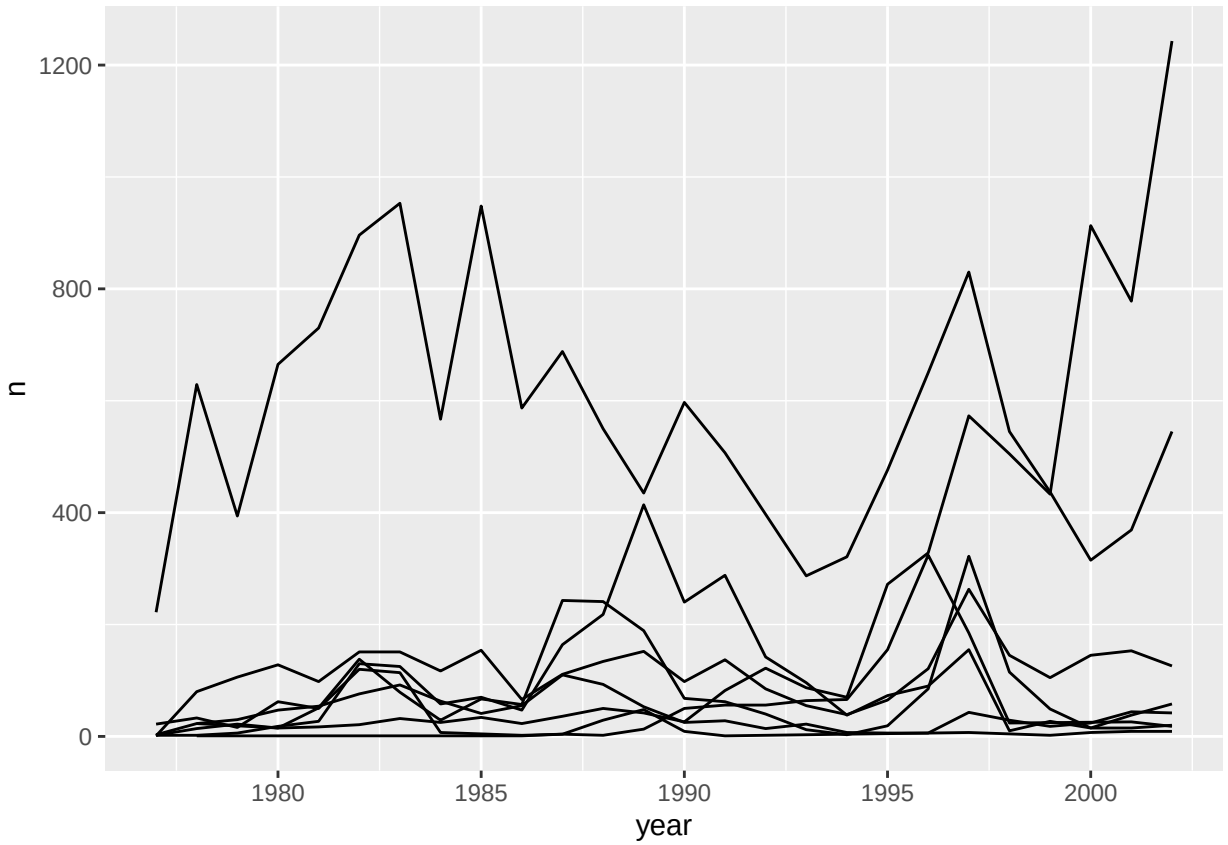
Time series data can be visualized as a line plot with years on the x axis and counts on the y axis:

```
# time series
ggplot(data = yearly_counts, mapping = aes(x = year, y = n)) +
     geom_line()
```



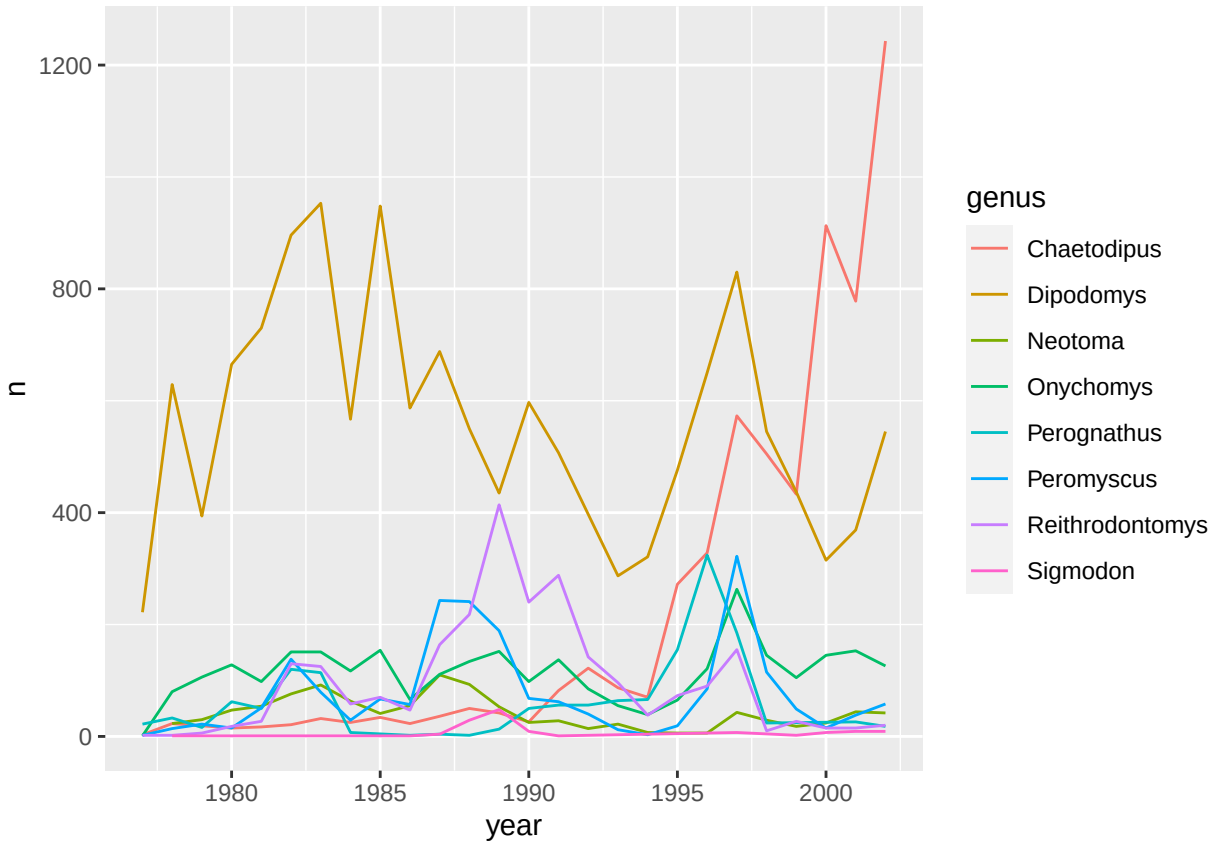

Unfortunately, this does not work because we plotted data for all the genera together. We need to tell ggplot to draw a line for each genus by modifying the aesthetic function to include `group = genus`:

```
# time series
ggplot(data = yearly_counts, mapping = aes(x = year, y = n, group = genus)) +
    geom_line()
```




We will be able to distinguish genera in the plot if we add colours (using `colour` also automatically groups the data):

```
# add colour
ggplot(data = yearly_counts, mapping = aes(x = year, y = n, colour = genus)) +
    geom_line()
```




## Faceting

`ggplot2` has a special technique called *faceting* that allows the user to split one plot into multiple plots based on a factor included in the dataset.

