Data visualization with ggplot2

Data Carpentry contributors & Allison Theobold, Elijah Meyer, Kelly Loucks

---

Learning Objectives

- Produce scatter plots, boxplots, density plots, and time series plots using `ggplot2`.
- Set universal and local plot settings.
- Describe what aesthetics are and how they are used by ggplot.
- Describe what faceting is and apply faceting to a `ggplot`.
- Modify the aesthetics of an existing ggplot plot (e.g. axis labels, color).
- Build multivariate and customized plots from data in a data frame.
- Arrange multiple plots in a grid format using `grid.arrange()` from `gridExtra`.
-

Export publication ready graphics using ggsave().

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

```
library(tidyverse)
```

Let's load the data we finished working with in the *Data Wrangling in `R`* workshop.

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

```
## Parsed with column specification:
## cols(
##   record_id = col_double(),
##   month = col_double(),
##   day = col_double(),
##   year = col_double(),
##   plot_id = col_double(),
##   species_id = col_character(),
##   sex = col_character(),
##   hindfoot_length = col_double(),
##   weight = col_double(),
##   date = col_date(format = ""),
##   day_of_week = col_character(),
##   plot_type = col_character(),
##   genus = col_character(),
##   species = col_character(),
##   taxa = col_character()
## )
```

Plotting with ggplot2

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 customization of plots.

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

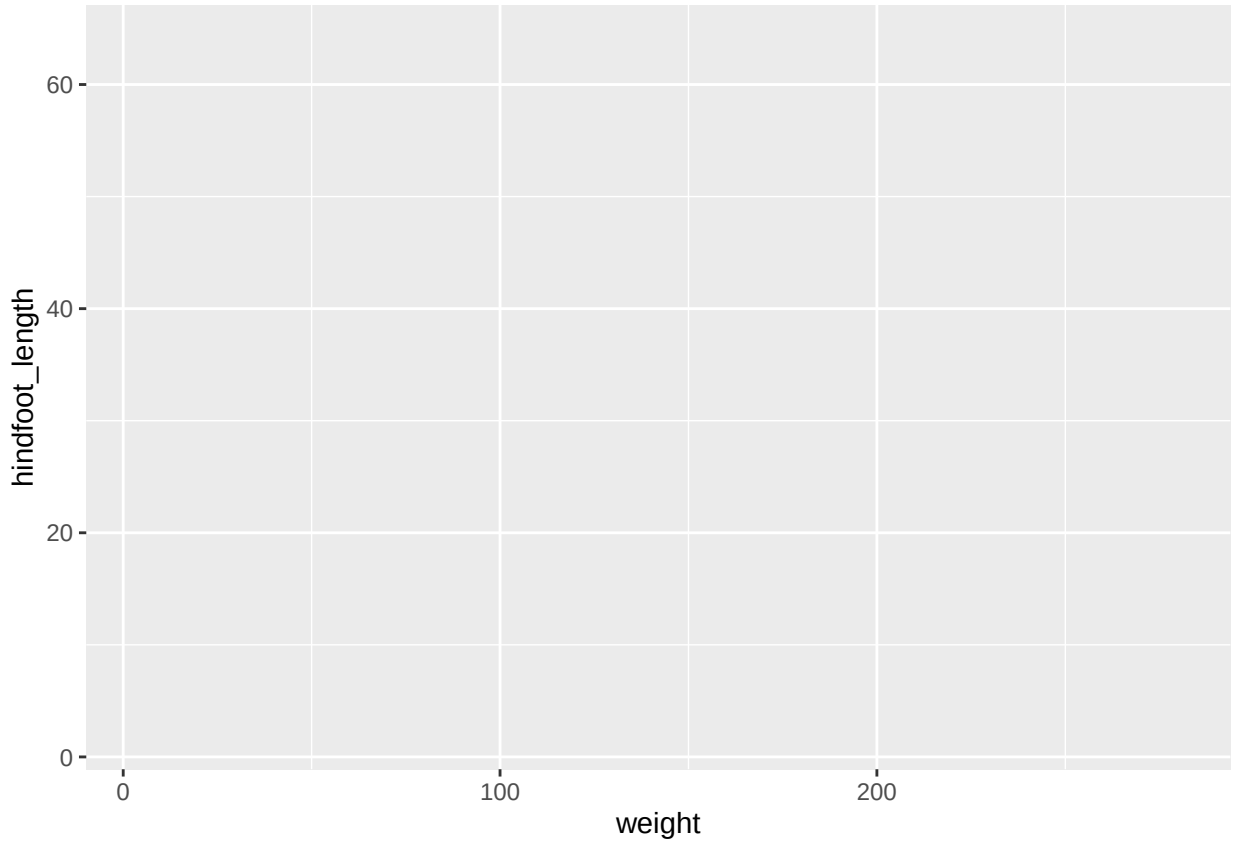
```
ggplot(data = <DATA>, mapping = aes(<MAPPINGS>)) +  <GEOM_FUNCTION>()
```

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

```
ggplot(data = surveys_complete)
## Creates a blank ggplot, referencing the surveys_complete dataset
```

- 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, color, etc.

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



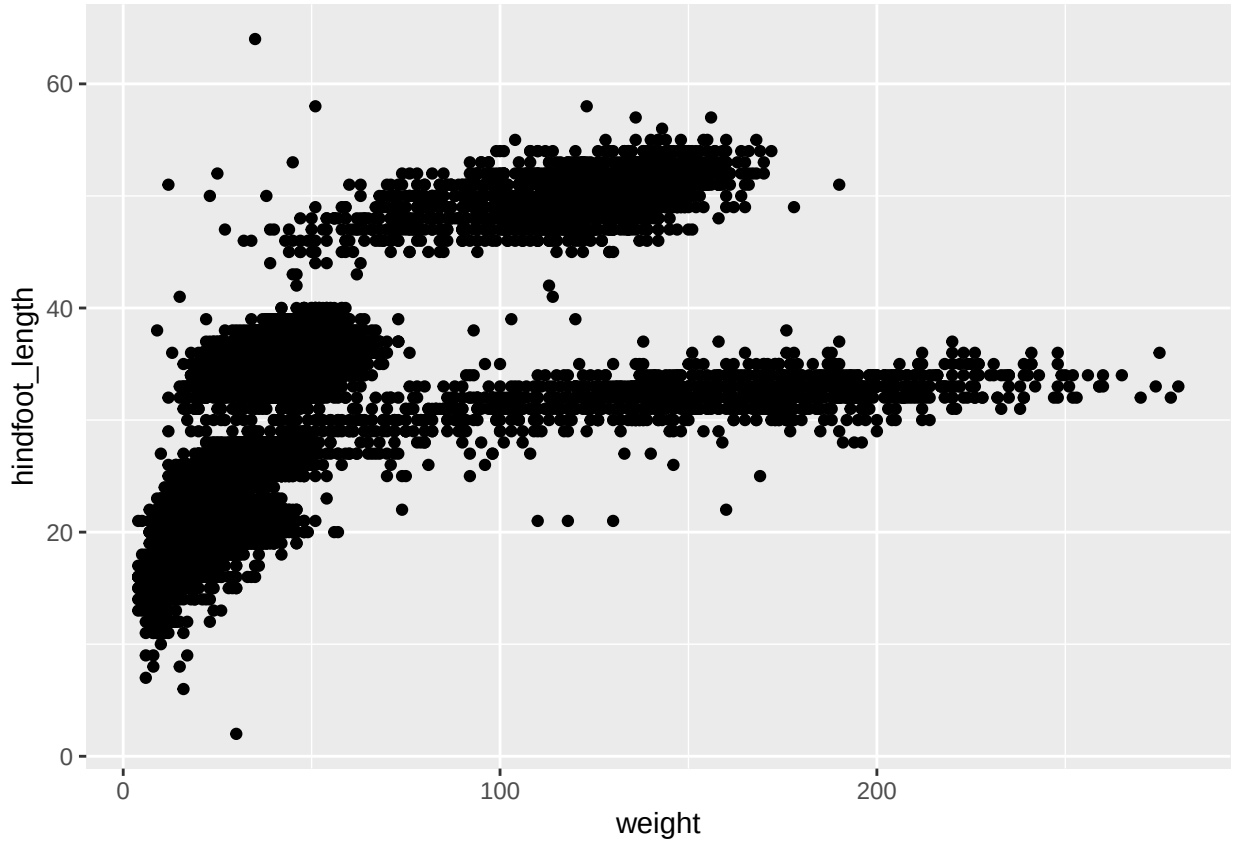
```
## Creates a blank ggplot, with the variables mapped to the x- and y-axis
```

- 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 boxplots
* `geom_bar()` for bar charts
* `geom_line()` for trend lines, time series, etc.
```

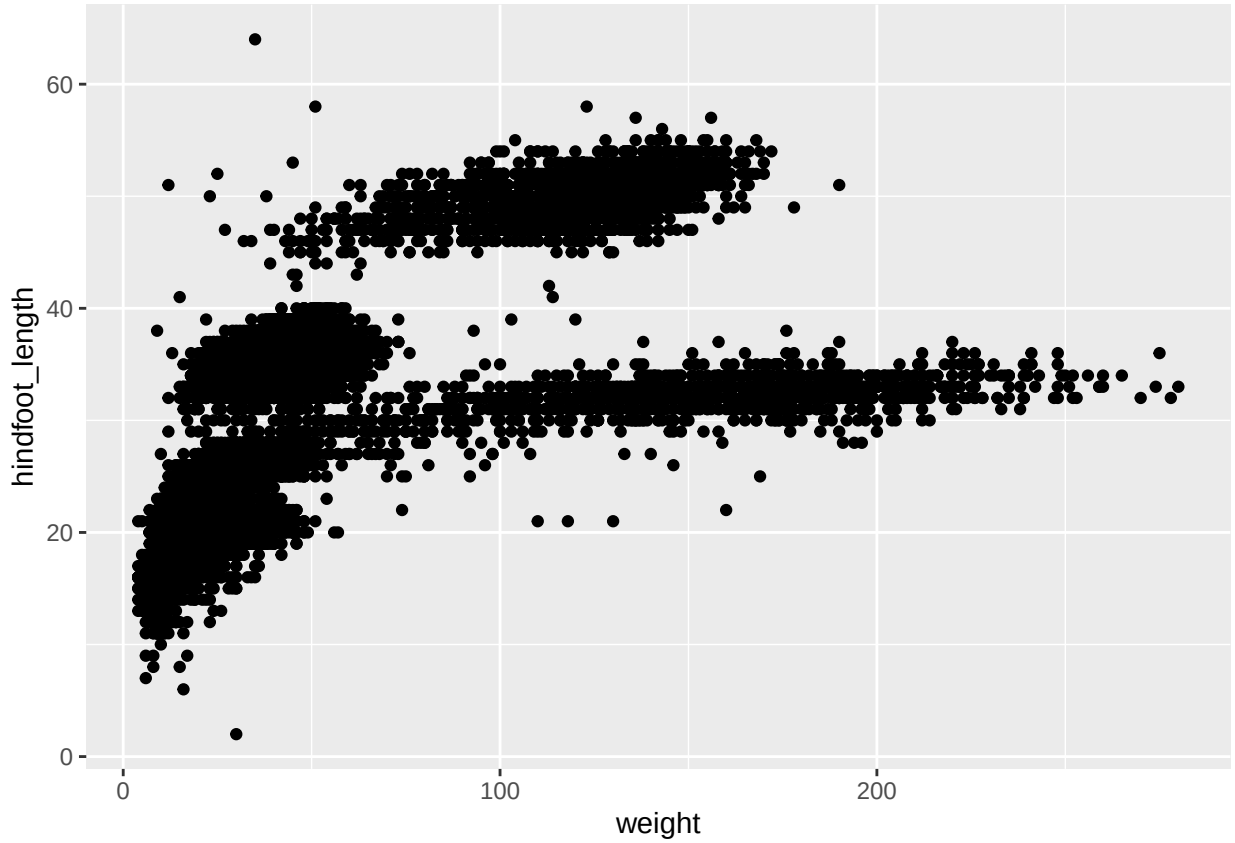
To add a geom to the plot use the `+` operator. Because we have two continuous variables in the data, let's use `geom_point()` first:

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



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))

# Add a specific type of plot (geom) to the defined variables
surveys_plot +
    geom_point()
```



Notes

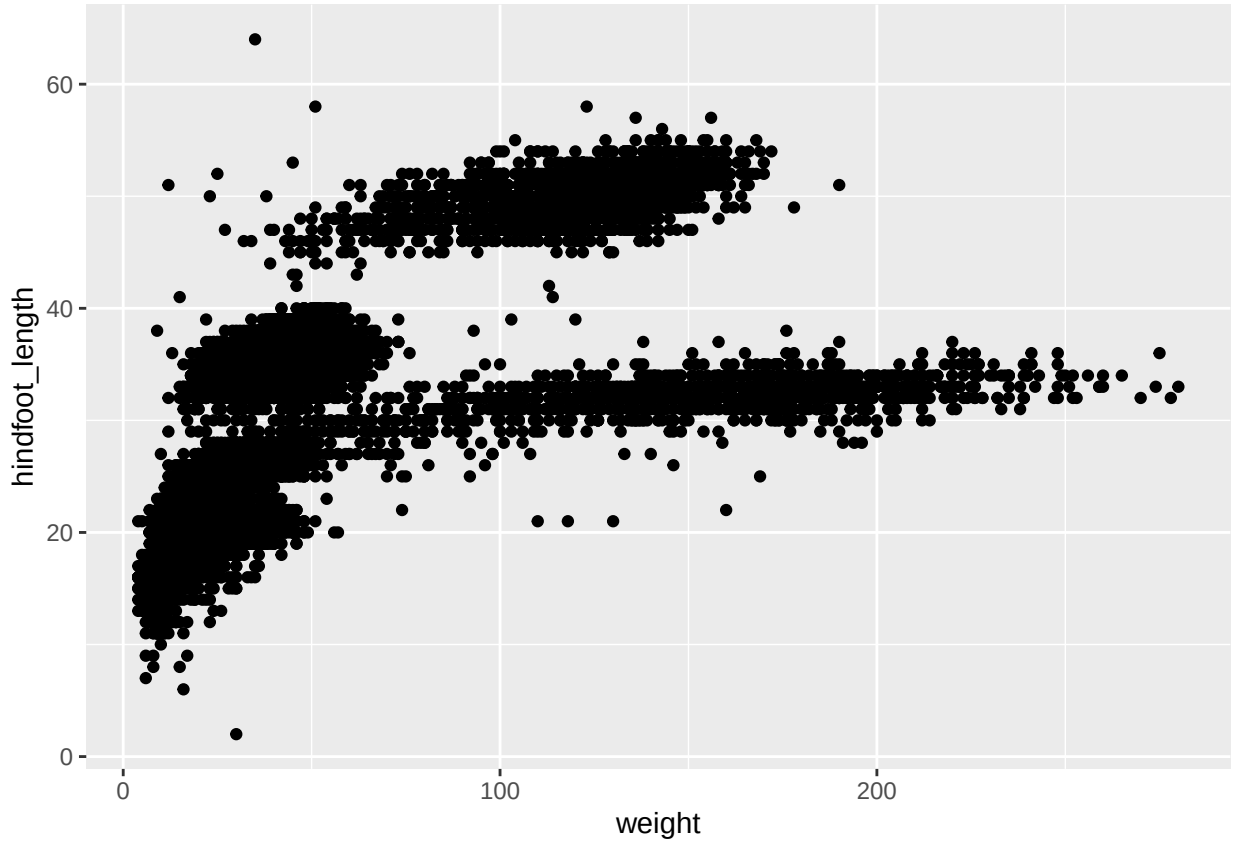
- Anything you put in the `ggplot()` function can be seen by any geom layers that you add (i.e., these are universal plot settings). This includes the x- and y-axis mapping you set up in `aes()`.
- You can also specify mappings for a given geom independently of the mappings defined globally in the `ggplot()` 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 Plots Iteratively

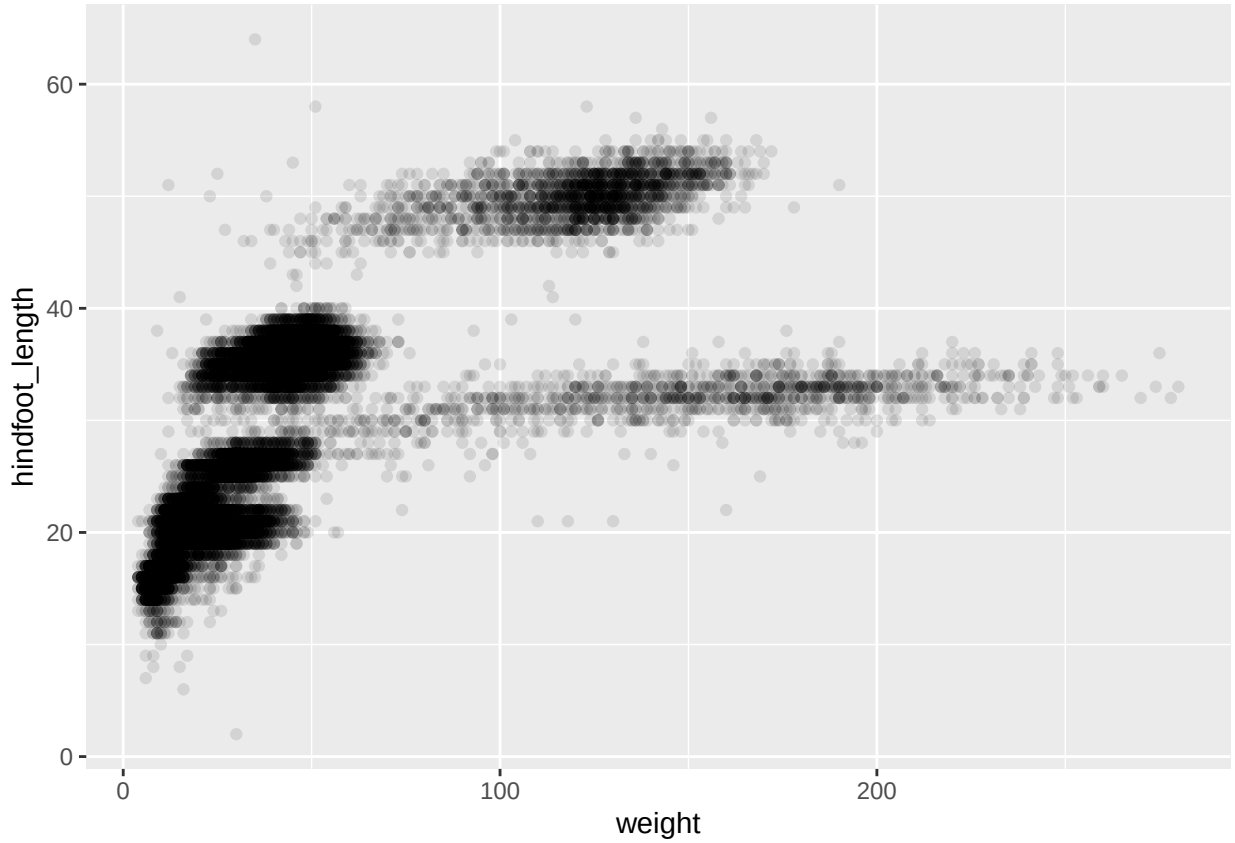
Building plots with **ggplot2** is typically an iterative process. We start by defining the dataset we'll use, lay out the axes, and choose a geom:

```
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:

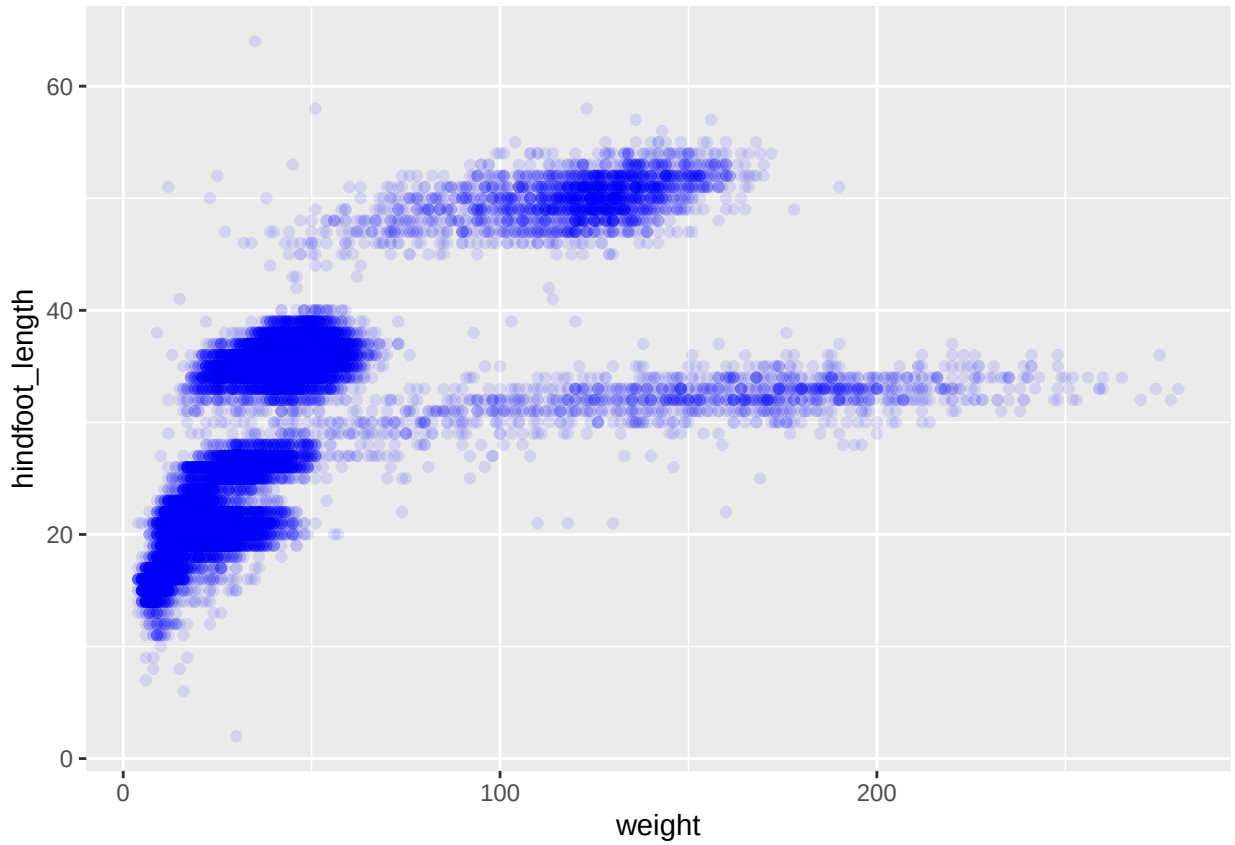
```
ggplot(data = surveys_complete,
       mapping = aes(x = weight, y = hindfoot_length)) +
    geom_point(alpha = 0.1)
```




```
    ## alpha reduces the opacity of the points
    ## 0 is fully transparent
    ## 1 is the original opacity
```

We can also add colors for all the points:

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



Challenge

`geom_point` also accepts aesthetics of size and shape. The size of a point is its width in mm. The shape of a point has five different options for plotting:

- an integer [0, 25] of defined plotting characters – same as base `R`
- the name of the shape in quotations (e.g. "circle open" or "diamond filled")
- a single character, to use that character as a plotting symbol
- a "." to draw the smallest point that is visible – typically 1 pixel
- an NA, to draw nothing

Reference for shapes in integers and characters:
https://ggplot2.tidyverse.org/articles/ggplot2-specs.html

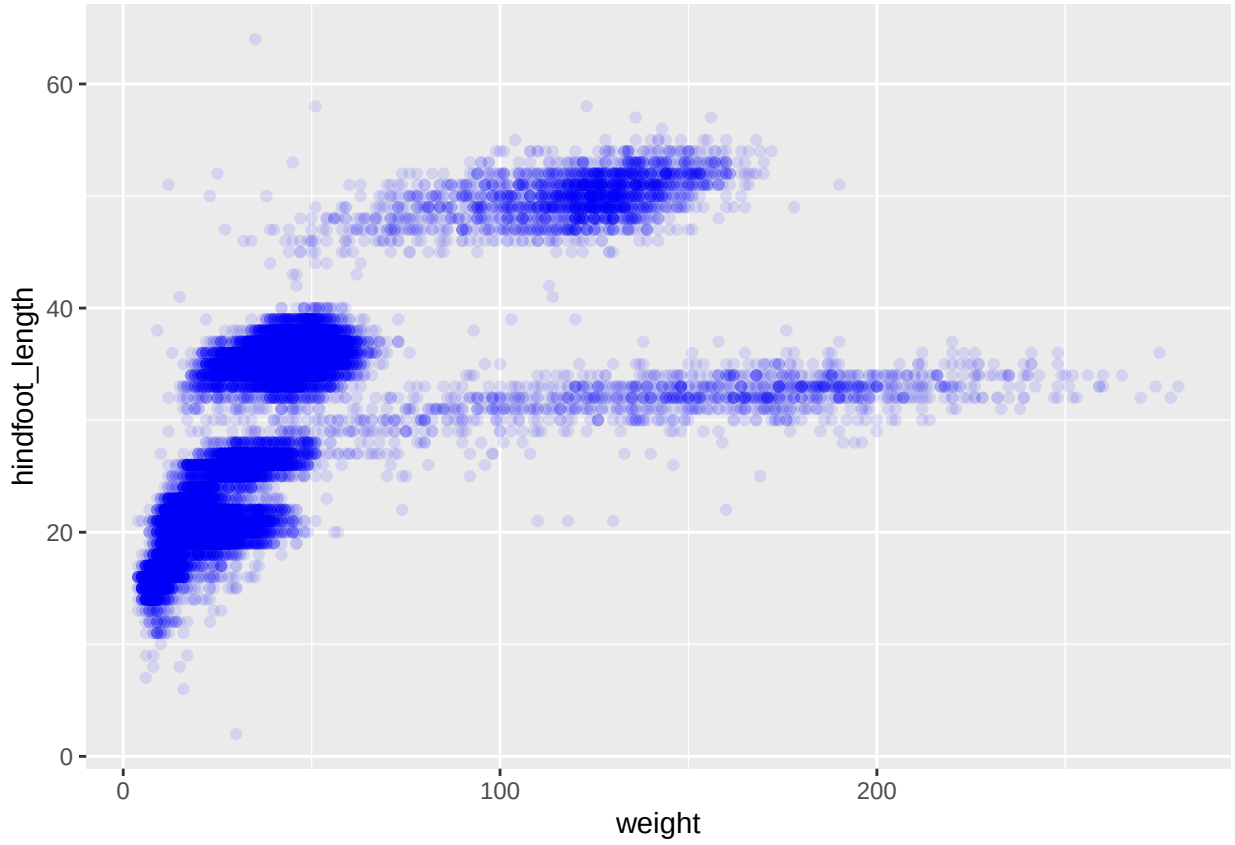
> Assign one of these aesthetics to the `geom_point` aspect of your plot. What happened?

Piping the Data In

Because **`ggplot2`** lives in the `tidyverse`, it is expected to work well with other packages in the `tidyverse`. Because of this, the first argument to creating a `ggplot` is the dataset you wish to be working with. Since the data is the first argument, we can use our familiar pipe operator (`%>%`) to pipe the data into the `ggplot()` function!

This would instead look like this:

```r
surveys_complete %>%
  ## data to be used in the ggplot
  ggplot(mapping = aes(x = weight, y = hindfoot_length)) +
  ## uses the data piped in as the first arguement to ggplot()
    geom_point(alpha = 0.1, color = "blue")
```



Once we pipe the data in, the first argument becomes the `mapping` of the aesthetics. Technically, this is a named argument, which is why it looks like:

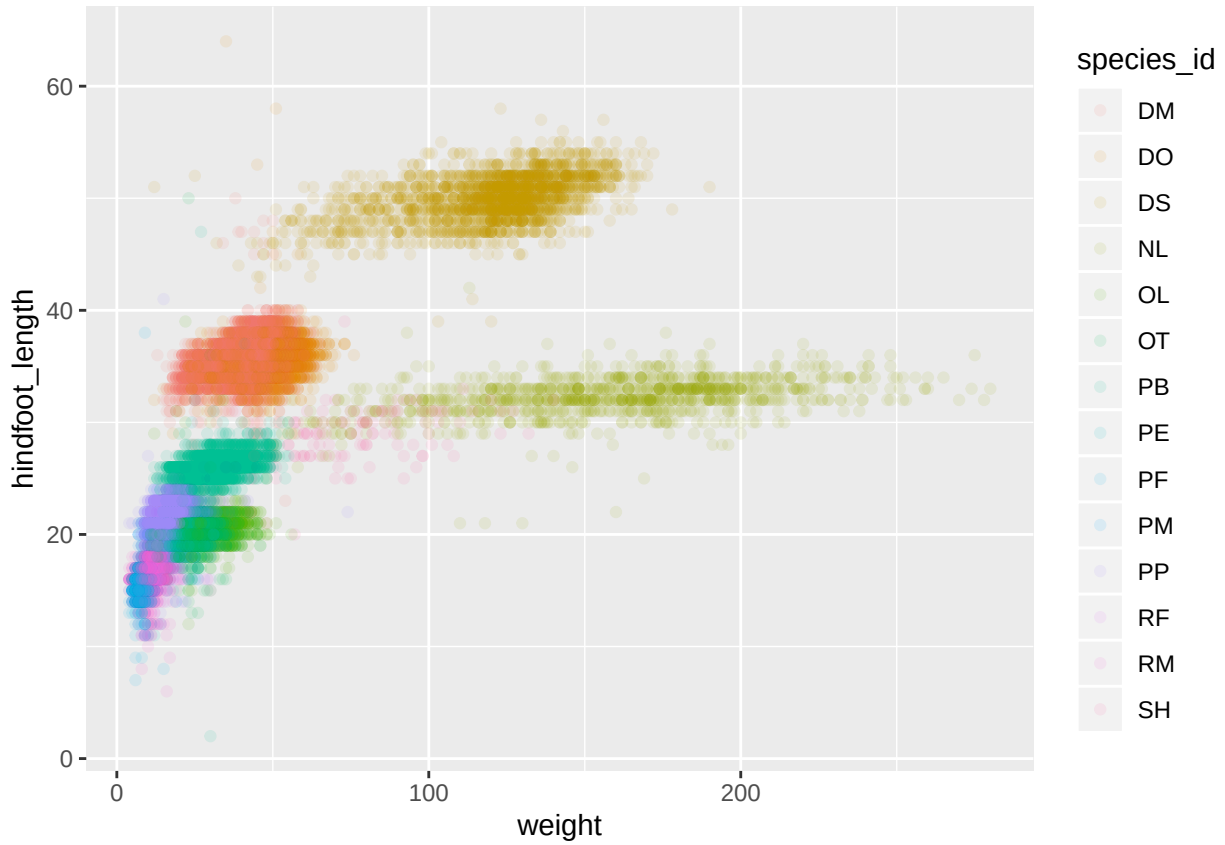
`mapping = aes(<VARIABLES>)`

When we pipe our data in, the first argument then becomes this `mapping` argument. Therefore, we can be lazy and leave the argument unnamed (e.g. `aes(<VARIABLES>)`) and `R` will still know what we are talking about!

Assigning More Variables to Aesthetics

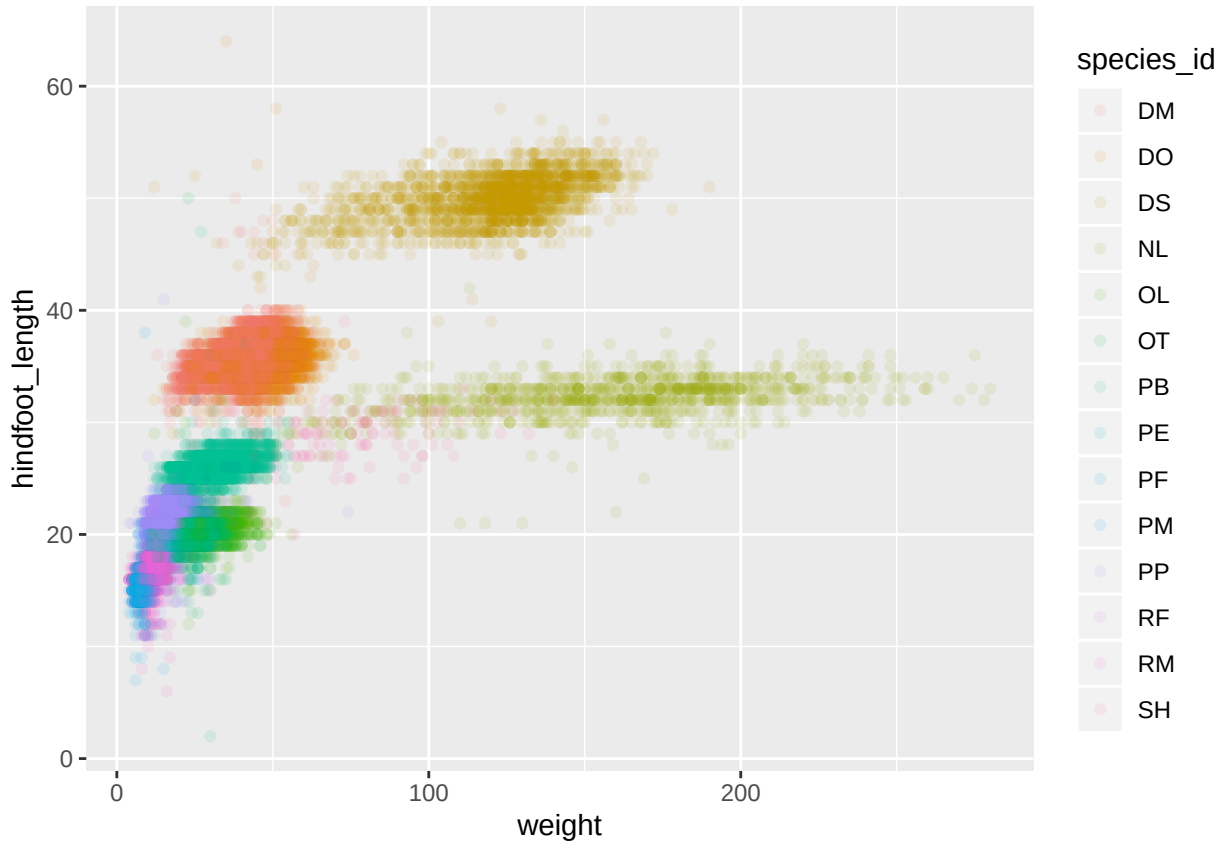
Or to color each species in the plot differently, you could use a vector as an input to the argument **color**. **ggplot2** will provide a different color corresponding to different values in the vector. Here is an example where we color with **species_id**:

```r
surveys_complete %>%
  ggplot(mapping = aes(x = weight, y = hindfoot_length)) +
    geom_point(alpha = 0.1, aes(color = species_id))
```



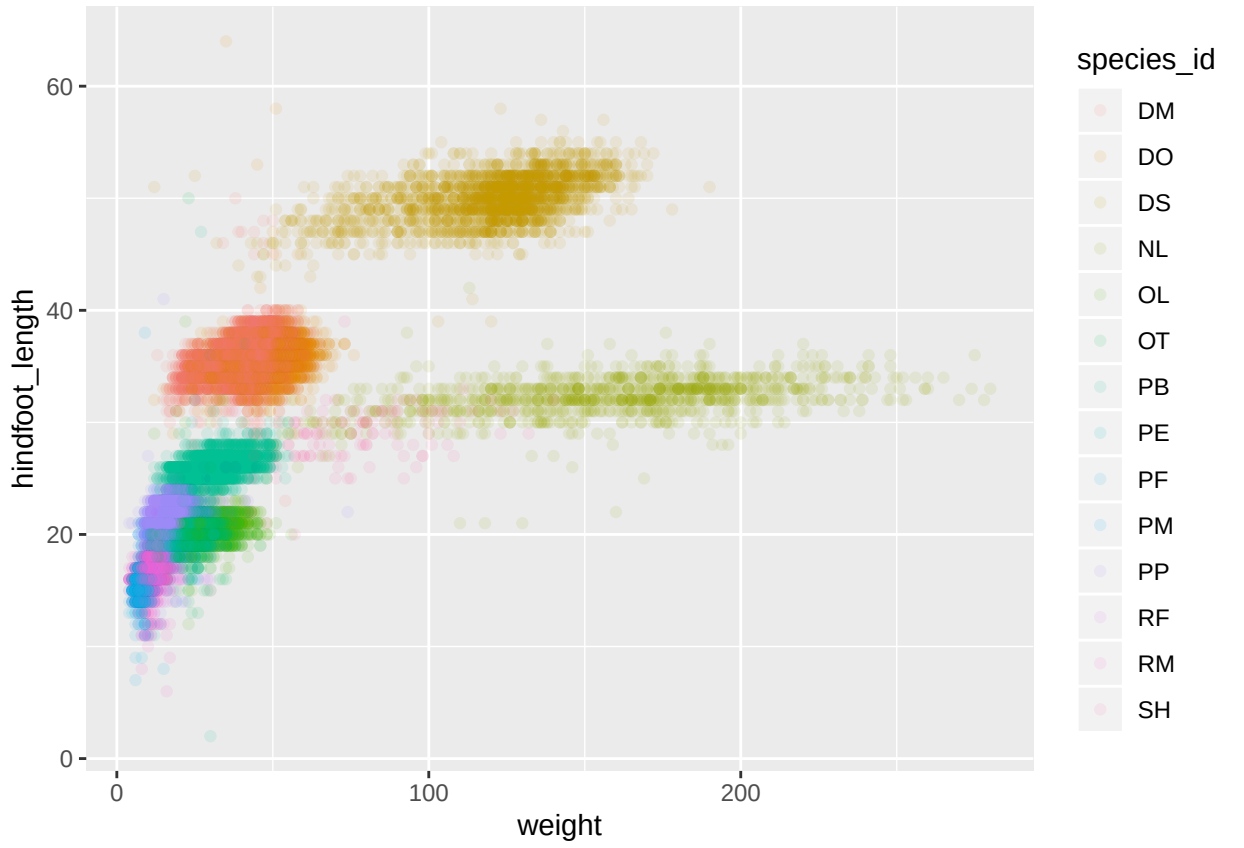
We can also specify the colors directly inside the mapping provided in the `ggplot()` function. This will be seen by **any** geom layers and the mapping will be determined by the x- and y-axis set up in `aes()`.