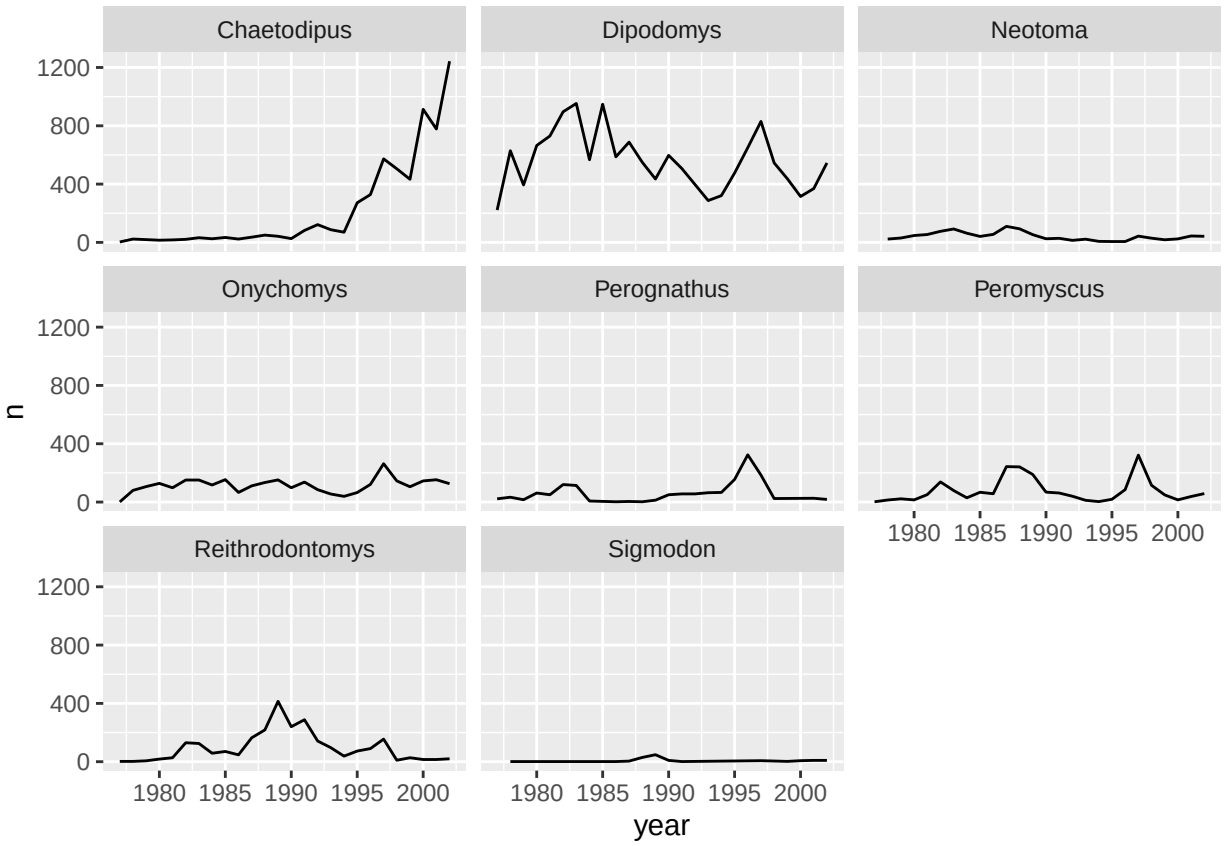
There are two types of `facet` functions:

- `facet_wrap()` arranges a one-dimensional sequence of panels to allow them to cleanly fit on one page.
- `facet_grid()` allows you to form a matrix of rows and columns of panels.

Both geometries allow to to specify faceting variables specified within `vars()`. For example, `facet_wrap(facets = vars(facet_variable))` or `facet_grid(rows = vars(row_variable), cols = vars(col_variable))`.

Let's start by using `facet_wrap()` to make a time series plot for each species:

```
# facet
ggplot(data = yearly_counts, mapping = aes(x = year, y = n)) +
    geom_line() +
    facet_wrap(facets = vars(genus))
```

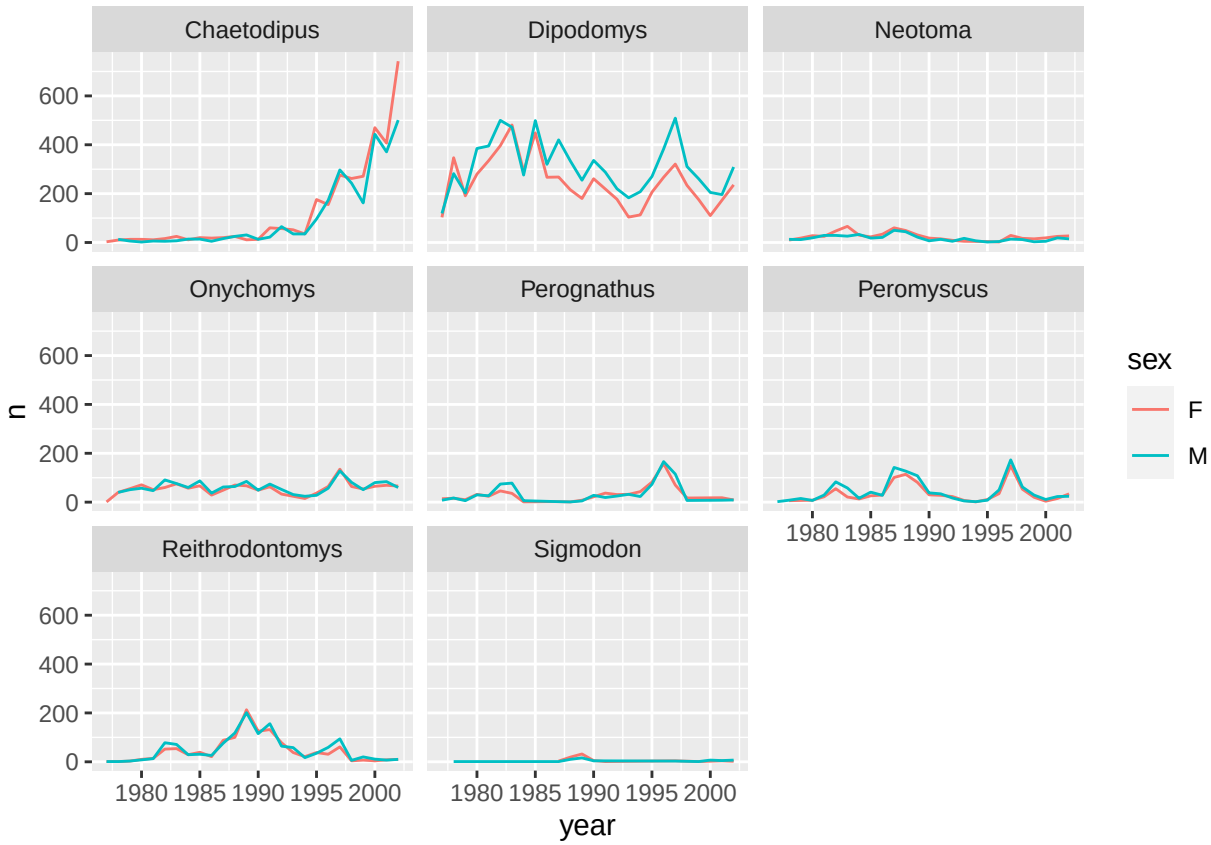



Now we would like to split the line in each plot by the sex of each individual measured. To do that we need to make counts in the data frame grouped by `year`, `genus`, and `sex`:

```
# split by sex
yearly_sex_counts <- surveys_complete %>%
  count(year, genus, sex)
```