```
surveys_complete %>%
  ggplot(mapping = aes(x = weight, y = hindfoot_length, color = species_id)) +
    geom_point(alpha = 0.1)
```



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

```
surveys_complete %>%
  ggplot(mapping = aes(x = weight, y = hindfoot_length, color = species_id)) +
    geom_point(alpha = 0.1)
```



Local Aesthetics versus Global Aesthetics

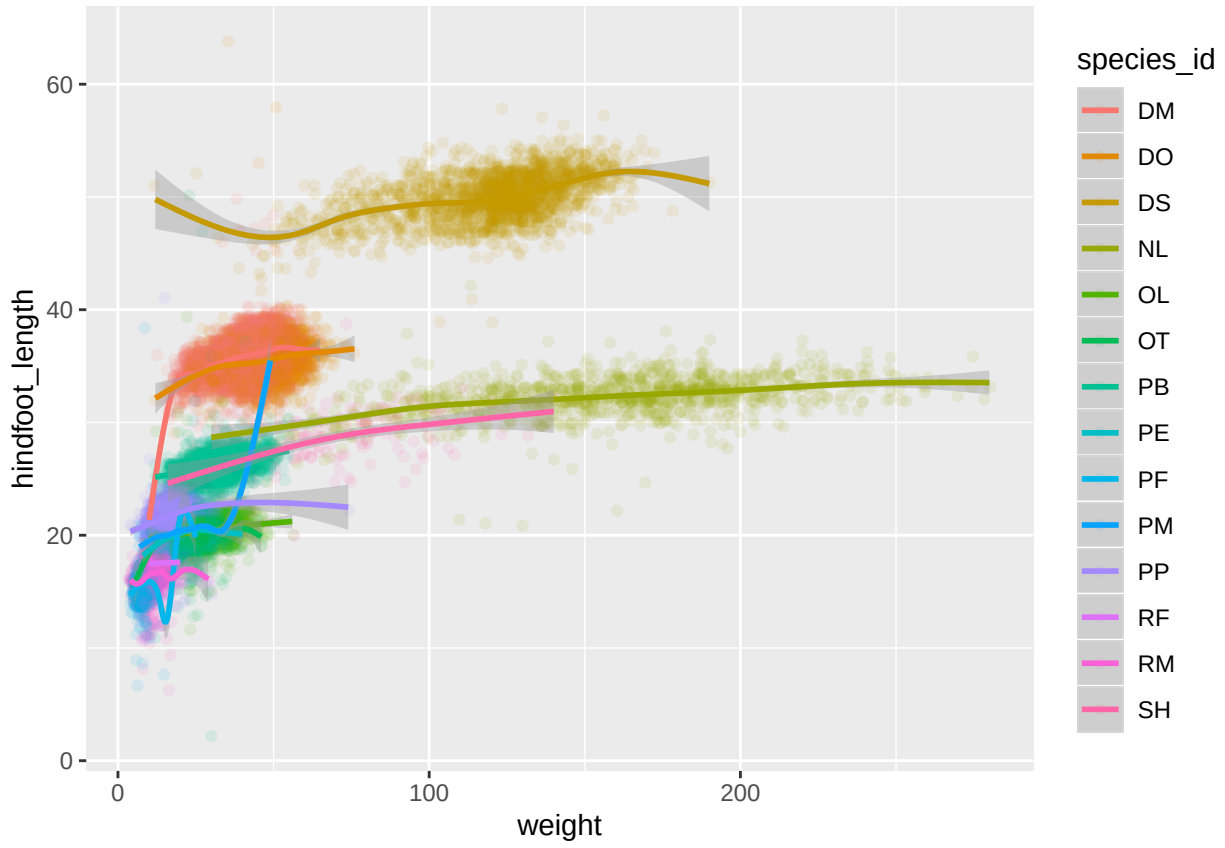
When you define aesthetics in the `ggplot()` function, those mappings hold for **every** aspect of your plot.

For example, if you chose to add a trend line to your plot of weight versus hindfoot length, you would get different lines depending on where you define your color aesthetics.

Globally

```
surveys_complete %>%
  ggplot(mapping = aes(x = weight, y = hindfoot_length, color = species_id)) +
  geom_jitter(alpha = 0.1) +
  geom_smooth()
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```

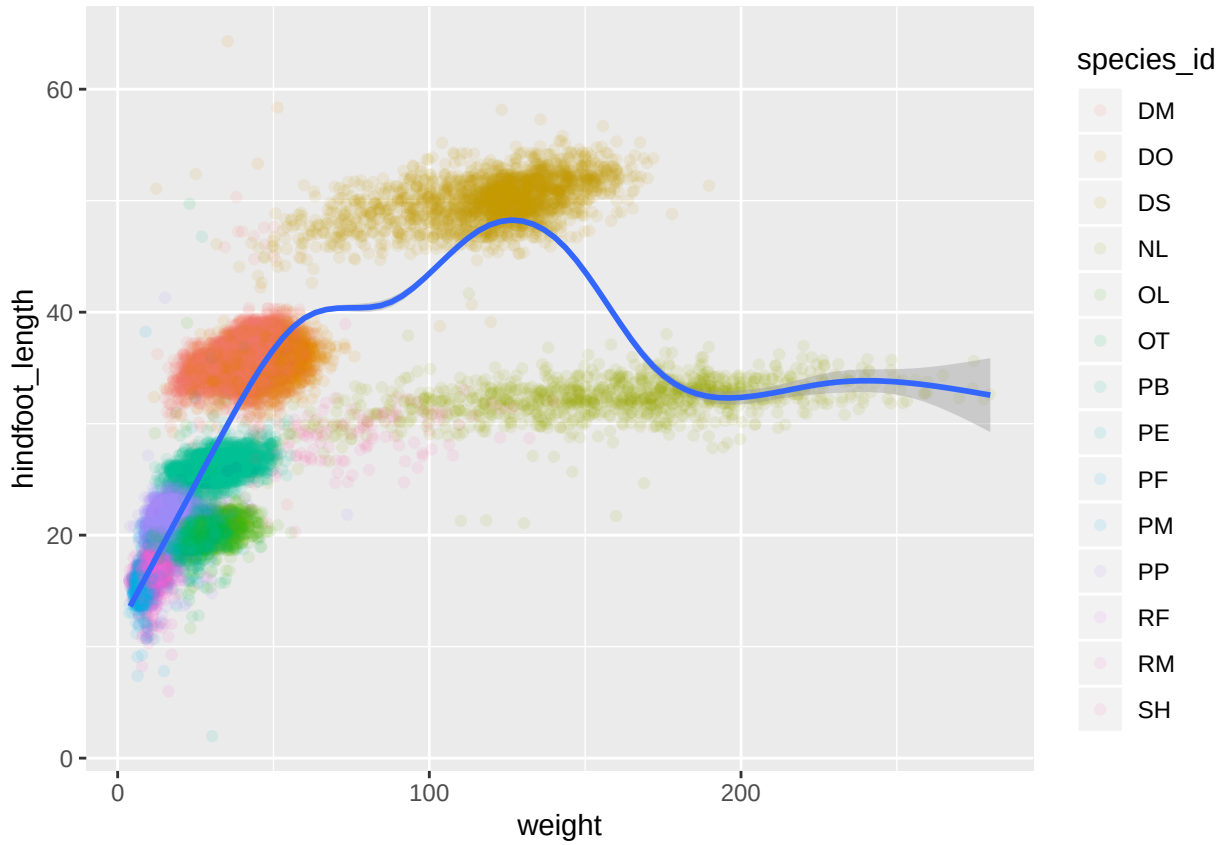



```
  ## trend line for each species_id -- because color is defined globally
```

Locally

```
surveys_complete %>%
  ggplot(mapping = aes(x = weight, y = hindfoot_length)) +
  geom_jitter(aes(color = species_id), alpha = 0.1) +
  geom_smooth()
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```




```
## one trend line -- no color defined globally
```

Challenge

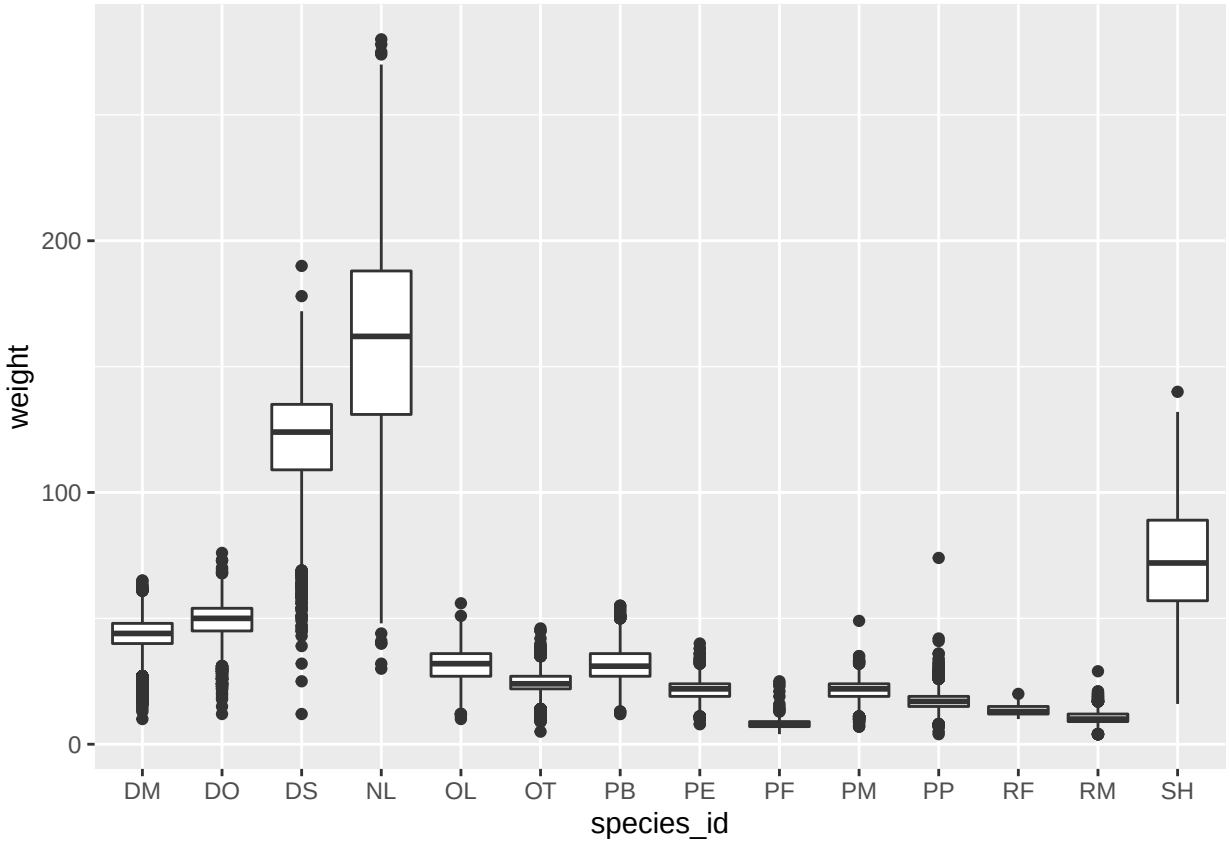
Part 1: Inspect the `geom_point` help file to see what other aesthetics are available. Map a new variable from the dataset to another aesthetic in your plot. What happened? Does the aesthetic change if you use a continuous variable versus a categorical/discrete variable?

Part 2: Use what you just learned to create a scatter plot of `weight` over `plot_id` with data from different plot types being showed in different colors. Is this a good way to show this type of data?

Boxplots & Violin Plots

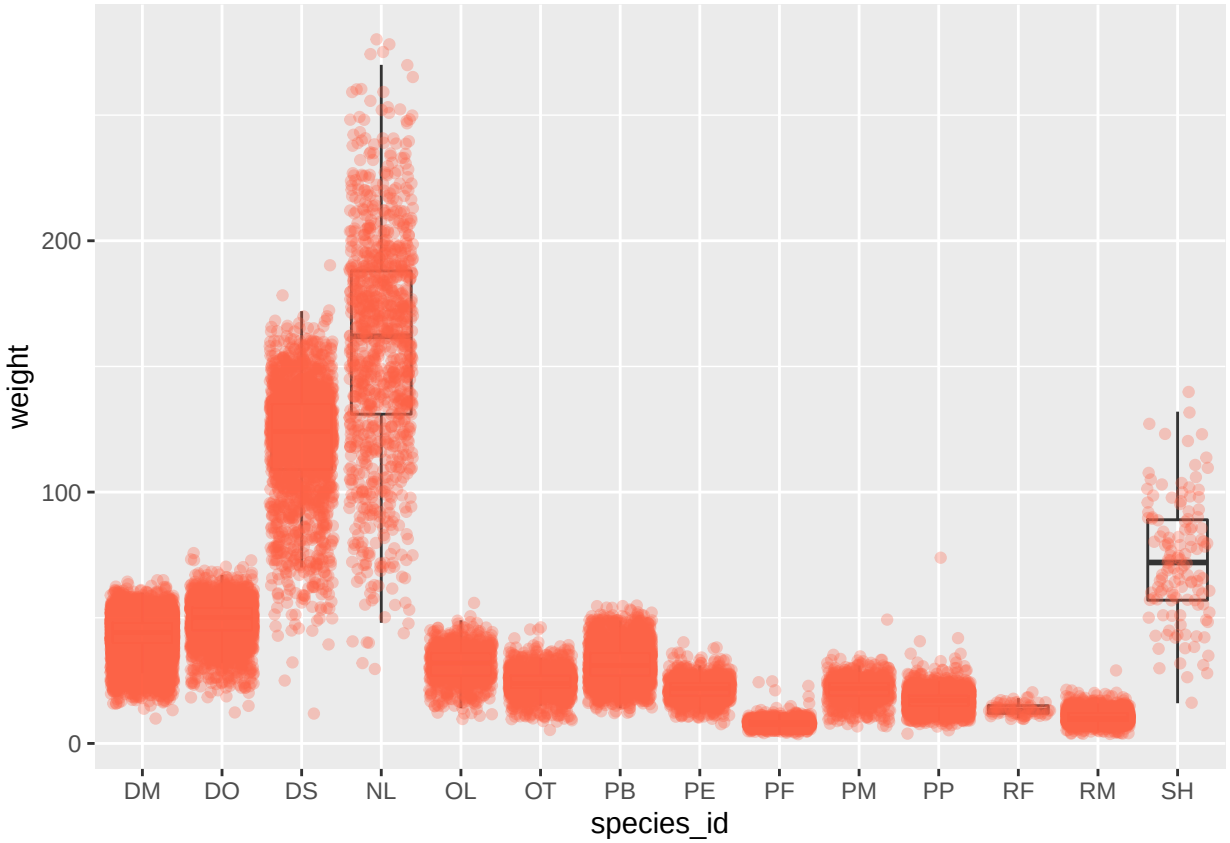
We can use boxplots to visualize the distribution of weight within each species:

```
surveys_complete %>%
  ggplot(mapping = aes(x = species_id, y = weight)) +
    geom_boxplot()
```



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

```
surveys_complete %>%
  ggplot(mapping = aes(x = species_id, y = weight)) +
    geom_boxplot(alpha = 0) +
    ## alpha = 0 eliminates the black (outlier) points, so they're not plotted twice
    geom_jitter(alpha = 0.3, color = "tomato")
```




```
## alpha = 0.3 decreases the opacity of the points, to not be too busy
```

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?

Challenges

Part 1: 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. A superior density plot 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()`.

Part 2: 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 plotting weight on the log (base 10) scale; see `scale_y_log10()`.

Part 3: 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 color 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?

Plotting Single Variables – Quantitative & Categorical

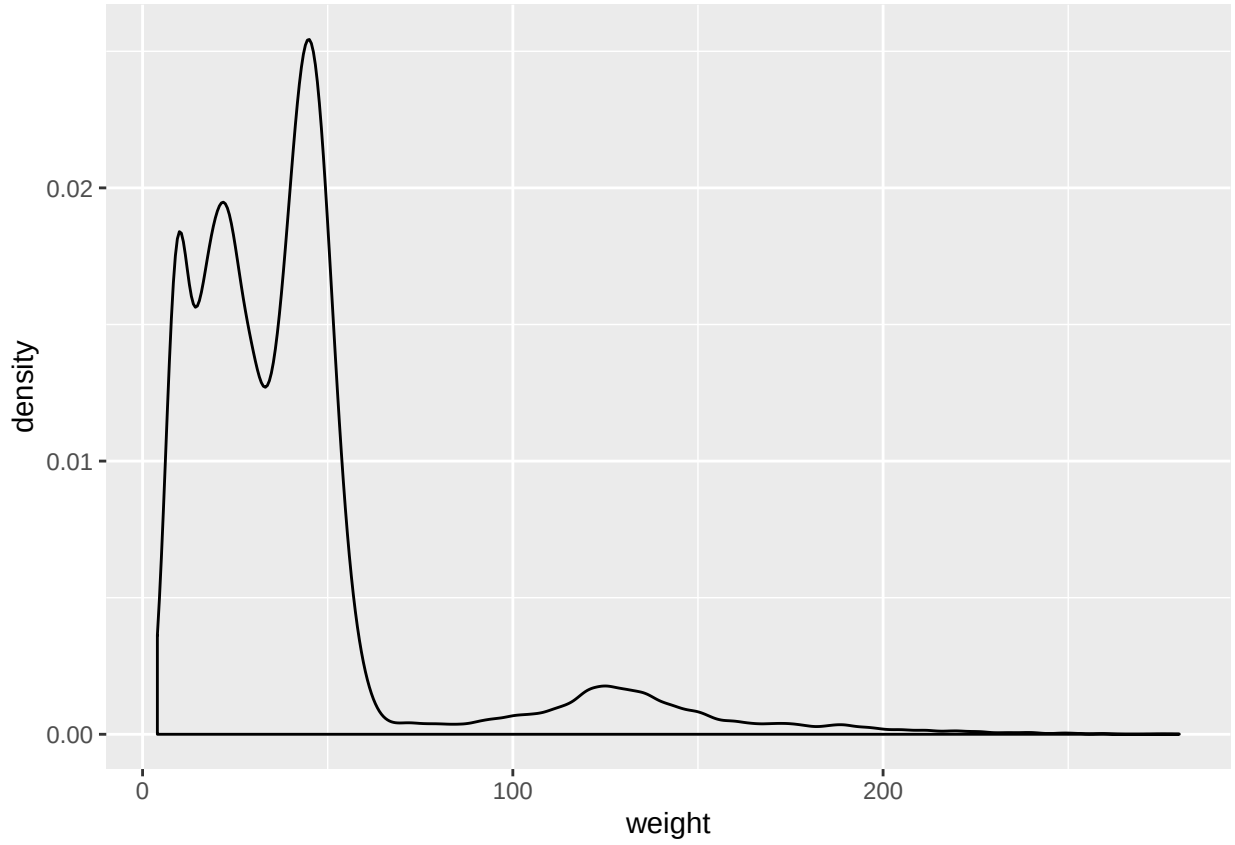
Distribution Plots

If we wish to visualize the distribution of a single quantitative variable, our plot changes a bit. Unfortunately, the `geom_violin()` function only accepts groups, so we cannot make a violin plot with no groups. Darn it!

But, a violin is simply a density plot that's been reflected across the y-axis. So, we could likely suffice with a density plot.

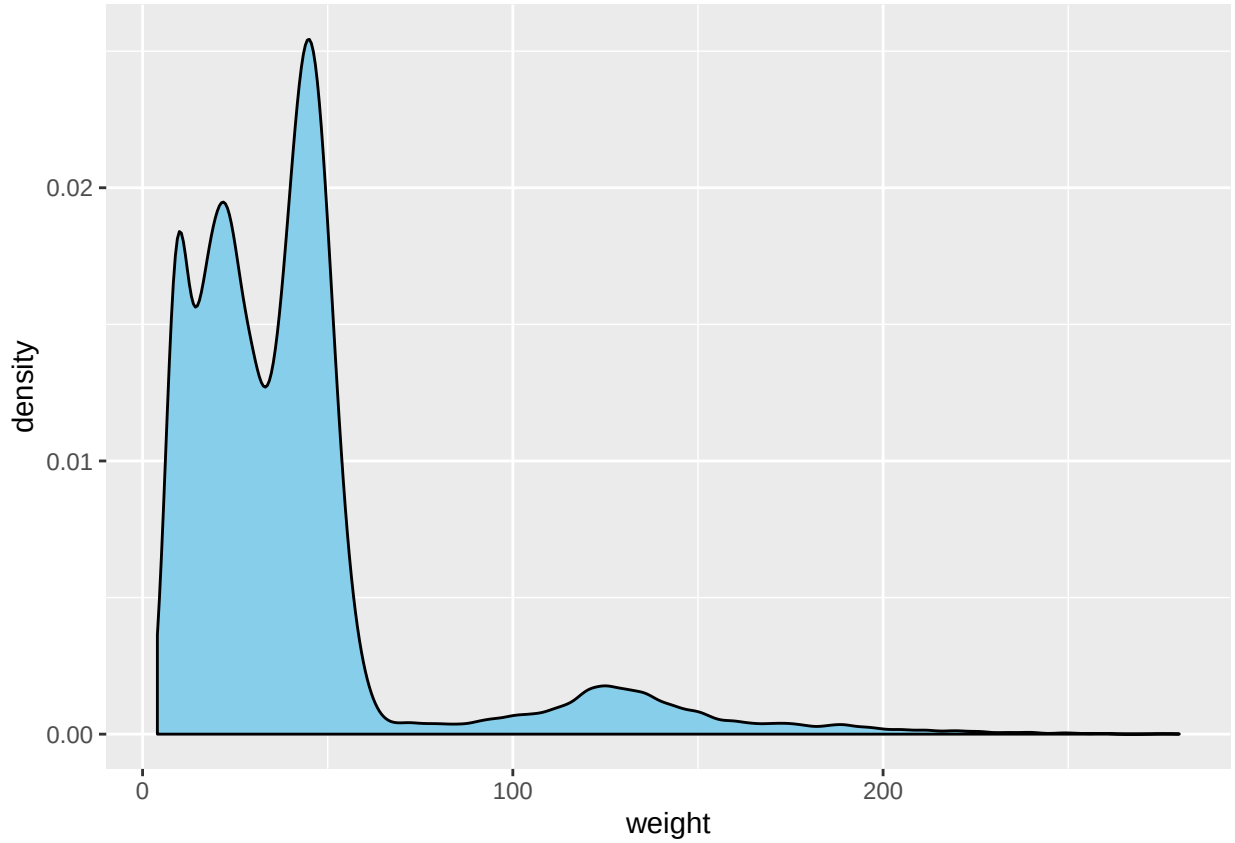
To visualize the distribution of rodent weights we could aggregate over all species, years, plots, etc. and produce a single density plot:

```
surveys_complete %>%
  ggplot(mapping = aes(x = weight)) +
  geom_density()
```



The default is an empty density plot, which is largely unsatisfying. By adding a `fill = <COLOR>` argument to `geom_density()` we can produce a nicer looking plot:

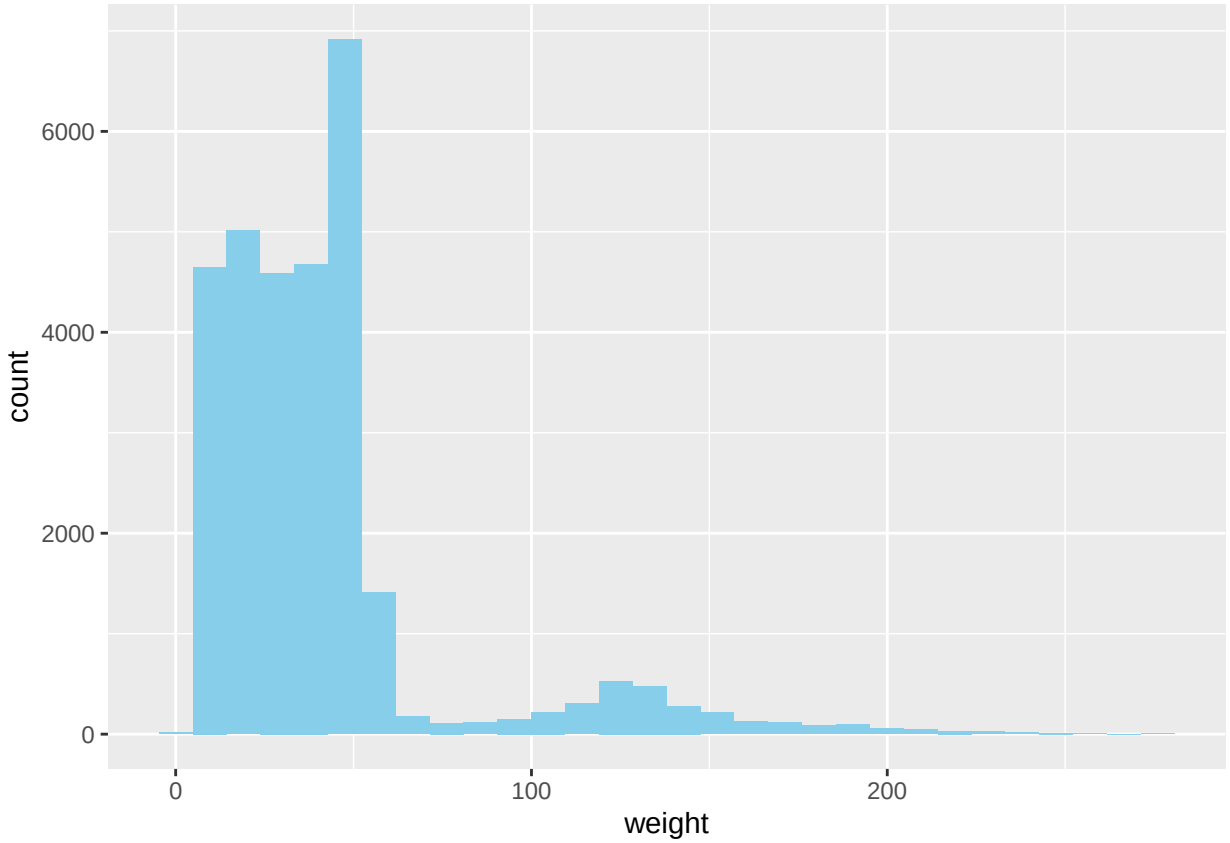
```
surveys_complete %>%
  ggplot(mapping = aes(x = weight)) +
  geom_density(fill = "sky blue")
```



Another frequently used plot for a single quantitative variable is the histogram. The same plot as above can be recreated using `geom_histogram()` instead of `geom_density()`. However, when you use `geom_histogram()` it gives you a warning.

What warning do you get and why? Do you get an error like this when you use `hist()` in base `R`?

```
surveys_complete %>%
  ggplot(mapping = aes(x = weight)) +
  geom_histogram(fill = "sky blue")
```



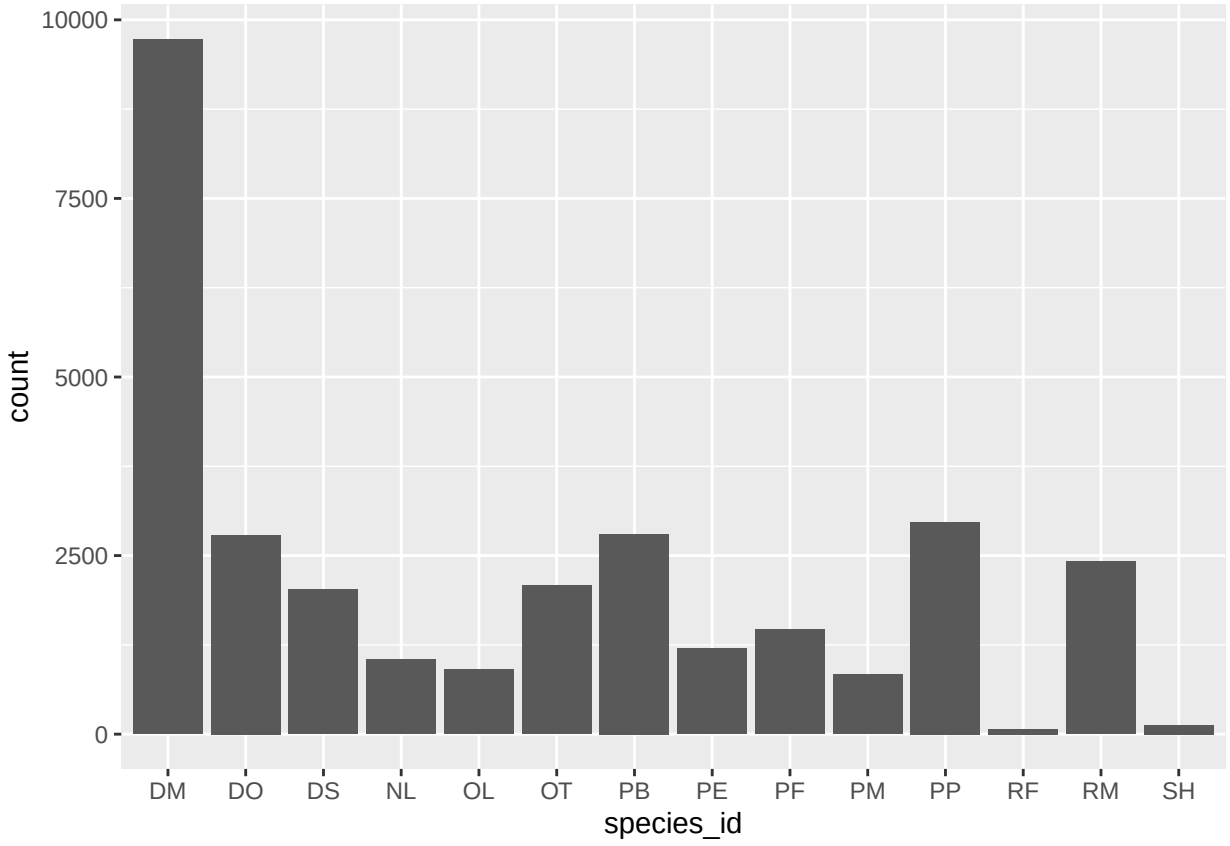
I was once told that the idea behind a good histogram was to make the plot look as smooth as possible – think trying to resemble the continuous shape of the density plot.

> Use the `bins` argument to play around with the number of bins in your histogram. Compare your chosen number of bins with your neighbors!

Bar Charts

At first glimpse, you would think that a bar plot would be simple to create, but bar plots reveal a subtle nuance of the plots we have created thus far. The following bar chart displays the total number of rodents in the `surveys_complete` dataset, grouped by their species ID.