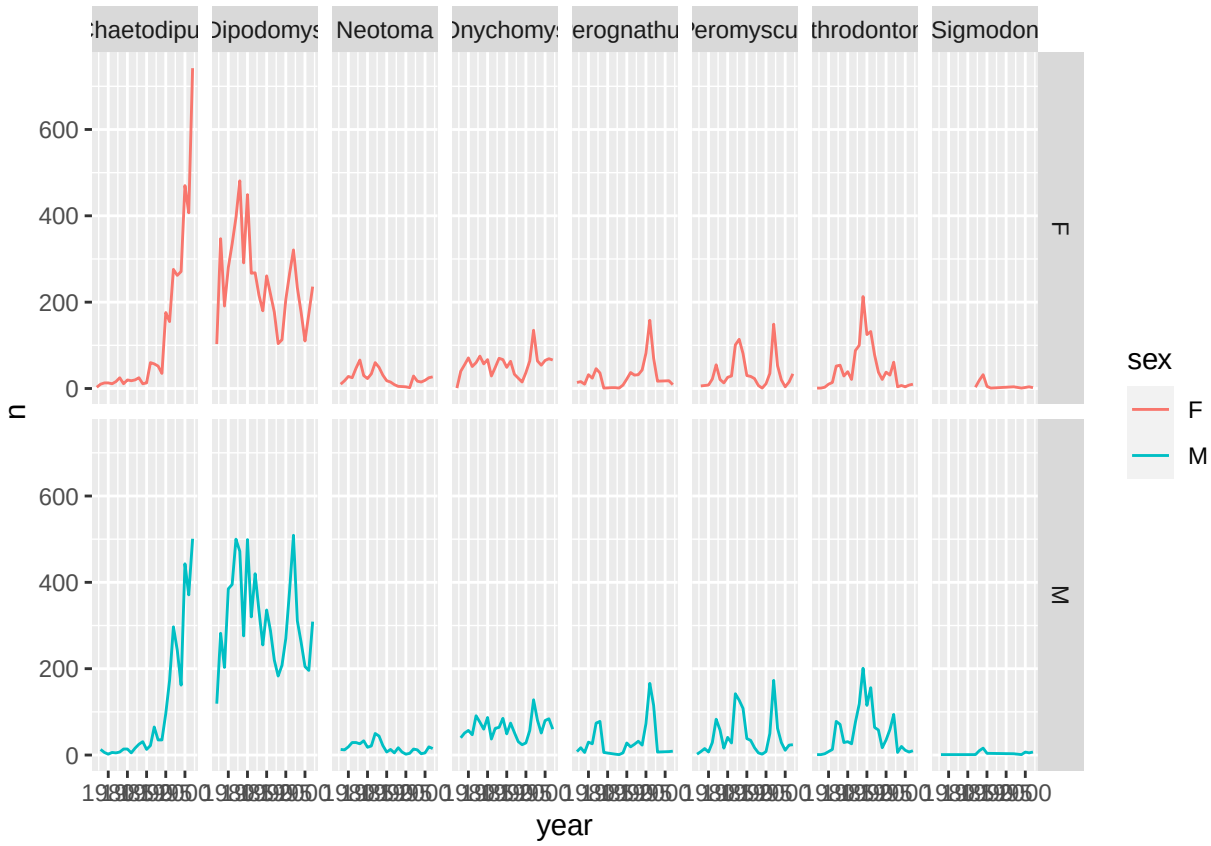
We can now make the faceted plot by splitting further by sex using `colour` (within each panel):

```
# add colour
ggplot(data = yearly_sex_counts, mapping = aes(x = year, y = n, colour = sex)) +
  geom_line() +
  facet_wrap(facets =  vars(genus))
```



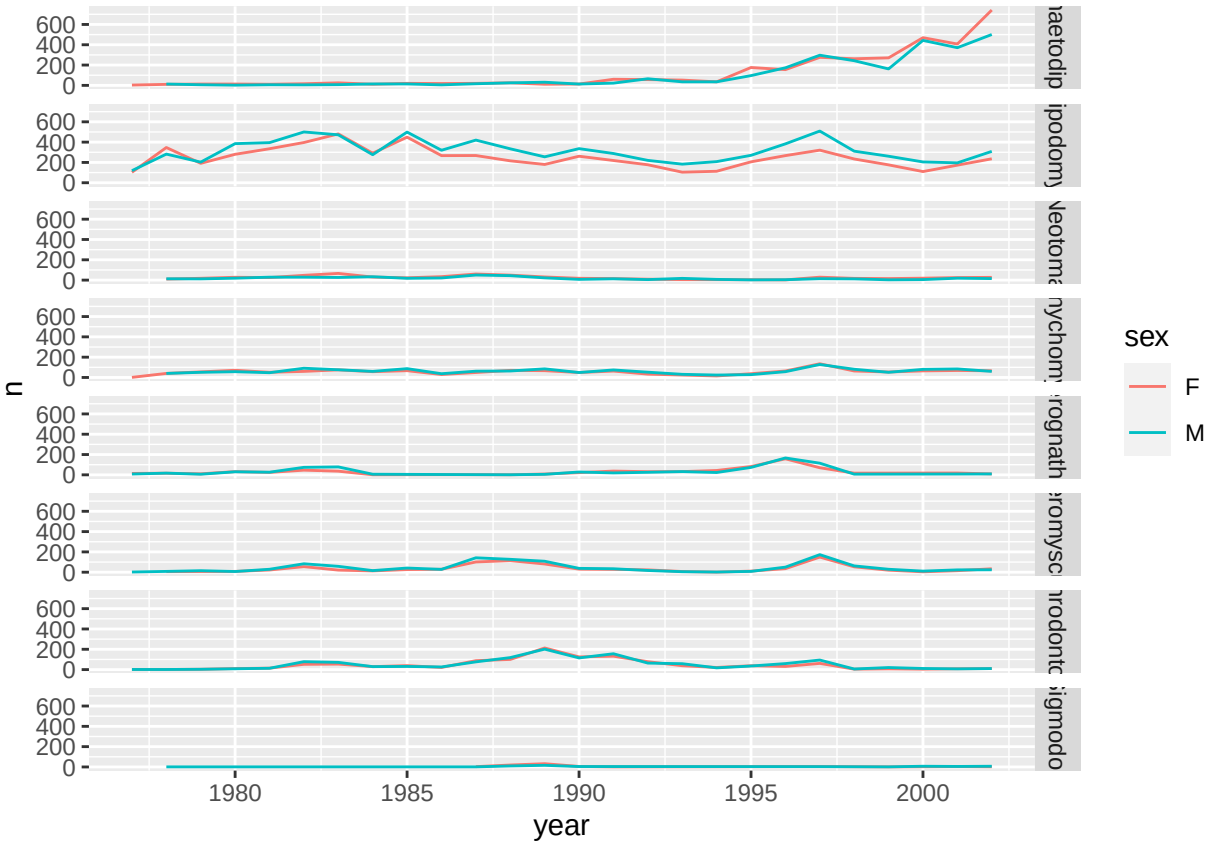

Now let's use `facet_grid()` to control how panels are organised by both rows and columns:

```
# facet_grid
ggplot(data = yearly_sex_counts,
       mapping = aes(x = year, y = n, colour = sex)) +
  geom_line() +
  facet_grid(rows = vars(sex), cols =  vars(genus))
```




You can also organise the panels only by rows (or only by columns):

```
# One column, facet by rows
ggplot(data = yearly_sex_counts,
       mapping = aes(x = year, y = n, colour = sex)) +
  geom_line() +
  facet_grid(rows = vars(genus))
```




**Note:** In earlier versions of `ggplot2` you need to use an interface using formulas to specify how plots are faceted (and this is still supported in new versions). The equivalent syntax is:

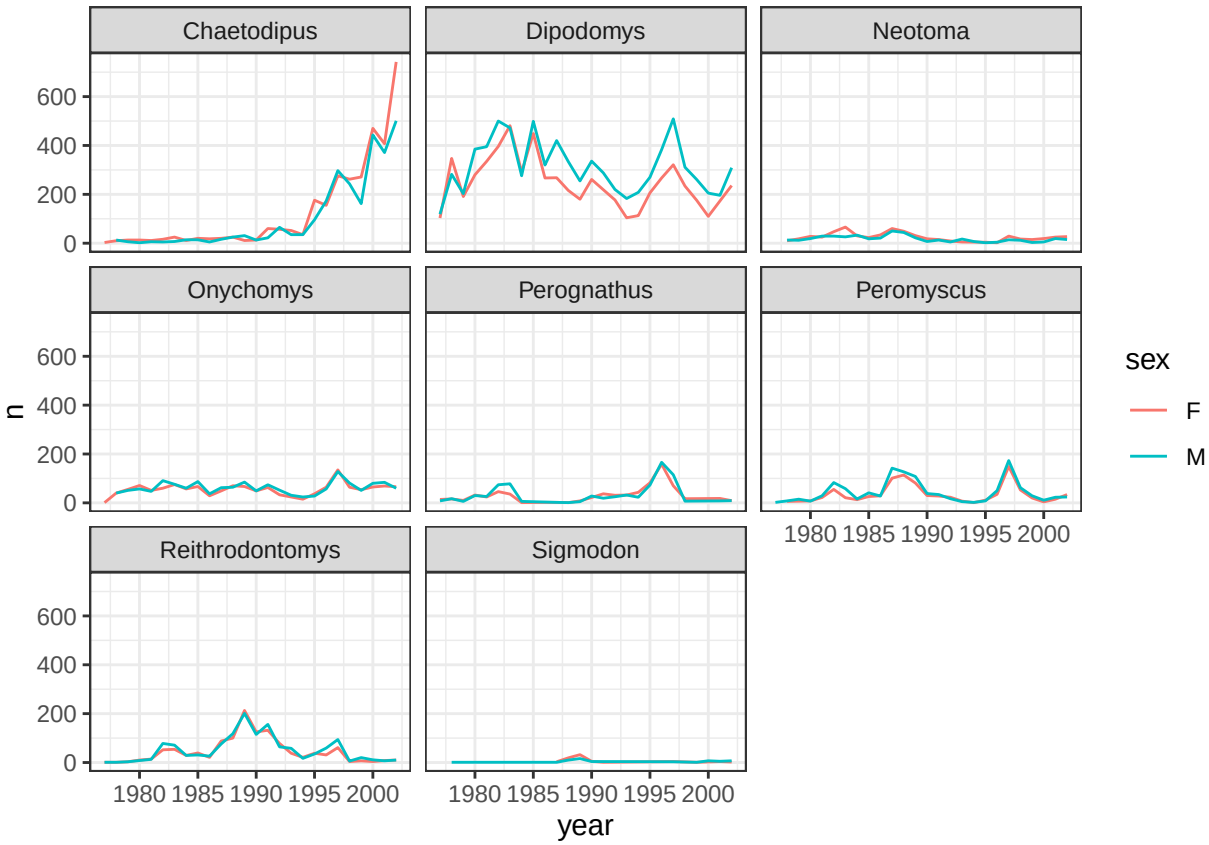
```
# facet wrap
facet_wrap(vars(genus))     # new
facet_wrap(~ genus)         # old

# grid on both rows and columns
facet_grid(rows = vars(genus), cols = vars(sex))    # new
facet_grid(genus ~ sex)                             # old

# grid on rows only
facet_grid(rows = vars(genus))    # new
facet_grid(genus ~ .)             # old

# grid on columns only
facet_grid(cols = vars(genus))    # new
facet_grid(. ~ genus)             # old
```

## `ggplot2` themes

Usually plots with white background look more readable when printed. Every single component of a `ggplot` graph can be customised using the generic `theme()` function, as we will see below. However, there are pre-loaded themes available that change the overall appearance of the graph without much effort.

For example, we can change our previous graph to have a simpler white background using the `theme_bw()` function:

```
# facet by species
 ggplot(data = yearly_sex_counts,
        mapping = aes(x = year, y = n, colour = sex)) +
     geom_line() +
     facet_wrap(vars(genus)) +
     theme_bw()
```




In addition to `theme_bw()`, which changes the plot background to white, **ggplot2** comes with several other themes which can be useful to quickly change the look of your visualization. The complete list of themes is available at https://ggplot2.tidyverse.org/reference/ggtheme.html. `theme_minimal()` and `theme_light()` are popular, and `theme_void()` can be useful as a starting point to create a new hand-crafted theme.

The ggthemes package provides a wide variety of options. The **ggplot2** extensions website provides a list of packages that extend the capabilities of **ggplot2**, including additional themes.

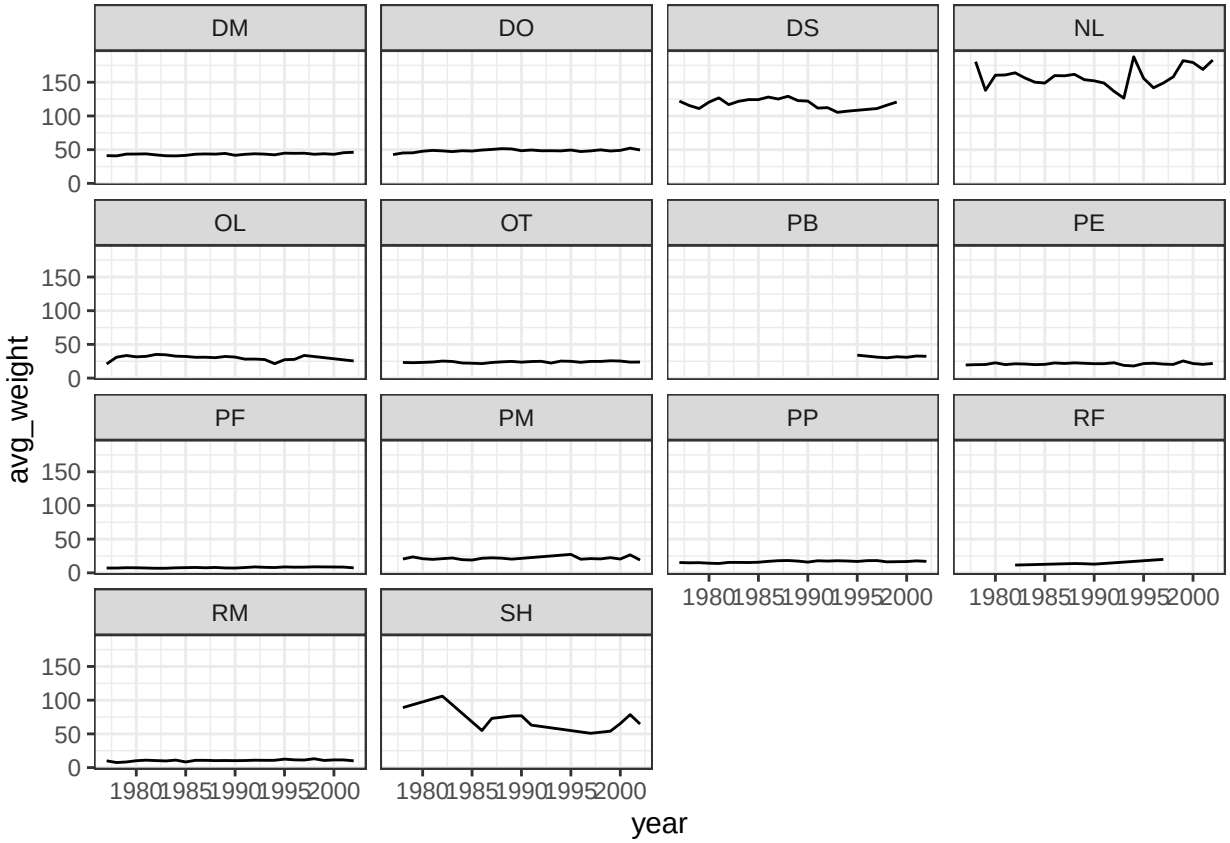
### Group Challenge

Use what you just learned to create a plot that depicts how the average weight of each species changes through the years.

```
# Challenge
yearly_weight <- surveys_complete %>%
  group_by(year, species_id) %>%
  summarize(avg_weight = mean(weight))
```

```
## `summarise()` has grouped output by 'year'. You can override using the
## `.groups` argument.
```

```
ggplot(data = yearly_weight, mapping = aes(x=year, y=avg_weight)) +
  geom_line() +
  facet_wrap(vars(species_id)) +
  theme_bw()
```