```
surveys_complete %>%
  ggplot(mapping = aes(x = species_id)) +
  geom_bar()
```



The x-axis displays the levels of `species_id`, a variable in the `surveys_complete` dataset. On the y-axis `count` is displayed, but `count` is **not** a variable in our dataset! Where did `count` come from? Graphs, such as the scatterplots, display the raw values of your data. Other graphs, like bar charts and boxplots, calculate new values (from your data) to plot.

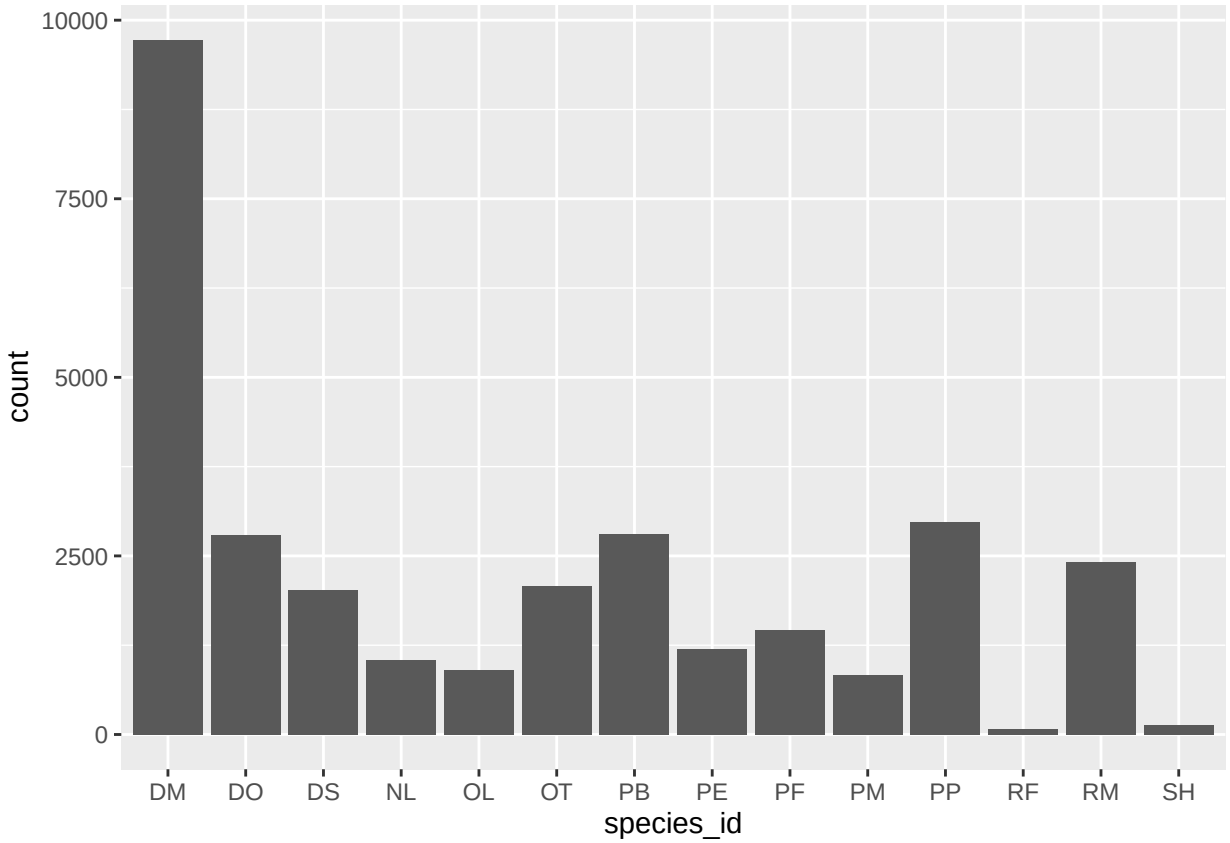
- Bar charts and histograms bin your data and then plot the number of observations that fall in each bin.
- Boxplots find summaries of your data (min, max, quantiles, median) and plot those summaries in a tidy box, with "outliers" (data over 1.5*IQR from min/max) plotted as points.
- Smoothers (as used in `geom_smooth`) fit a model to your data (you can specify, but we used the `loess` default) and then plot the predictions from that model (with associated confidence intervals).

To calculate each of these summaries of the data, `R` uses a different statistical transformation, or *stat* for short. With a bar chart this looks like the following process:

1. `geom_bar` first looks at the entire data frame
2. `geom_bar` then transforms the data using the `count` statistic
3. the `count` statistic returns a data frame with the number of observations (rows) associated with each level of `species_id`
4. `geom_bar` uses this summary data frame, to build the plot – levels of `species_id` are plotted on the x-axis and `count` is plotted on the y-axis

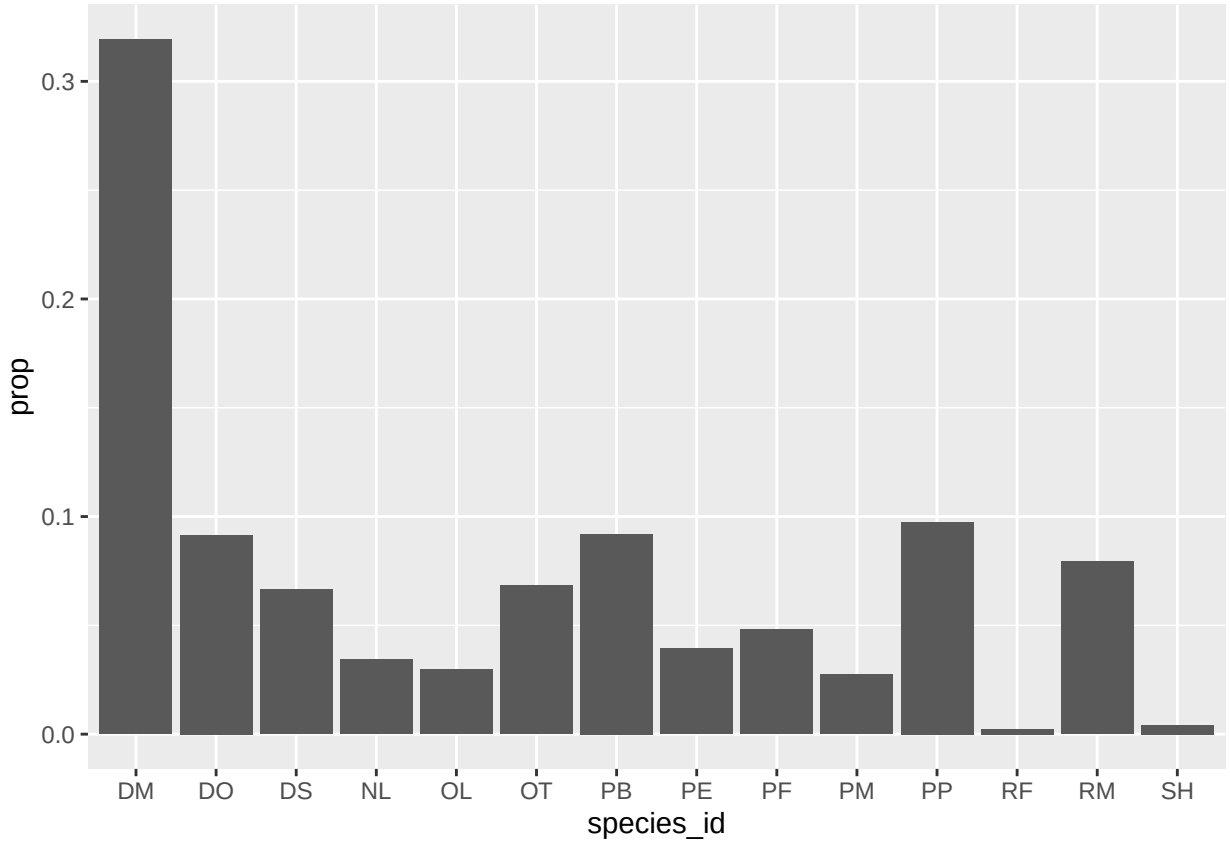
Generally, you can use geoms and stats interchangeably. This is because every geom has a default stat and visa versa. For example, the following code produces the same output as above:

```
surveys_complete %>%
  ggplot(mapping = aes(x = species_id)) +
  stat_count()
```



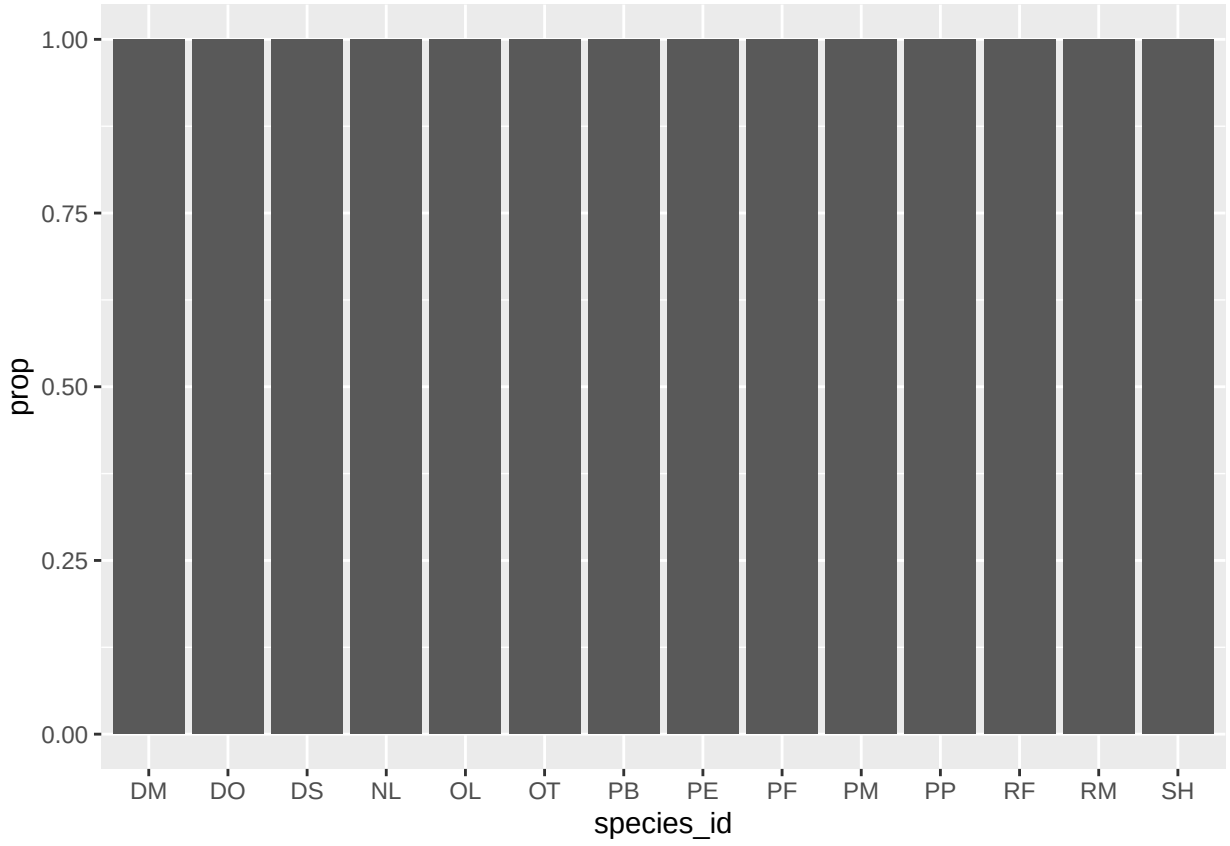
If you so wish you could override the default stat for that geom. For example, if you wanted to plot a bar chart of proportions you would use the following code to override the `count` stat:

```
surveys_complete %>%
  ggplot(mapping = aes(x = species_id)) +
  geom_bar(aes(y = ..prop.., group = 1))
```



Why do we need to set `group = 1` in the above proportion bar chart? In other words, what is wrong with the plot below?**

```
surveys_complete %>%
  ggplot(aes(x = species_id)) +
  geom_bar(aes(y = ..prop..))
```

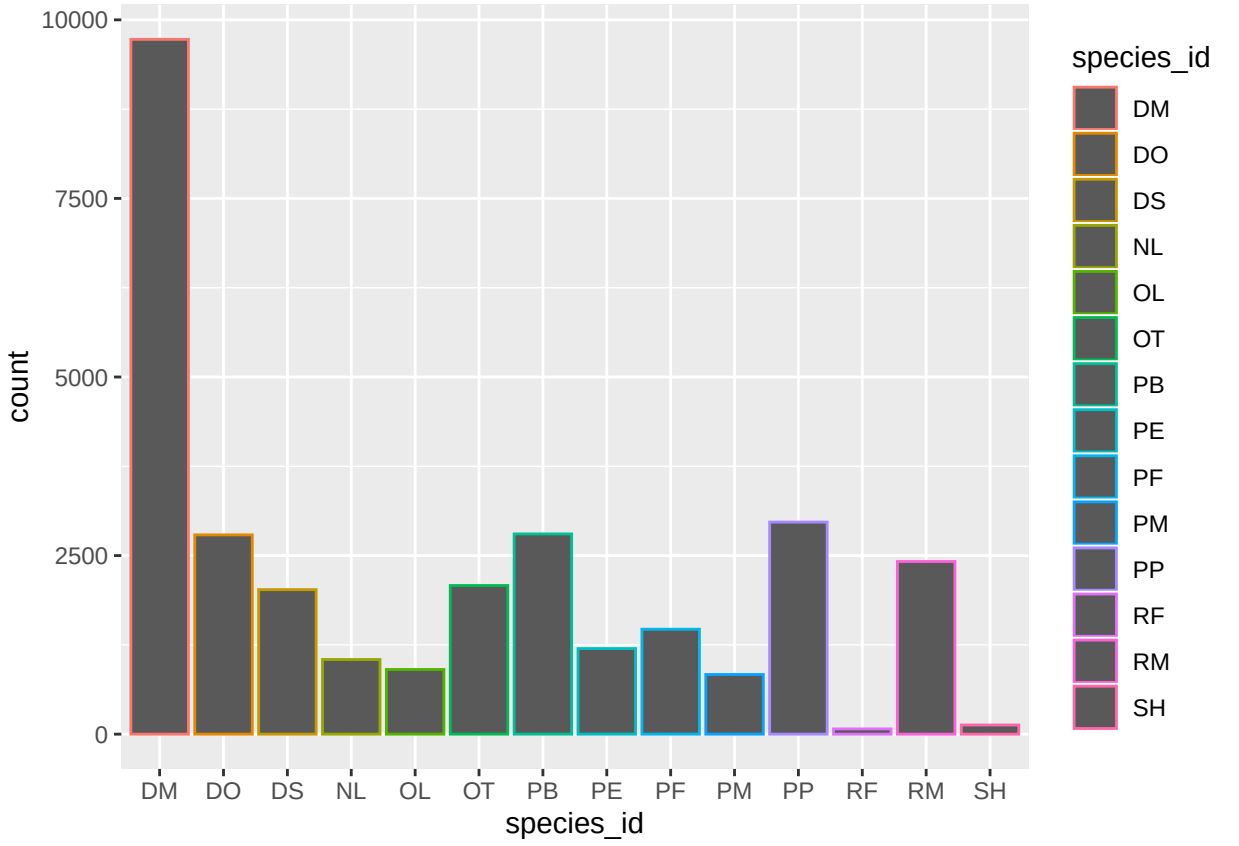


Colored and/or Stacked Bar Charts

Another piece of visual appeal to creating a bar chart is the ability to use colors to differentiate the different groups, or to plot two different variables in one bar chart (stacked bar chart). Let's start with adding color to our bar chart.

As we saw before, to add a color aesthetic to the plot we need to map it to a variable. However, if we use the `color` option that we used before we get a slightly unsatisfying result.

```
surveys_complete %>%
  ggplot(mapping = aes(x = species_id, color = species_id)) +
  geom_bar()
```



We notice that the color only appears in the outline of the bars. For a bar chart, the aesthetic that we are interested in is the `fill` of the bars.

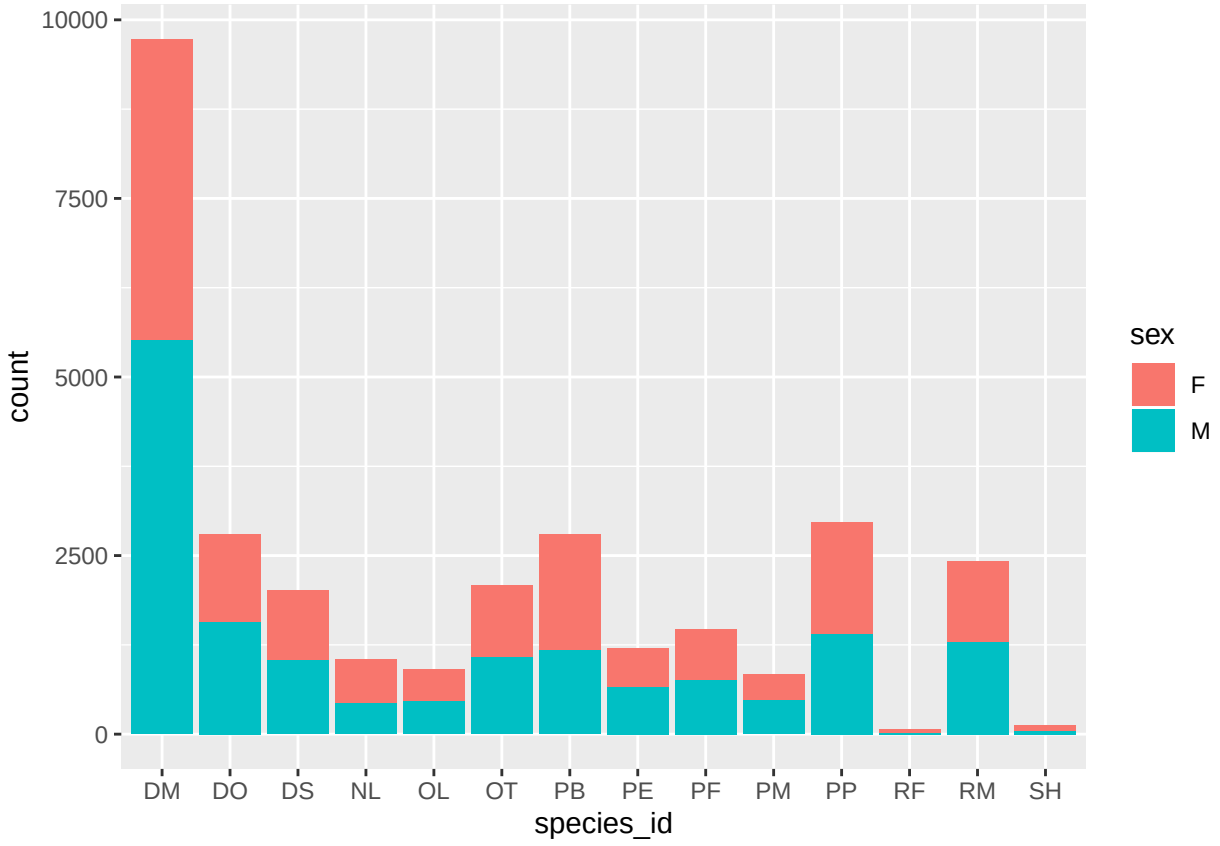
Change the above code so that each bar is filled with a different color.

Now suppose you are interested in whether the number of rodents in each species captured differs by sex. This would require for you to create a bar plot with two categorical variables. You have two options:

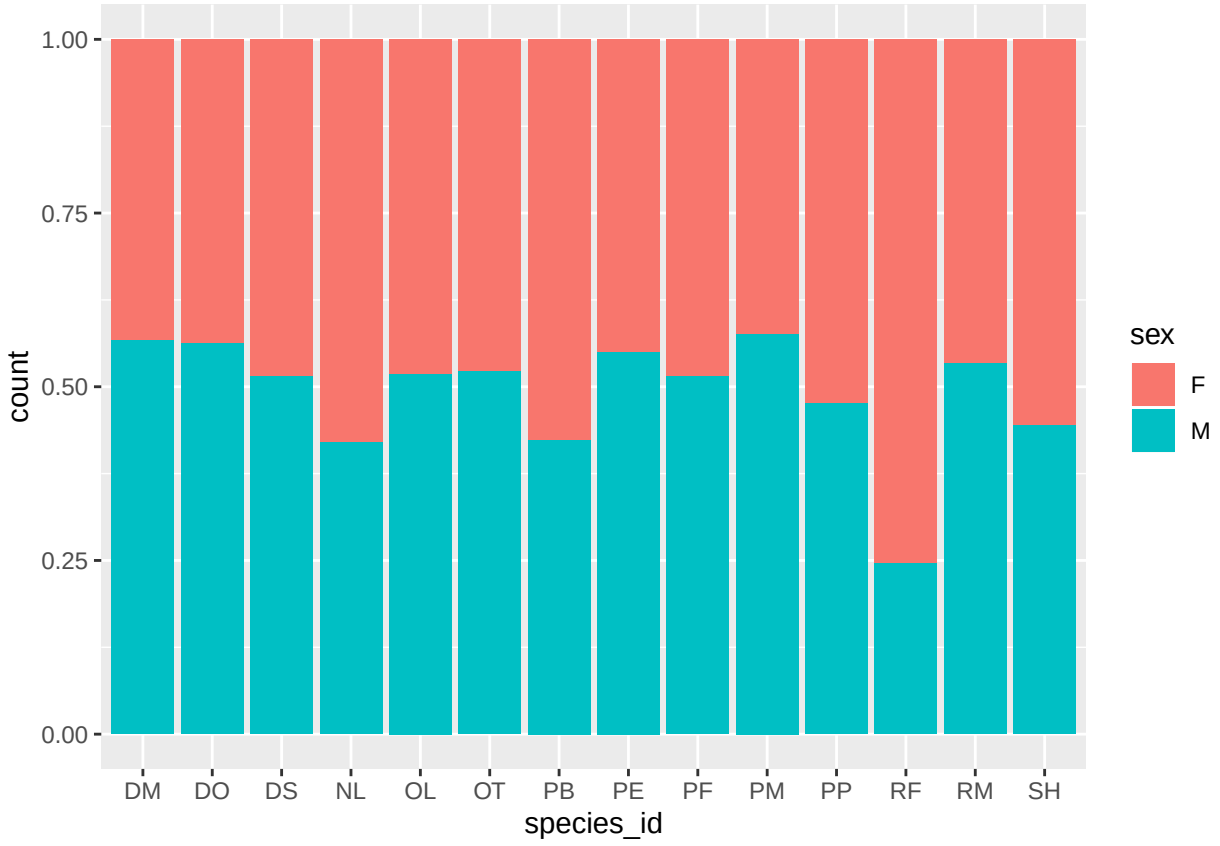
1. each of the bars for sex could be stacked within a species *OR*
2. the bars for sex could be side-by-side within a species

Let's see how the two approaches differ. To stack bars of a second categorical variable we would instead use this second categorical variable as the `fill` of the bars. Run these two lines of code and see how they differ.

```
surveys_complete %>%
  ggplot(mapping = aes(x = species_id, fill = sex)) +
  geom_bar()
```



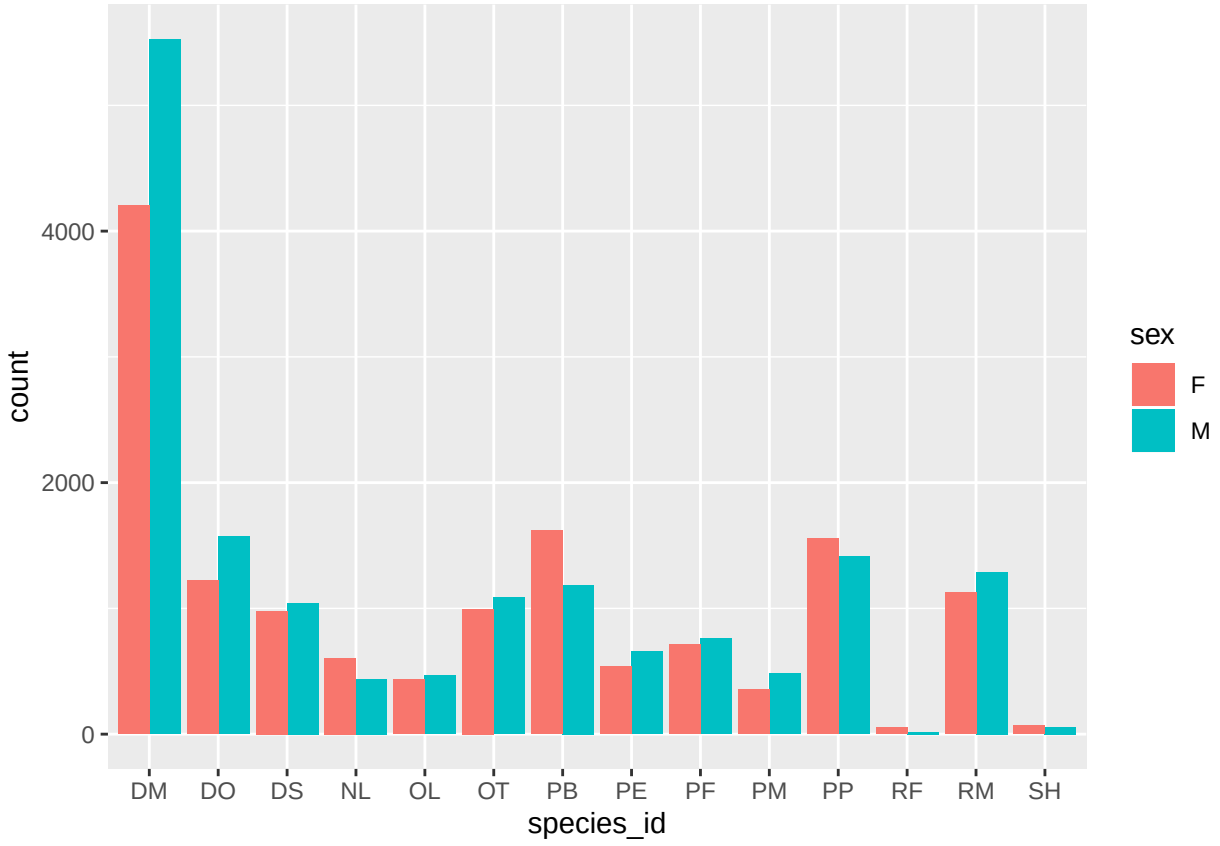

```
surveys_complete %>%
  ggplot(mapping = aes(x = species_id, fill = sex)) +
  geom_bar(position = "fill")
```



In the first plot, the position was chosen automatically, but in the second plot the `position` argument was made explicit. What changes did this make in the plots?

Another position option is `position = "dodge"` which places the previously overlapping objects directly next to each other. This position makes it easier to directly compare individual values.

```
surveys_complete %>%
  ggplot(mapping = aes(x = species_id, fill = sex)) +
  geom_bar(position = "dodge")
```



Plotting Time-series Data

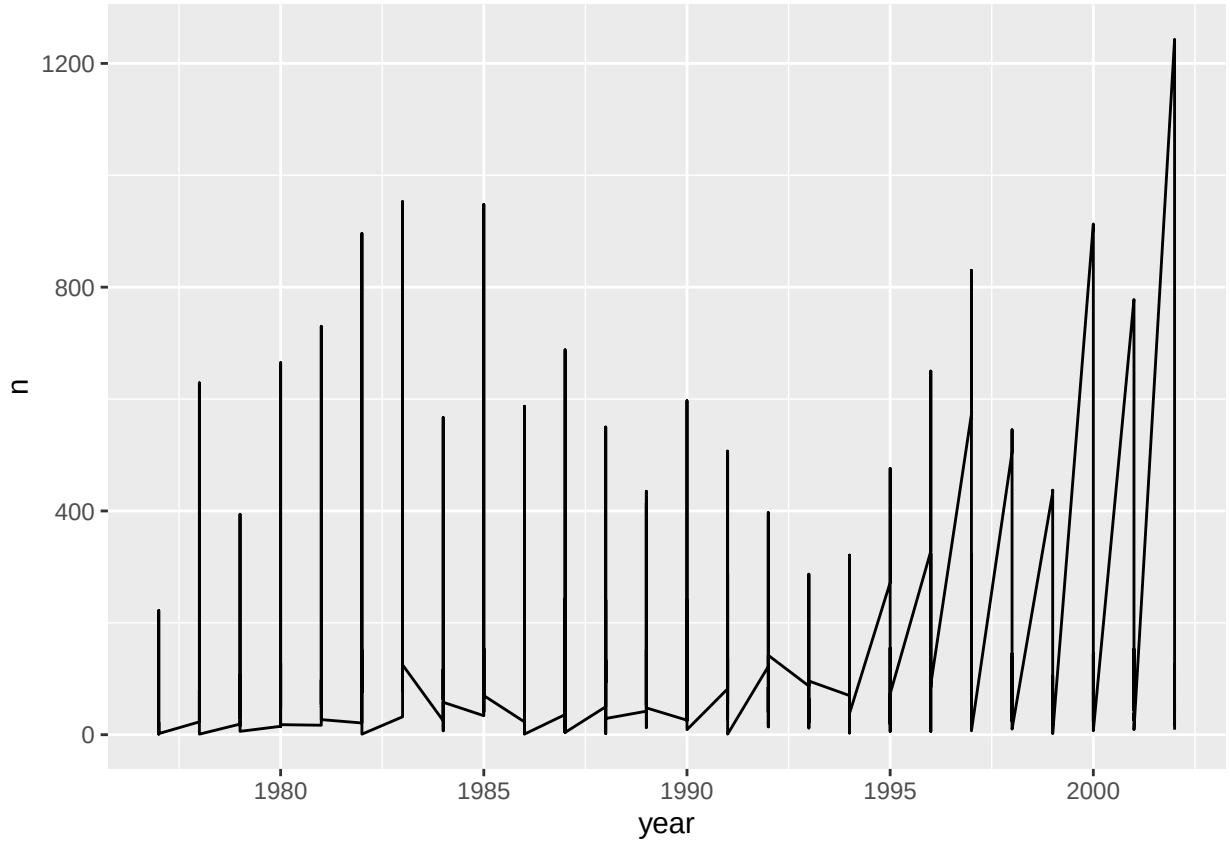
Let's calculate number of counts per year for each genus.

Remember from Data Wrangling: First we need to group the data and count records within each group!

```
yearly_counts <- surveys_complete %>%
  count(year, genus)
 ## counts the number of observations (rows) for each year, genus combination
```

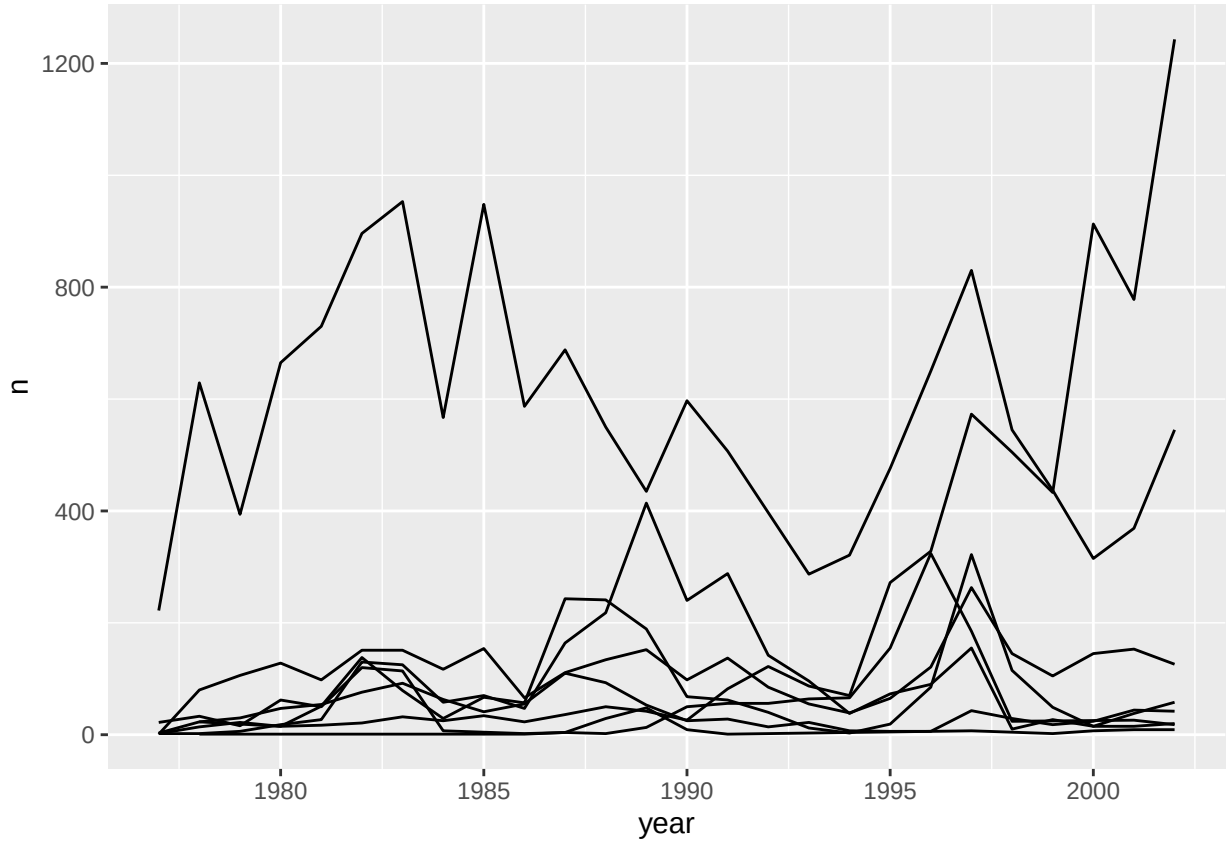
Time series data can be visualized as a line plot with years on the x-axis and counts on the y-axis:

```
yearly_counts %>%
  ggplot(mapping = aes(x = year, y = n)) +
     geom_line()
```



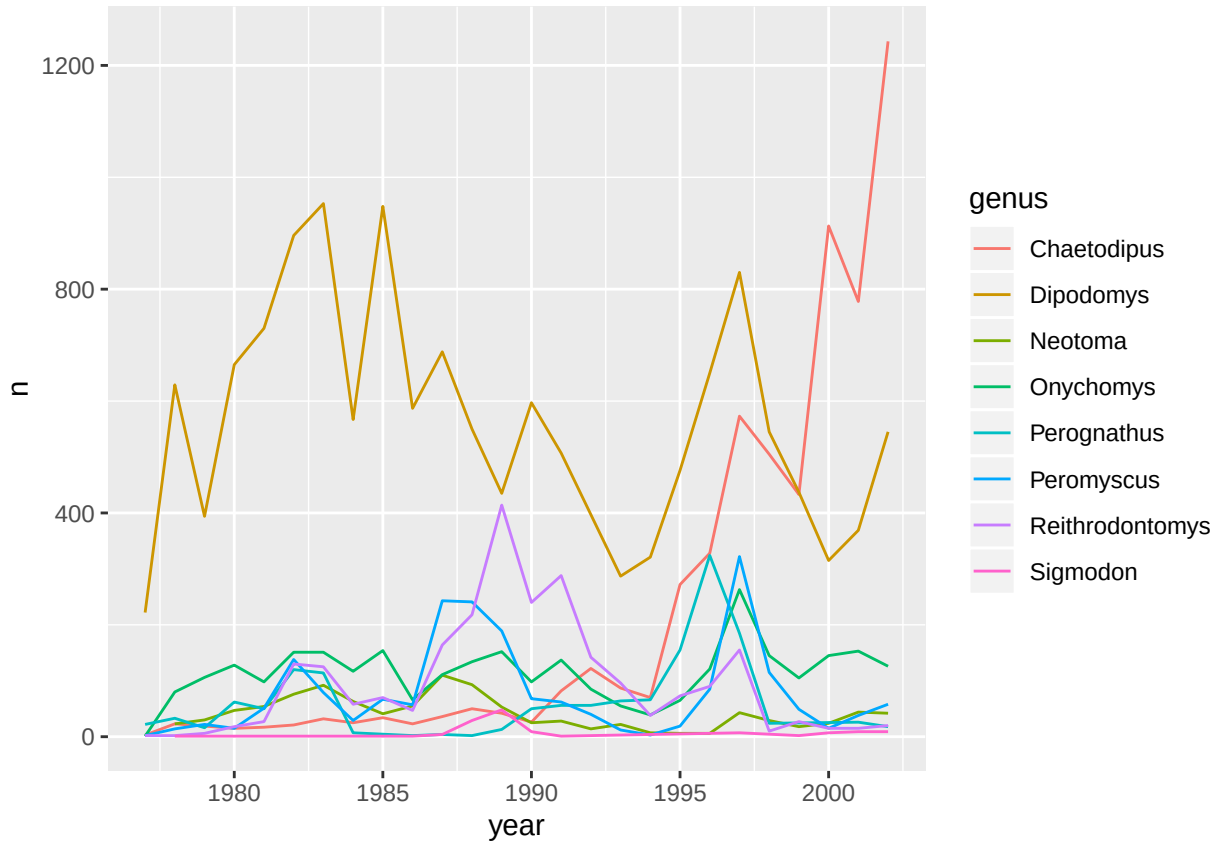
Unfortunately, this does not work because we plotted data for all the genera together. We need to tell ggplot to draw a line for each genus by modifying the aesthetic function to include `group = genus`:

```
yearly_counts %>%
  ggplot(mapping = aes(x = year, y = n, group = genus)) +
    geom_line()
```



Unfortunately, we can't tell what line corresponds to which genus. We will be able to distinguish genera in the plot if we add colors (using `color` also automatically groups the data):

```
yearly_counts %>%
ggplot(mapping = aes(x = year, y = n, color = 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 categorical variable included in the dataset.