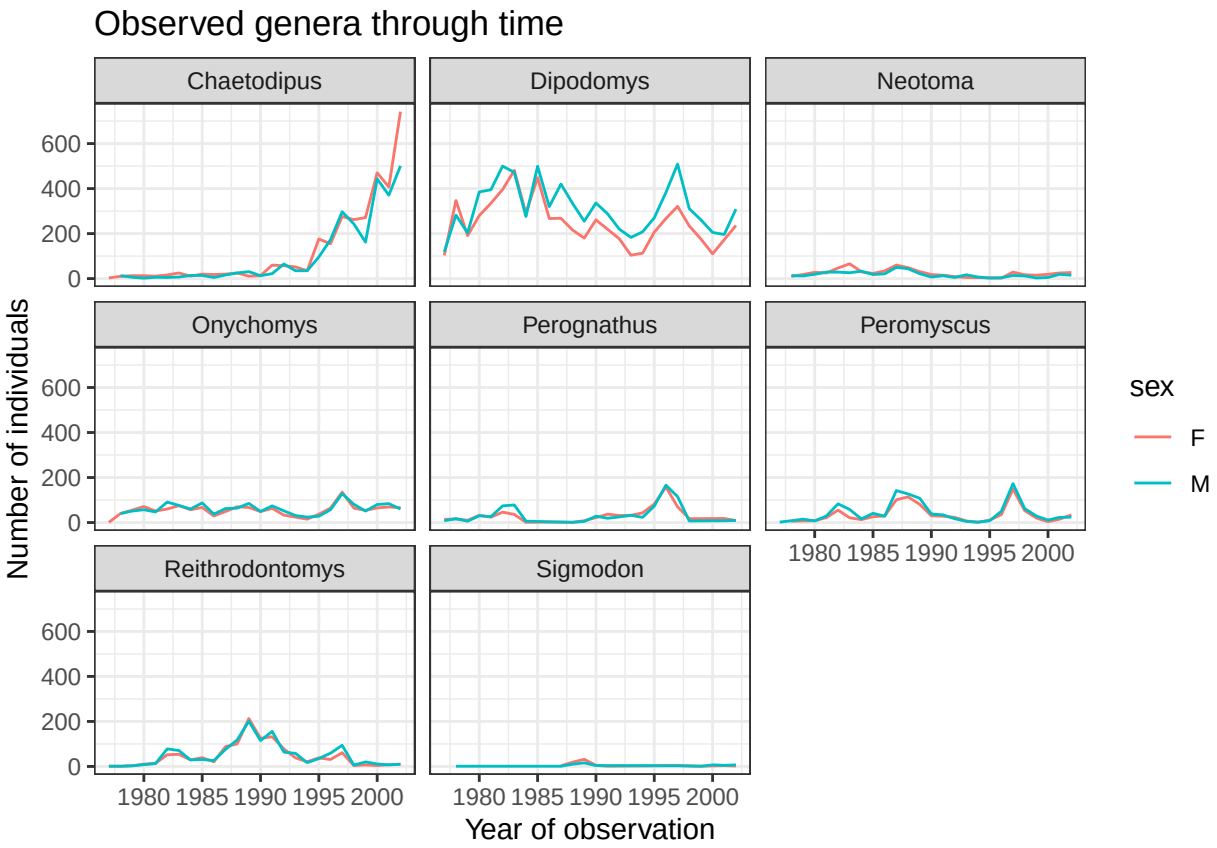



## customisation

Take a look at the **ggplot2** cheat sheet, and think of ways you could improve the plot.

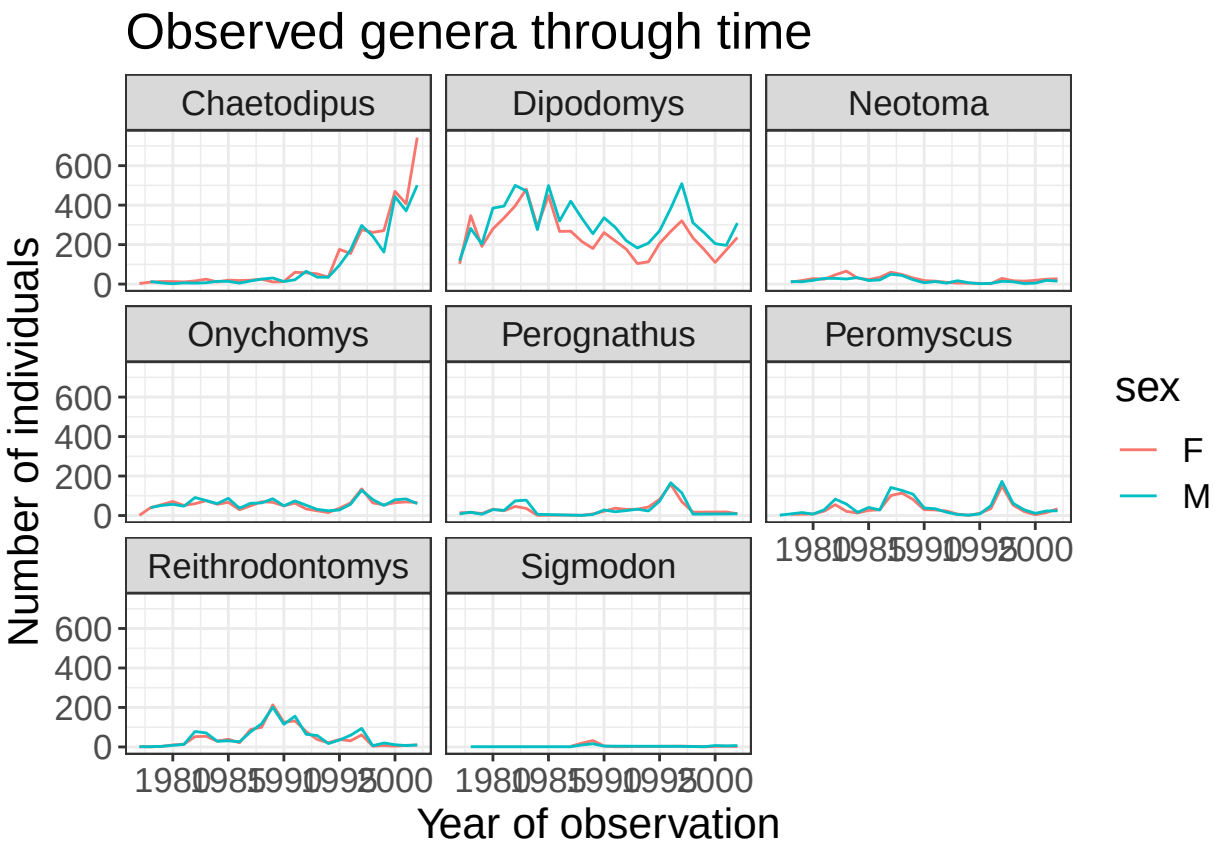
Now, let's change names of axes to something more informative than 'year' and 'n' and add a title to the figure:

```
# Customise
ggplot(data = yearly_sex_counts, mapping = aes(x = year, y = n, colour = sex)) +
    geom_line() +
    facet_wrap(vars(genus)) +
    labs(title = "Observed genera through time",
         x = "Year of observation",
         y = "Number of individuals") +
    theme_bw()
```




The axes have more informative names, but their readability can be improved by increasing the font size. This can be done with the generic `theme()` function:

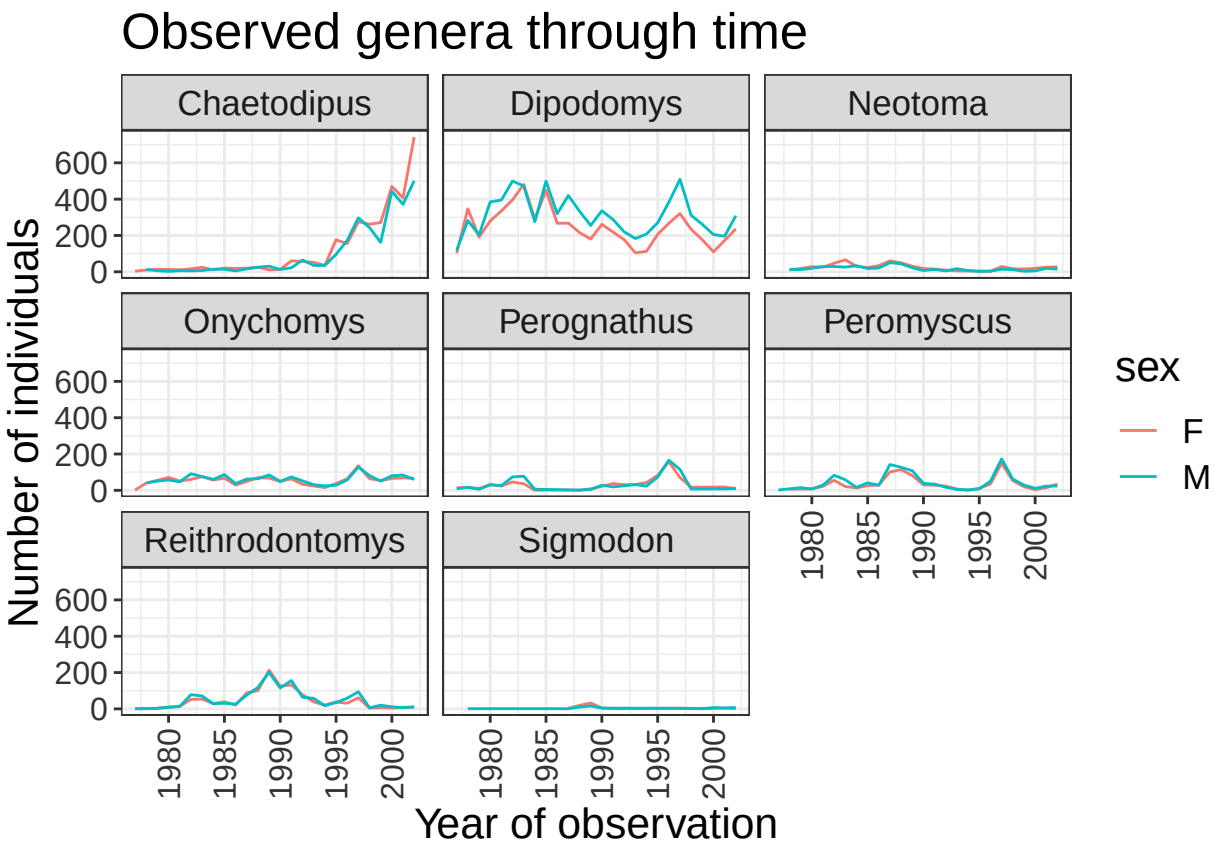
```
# theme
ggplot(data = yearly_sex_counts, mapping = aes(x = year, y = n, colour = sex)) +
    geom_line() +
    facet_wrap(vars(genus)) +
    labs(title = "Observed genera through time",
        x = "Year of observation",
        y = "Number of individuals") +
    theme_bw() +
    theme(text=element_text(size = 16))
```




Note that it is also possible to change the fonts of your plots. If you are on Windows, you may have to install the **extrafont** package, and follow the instructions included in the README for this package.

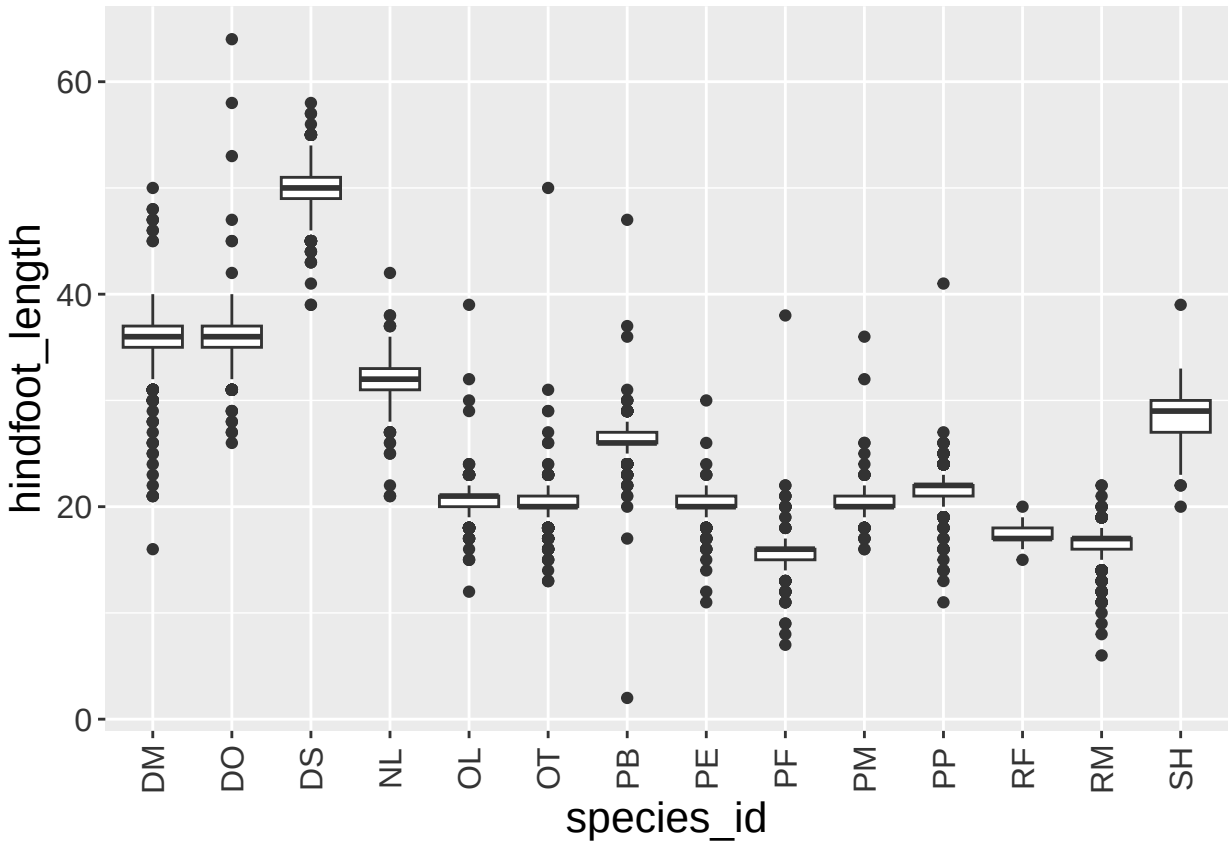
After our manipulations, you may notice that the values on the x-axis are still not properly readable. Let's change the orientation of the labels and adjust them vertically and horizontally so they don't overlap. You can use a 90-degree angle, or experiment to find the appropriate angle for diagonally oriented labels:

```
ggplot(data = yearly_sex_counts, mapping = aes(x = year, y = n, colour = sex)) +
    geom_line() +
    facet_wrap(vars(genus)) +
    labs(title = "Observed genera through time",
        x = "Year of observation",
        y = "Number of individuals") +
    theme_bw() +
    theme(axis.text.x = element_text(colour = "grey20", size = 12, angle = 90, hjust = 0.5, vjust = 0.5)
                      axis.text.y = element_text(colour = "grey20", size = 12),
          text = element_text(size = 16))
```




If you like the changes you created better than the default theme, you can save them as an object to be able to easily apply them to other plots you may create:

```
# define custom theme
grey_theme <- theme(axis.text.x = element_text(colour = "grey20", size = 12, angle = 90, hjust = 0.5, v
                    axis.text.y = element_text(colour = "grey20", size = 12),
                    text = element_text(size = 16))

# create a boxplot with the new theme
ggplot(surveys_complete, aes(x = species_id, y = hindfoot_length)) +
    geom_boxplot() +
    grey_theme
```




**Individual Challenge**

With all of this information in hand, please take another five minutes to either improve one of the plots generated in this exercise or create a beautiful graph of your own. Use the RStudio **ggplot2** cheat sheet for inspiration. Here are some ideas: * See if you can change the thickness of the lines. * Can you find a way to change the name of the legend? What about its labels? * Try using a different colour palette (seehttp://www.cookbook-r.com/Graphs/colours_(ggplot2)/).