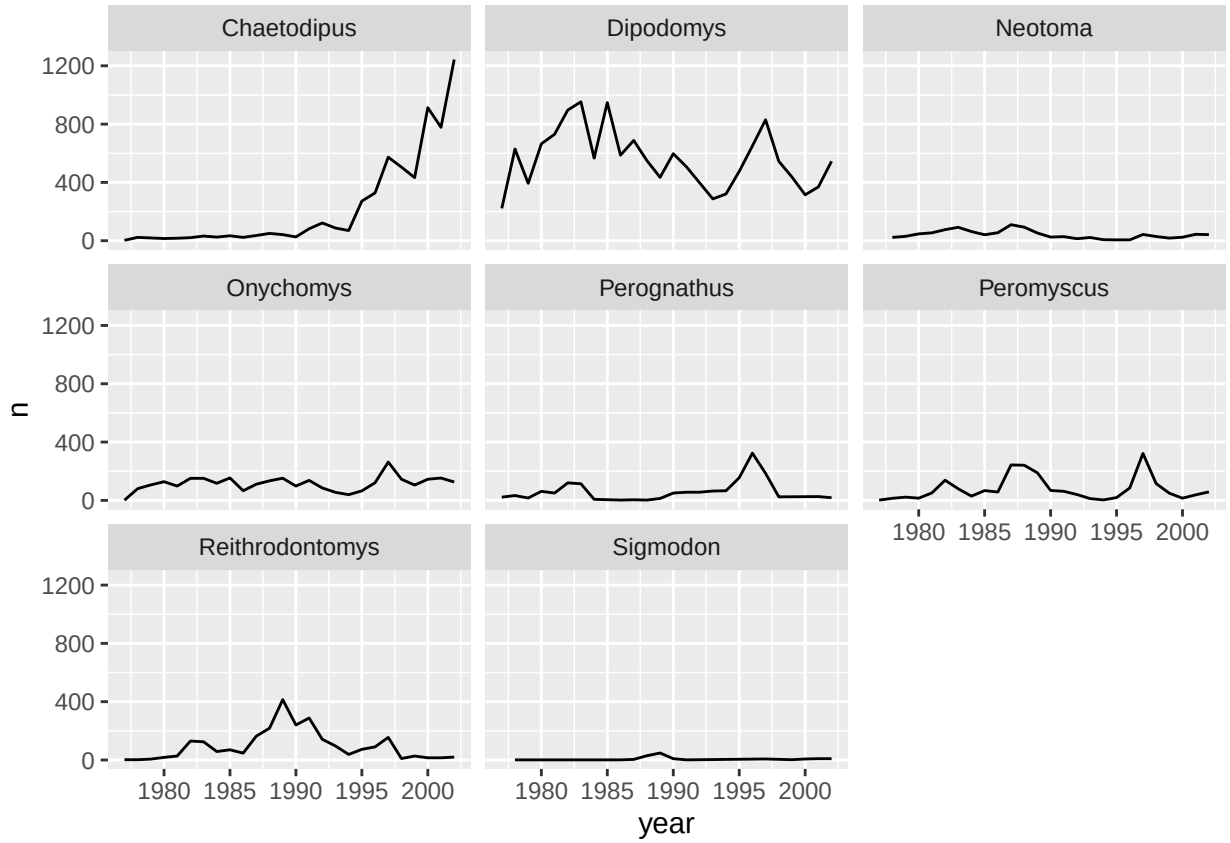
There are two types of `facet` functions:

- `facet_wrap()` arranges a one-dimensional sequence of panels to allow them to cleanly fit on one page – used for one variable
- `facet_grid()` allows you to form a matrix of rows and columns of panels – used for two variables

Both geometries allow to to specify faceting variables specified within the `vars()` function. 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:

```
yearly_counts %>%
  ggplot(mapping = aes(x = year, y = n)) +
    geom_line() +
    facet_wrap(facets = vars(genus))
```

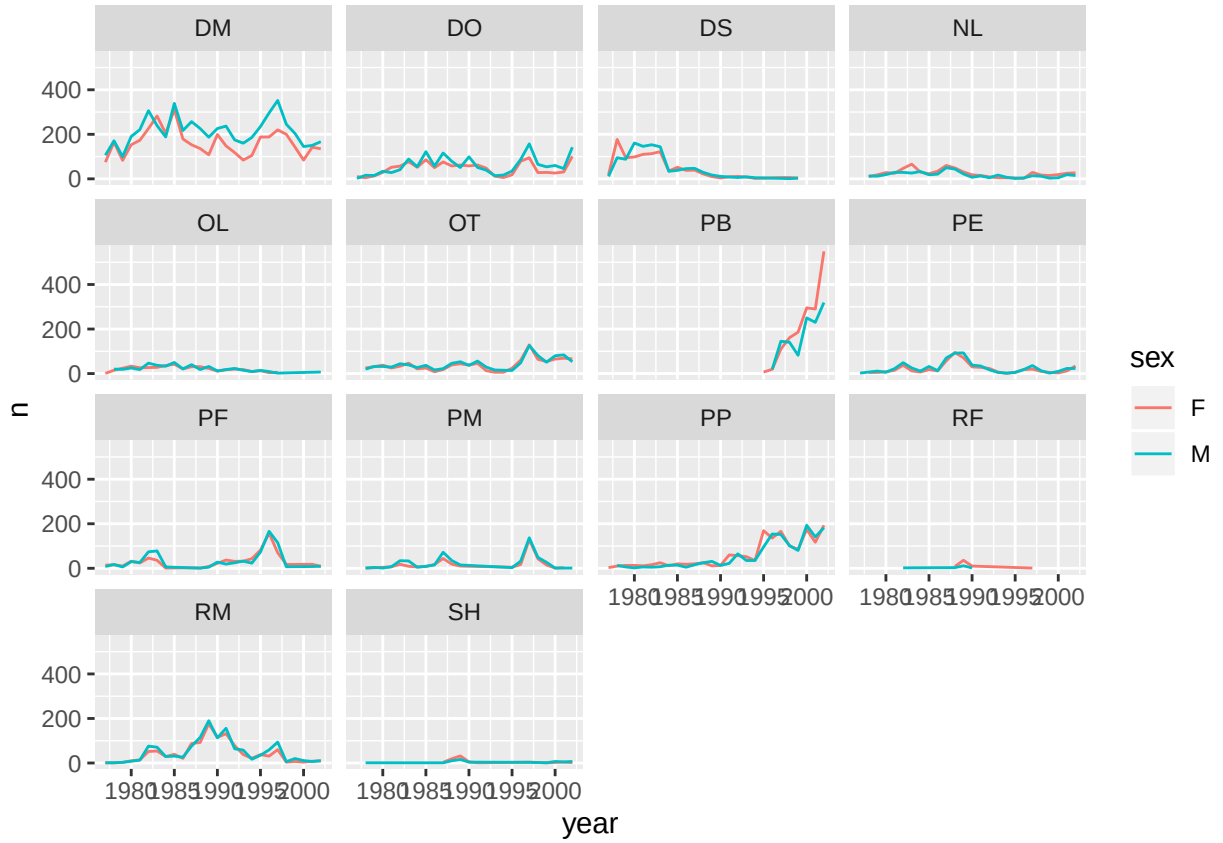


Now we would like to split the line in each plot by the sex of the rodent caputured. To do that we need to make counts in the data frame grouped by `year`, `species_id`, and `sex`:

```
yearly_sex_counts <- surveys_complete %>%
  count(year, species_id, sex)
```

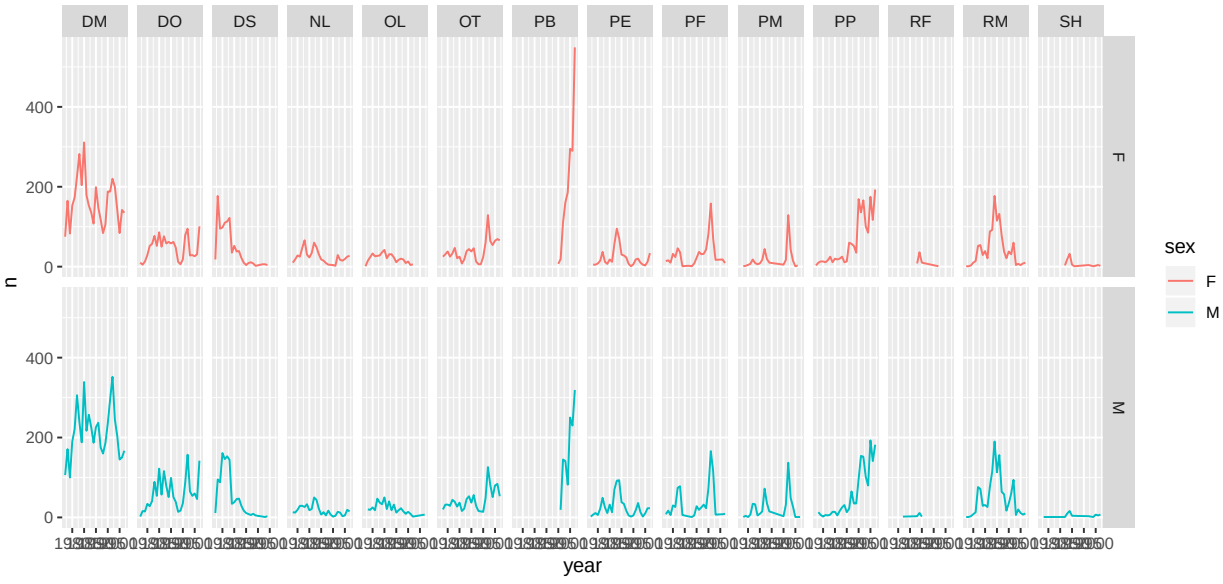
We can now make the faceted plot by splitting further by sex using `color` (within each panel):

```
yearly_sex_counts %>%
  ggplot(mapping = aes(x = year, y = n, color = sex)) +
    geom_line() +
    facet_wrap(facets =  vars(species_id))
```



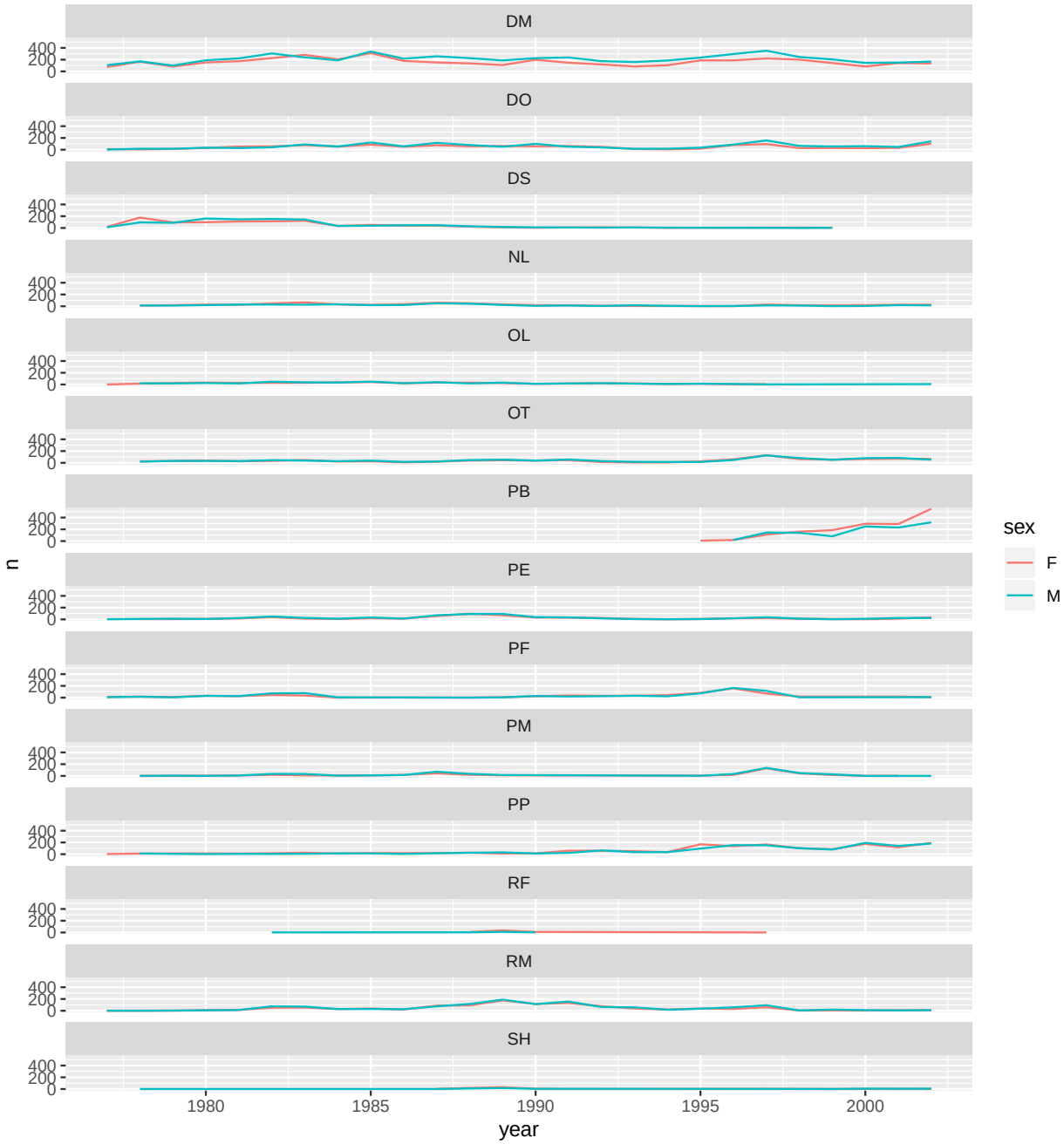
Now let's use `facet_grid()` to control how panels are organised by both rows and columns:

```
yearly_sex_counts %>%
  ggplot(mapping = aes(x = year, y = n, color = sex)) +
    geom_line() +
    facet_grid(rows = vars(sex), cols =  vars(species_id))
```



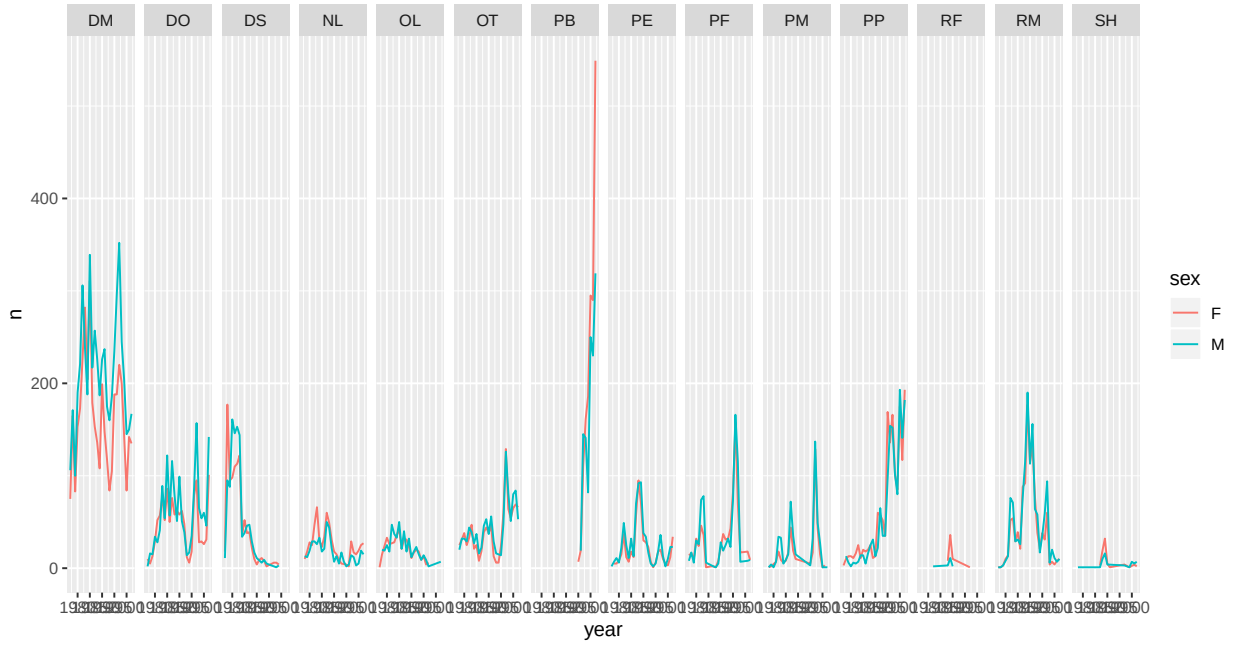
You can also organize the panels only by rows (or only by columns):

```
yearly_sex_counts %>%
  ggplot(mapping = aes(x = year, y = n, color = sex)) +
  geom_line() +
  facet_wrap(vars(species_id), ncol = 1)
```




```
  # One column, facet by rows
```

```
yearly_sex_counts %>%
  ggplot(mapping = aes(x = year, y = n, color = sex)) +
    geom_line() +
    facet_wrap(vars(species_id), nrow = 1)
```




```
  # One row, facet by columns
```

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
```

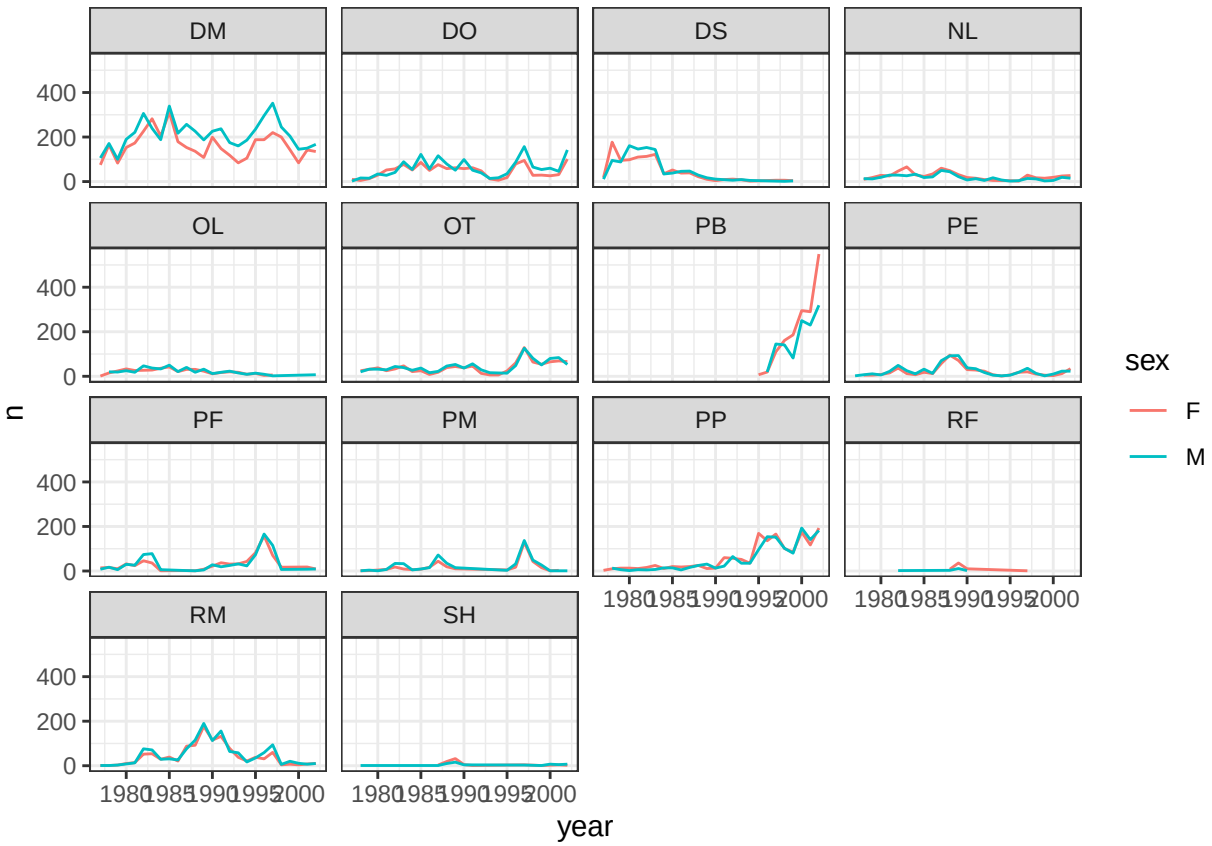
ggplot2 Themes

Usually plots with white background look more readable when printed. Every single component of a `ggplot` graph can be customized 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:

```
yearly_sex_counts %>%
 ggplot(mapping = aes(x = year, y = n, color = sex)) +
     geom_line() +
```

```
    facet_wrap(vars(species_id)) +
    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.