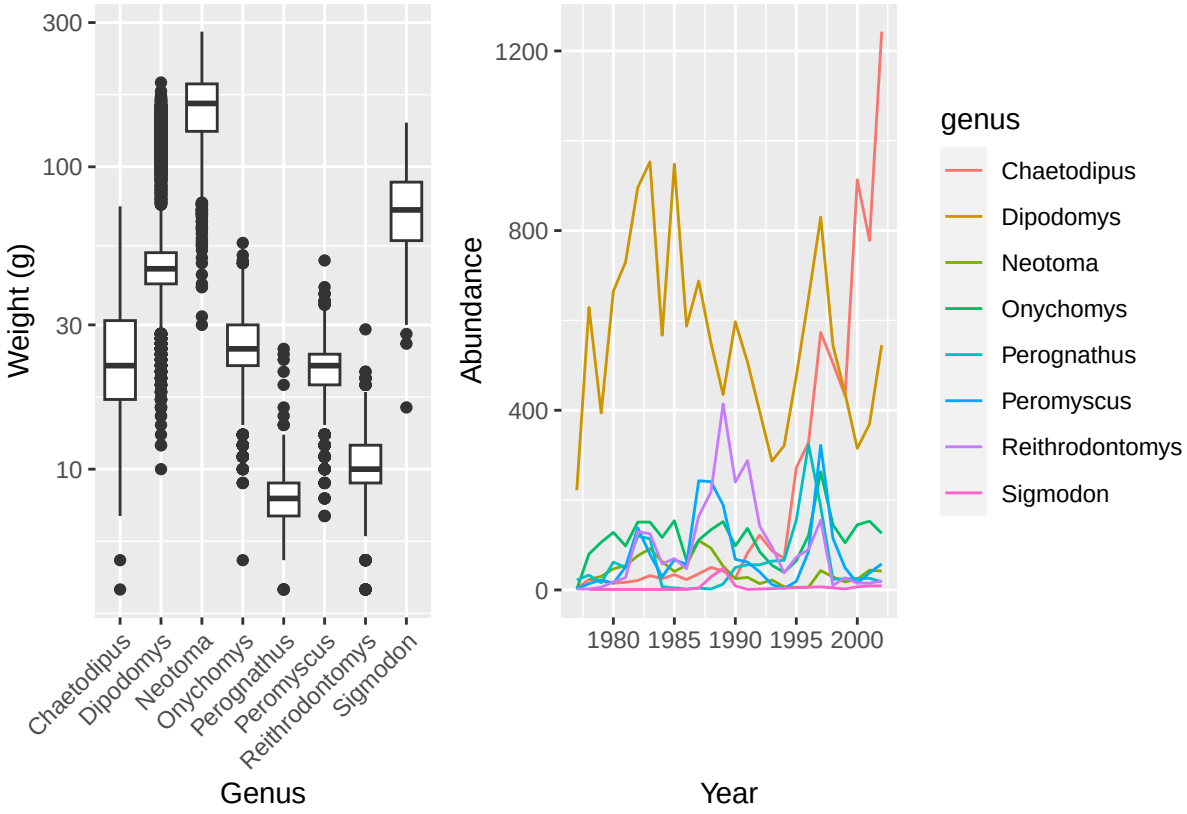
## Arranging and exporting plots

Faceting is a great tool for splitting one plot into multiple plots, but sometimes you may want to produce a single figure that contains multiple plots using different variables or even different data frames. The **patchwork** package allows us to combine separate ggplots into a single figure using arithmetic operators (+, /):

```
#install.packages("patchwork")
```

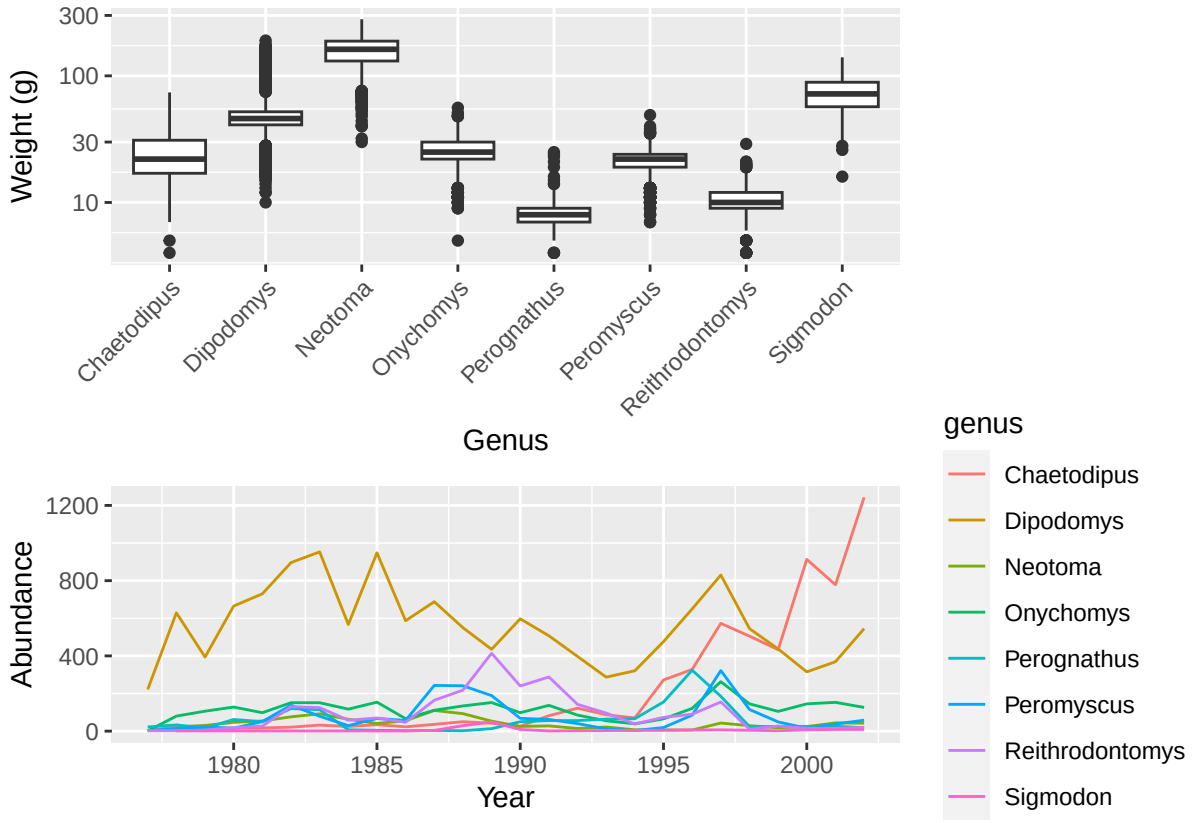
```
library(patchwork)

spp_weight_boxplot <- ggplot(data = surveys_complete,
                             mapping = aes(x = genus, y = weight)) +
  geom_boxplot() +
  scale_y_log10() +
  labs(x = "Genus", y = "Weight (g)") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))

spp_count_plot <- ggplot(data = yearly_counts,
                         mapping = aes(x = year, y = n, colour = genus)) +
```

```
  geom_line() +
  labs(x = "Year", y = "Abundance")

spp_weight_boxplot + spp_count_plot
```




Plots can also be stacked using `/`:

```
spp_weight_boxplot / spp_count_plot
```



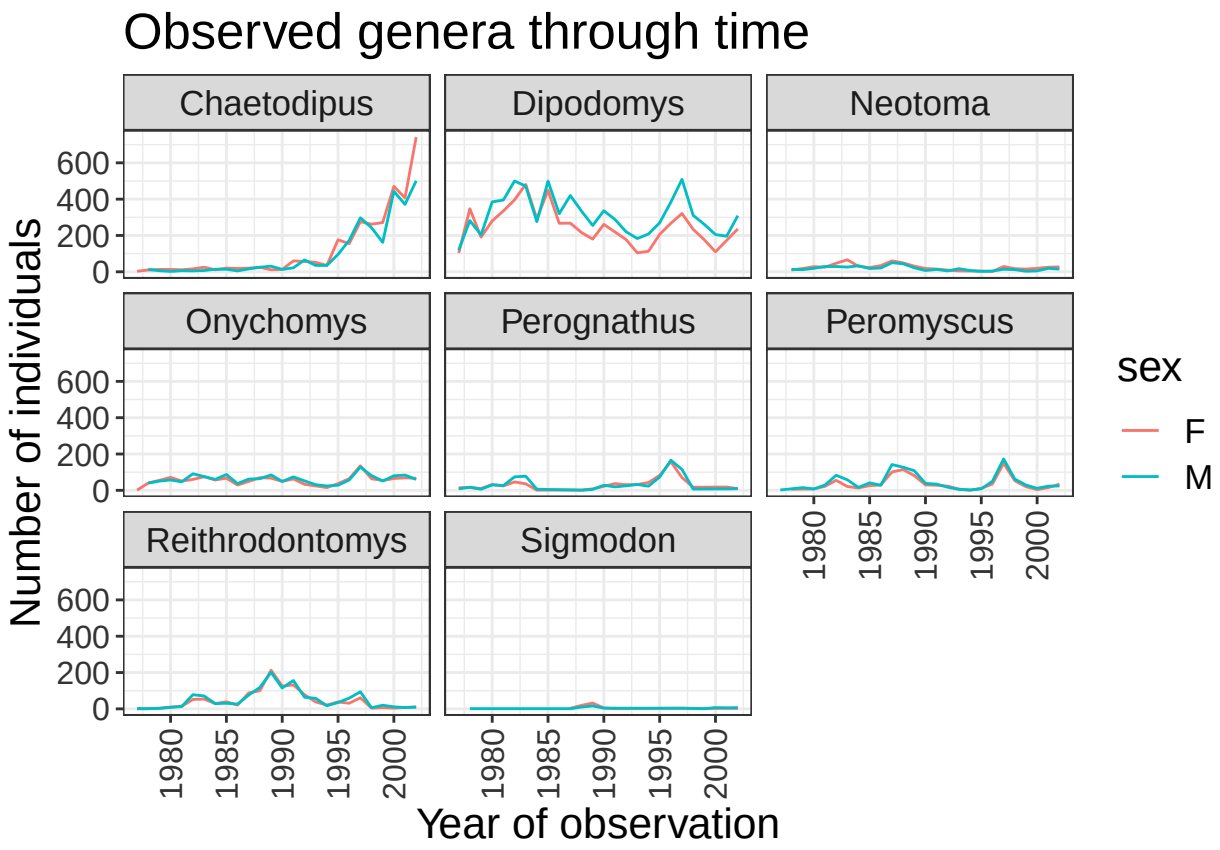

In addition to adding and stacking plots, there are tools for constructing more complex layouts.

After creating your plot, you can save it to a file in your favorite format. The Export tab in the **Plot** pane in RStudio will save your plots at low resolution, which will not be accepted by many journals and will not scale well for posters.

Instead, use the `ggsave()` function, which allows you easily change the dimension and resolution of your plot by adjusting the appropriate arguments (`width`, `height` and `dpi`).

Make sure you have the `fig/` folder in your working directory.

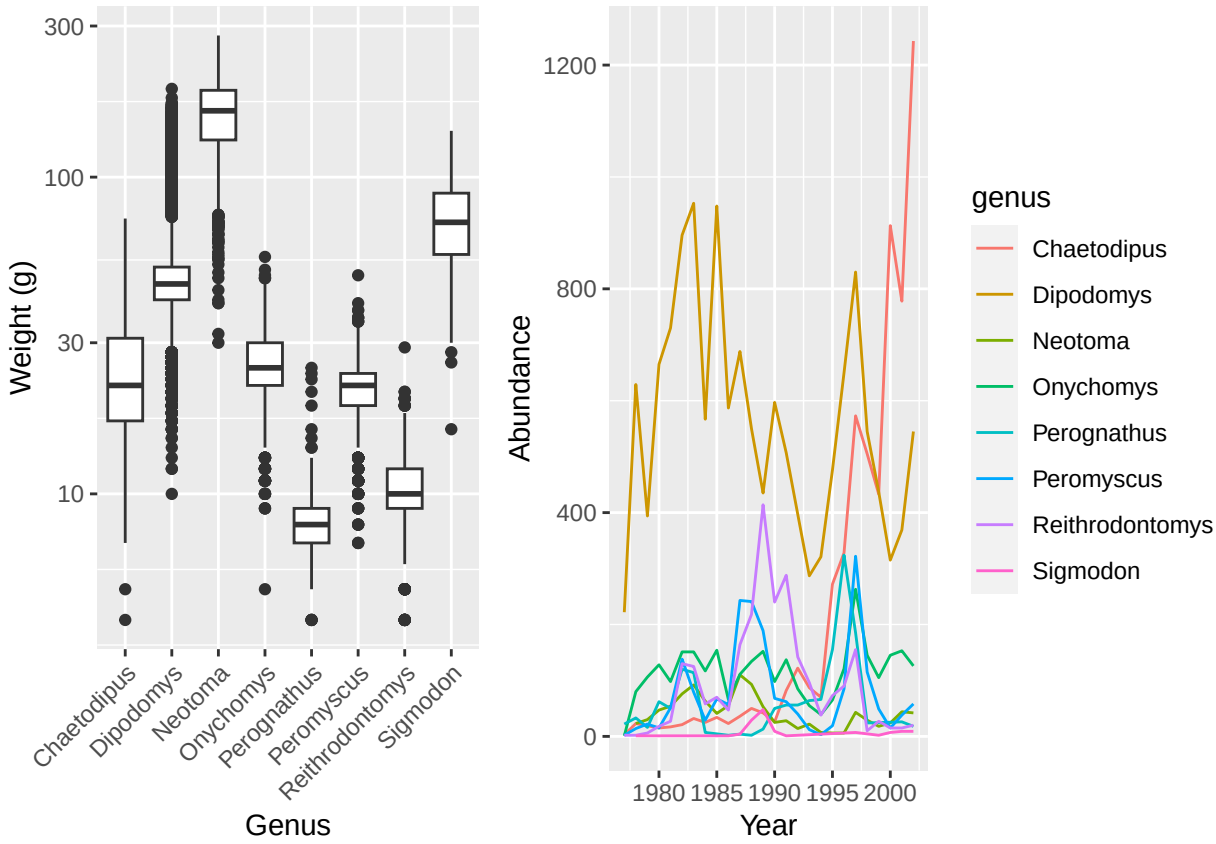
```
# create the fig directory if it does not exist
if (!dir.exists("fig")) dir.create("fig")

my_plot <- ggplot(data = yearly_sex_counts,
                  mapping = aes(x = year, y = n, colour = sex)) +
    geom_line() +
    facet_wrap(vars(genus)) +
    labs(title = "Observed genera through time",
        x = "Year of observation",
        y = "Number of individuals") +
    theme_bw() +
    theme(axis.text.x = element_text(colour = "grey20", size = 12, angle = 90, hjust = 0.5, vjust = 0.5
                      axis.text.y = element_text(colour = "grey20", size = 12),
          text=element_text(size = 16))

ggsave("fig/yearly_sex_counts.png", my_plot, width = 15, height = 10)

my_plot
```




```
# This also works for grid.arrange() plots
# The package gridExtra provides a number of user-level functions to work with "grid" graphics, notably

#install.packages("gridExtra")
library(gridExtra)
```

```
##
## Attaching package: 'gridExtra'

## The following object is masked from 'package:dplyr':
##
##     combine
```

```
combo_plot <- grid.arrange(spp_weight_boxplot, spp_count_plot, ncol = 2, widths = c(4, 6))
```




```
ggsave("fig/combo_plot_abun_weight.png", combo_plot, width = 10, dpi = 300)
```

```
## Saving 10 x 4.5 in image
```

Note: The parameters `width` and `height` also determine the font size in the saved plot.