Challenge

> Use what you just learned to create a plot that depicts how the average weight of each species changes through the years. Play around with which variable you facet by versus plot by! Use the plotting background theme of your choosing!

```
## To get you started:
yearly_weight <- surveys_complete %>%
  group_by(year, species_id) %>%
  ## Variables to group by!
  summarize(avg_weight = mean(weight))
```

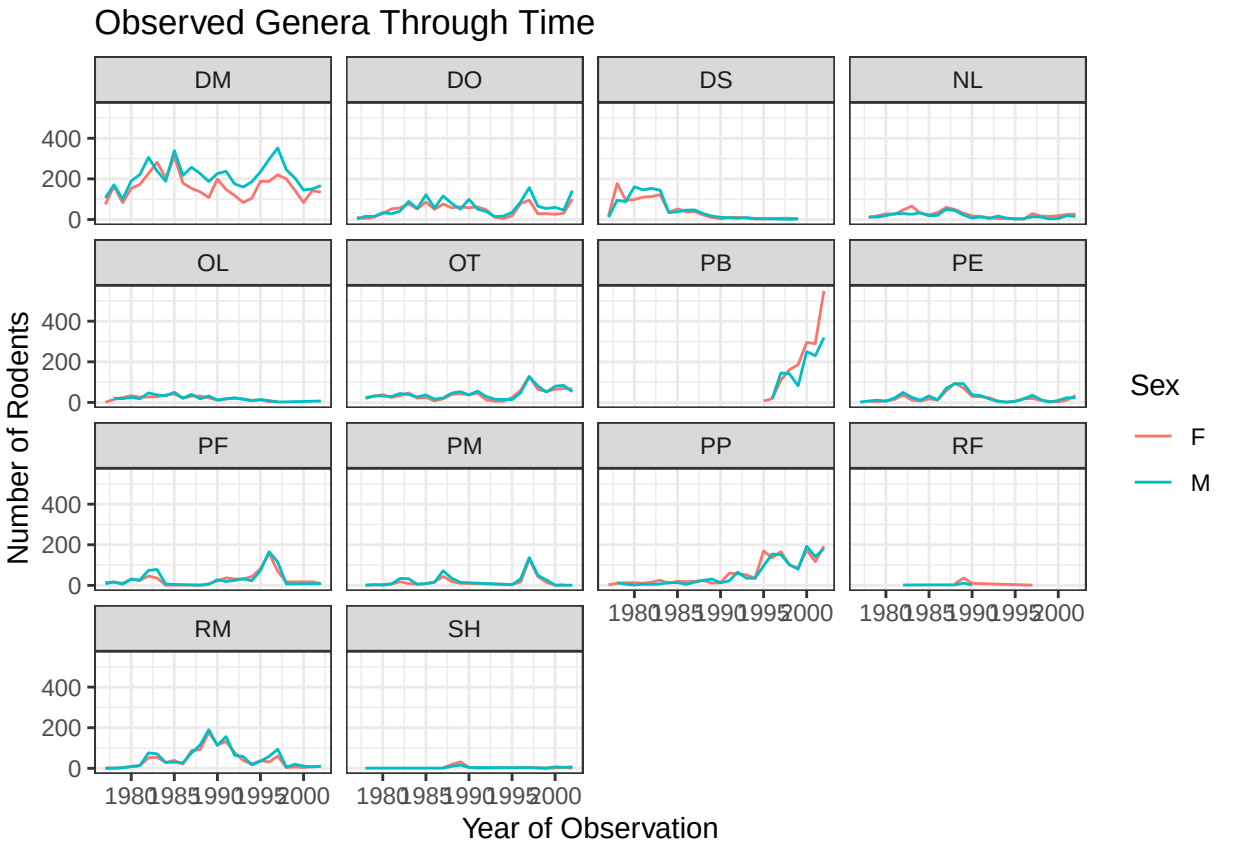
Customization

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

Plot Labels

Now, let's change names of axes to something more informative than 'year' and 'n' and add a title to the figure. Label customizations are done using the `labs()` function like so:

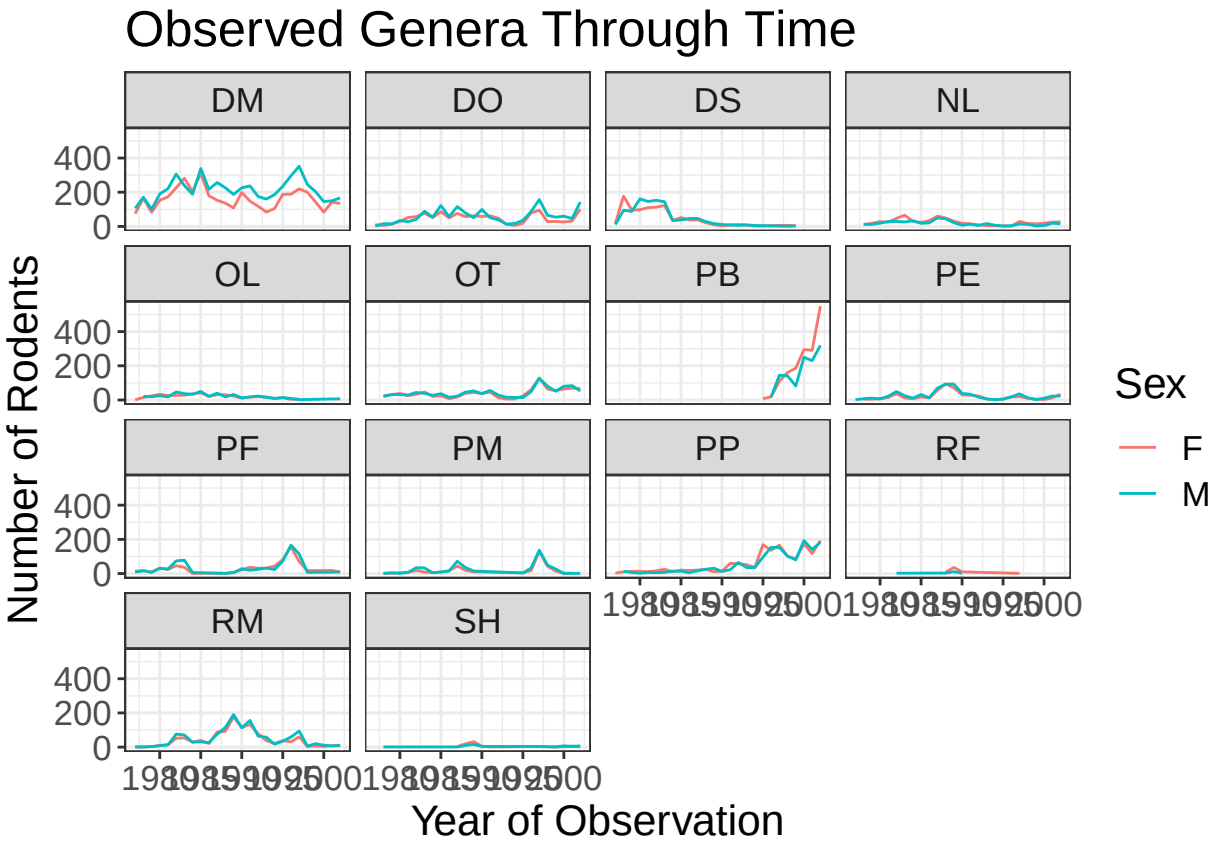
```
yearly_sex_counts %>%
  ggplot(mapping = aes(x = year, y = n, color = sex)) +
    geom_line() +
    facet_wrap(vars(species_id)) +
    labs(title = "Observed Genera Through Time",
         x = "Year of Observation",
         y = "Number of Rodents",
         color = "Sex") +
    theme_bw()
```



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:

```
yearly_sex_counts %>%
  ggplot(mapping = aes(x = year, y = n, color = sex)) +
    geom_line() +
    facet_wrap(vars(species_id)) +
```

```
    labs(title = "Observed Genera Through Time",
        x = "Year of Observation",
        y = "Number of Rodents",
        color = "Sex") +
    theme_bw() +
    theme(text = element_text(size = 16))
```




```
    ## sets ALL the text on the plot to be size 16
```

Tip: Wrapping Titles
Sometimes the titles we wish to have for our plots are longer than the space originally allotted. If you create a title and the text is running off the plot you can add a `\n` inside your title to force a line break (`\n` stands for new line).

Label & Plot Fonts

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 using the `hjust` and `vjust` options in the text theme.

The value of `hjust` and `vjust` are defined between 0 and 1.

`hjust` controls horizontal justification:

- 0 means right-justified (text on right)
- 1 means left-justified (text on left)

`vjust` controls vertical justification:

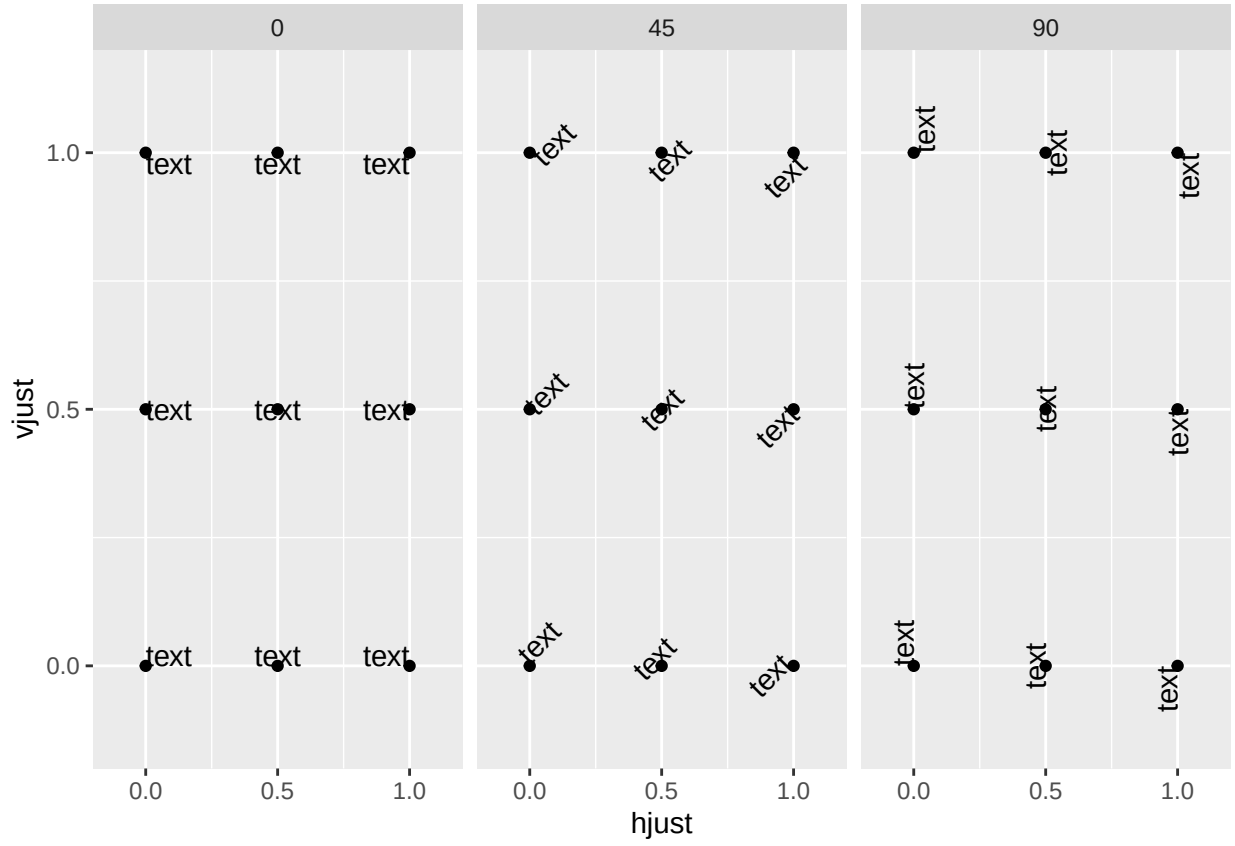
- 0 means top-justified (text above)
- 1 means bottom-justified (text below)

`angle` controls the angle of the text with the x-axis
- The angle is defined from x-axis, such that:

```
* angles from 0-90 rotate the text counter-clockwise
* negative angles rotate clockwise (from 0 to -90)
```

An example should make this more clear:

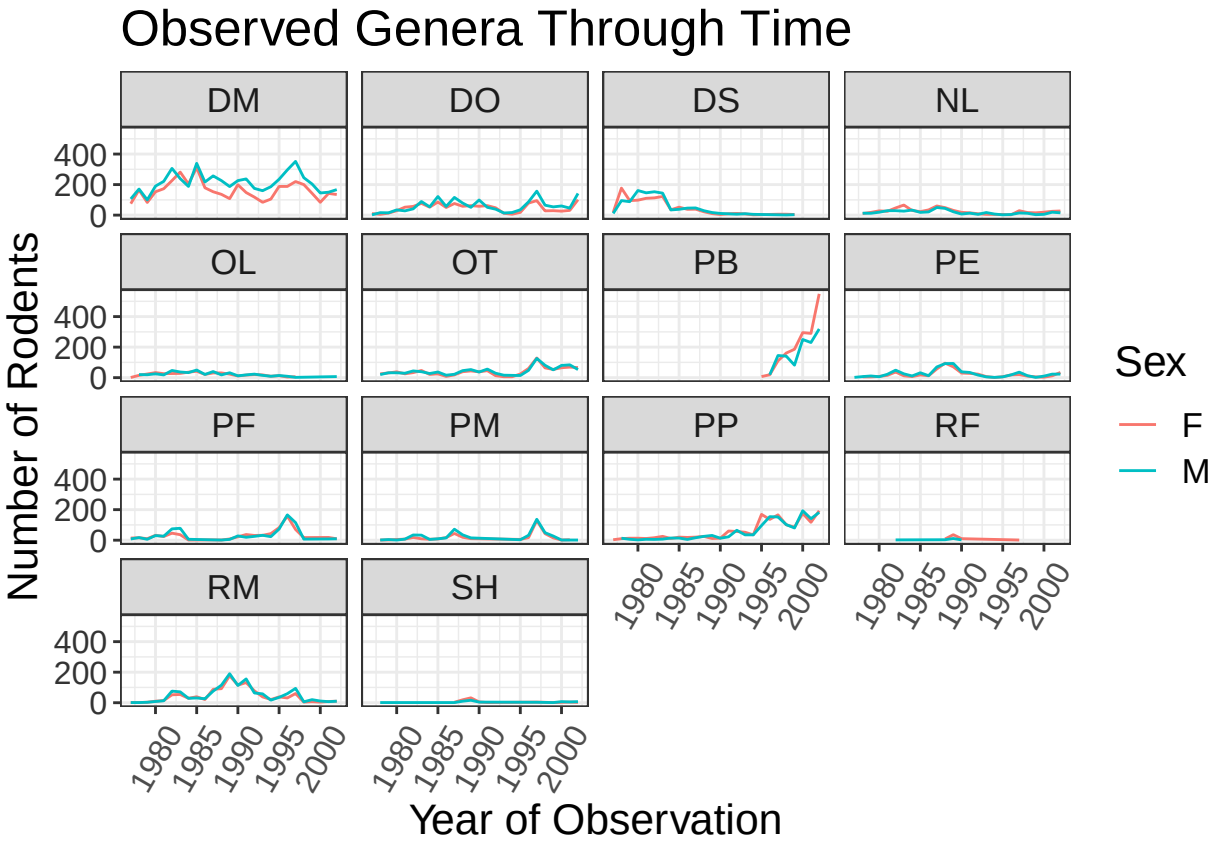
```
td <- expand.grid(
    hjust = c(0, 0.5, 1),
    vjust = c(0, 0.5, 1),
    angle = c(0, 45, 90),
    text = "text"
)

ggplot(td, aes(x = hjust, y = vjust)) +
    geom_point() +
    geom_text(aes(label = text, angle = angle, hjust = hjust, vjust = vjust)) +
    facet_wrap(vars(angle), nrow = 1) +
    scale_x_continuous(breaks = c(0, 0.5, 1), expand = c(0, 0.2)) +
    scale_y_continuous(breaks = c(0, 0.5, 1), expand = c(0, 0.2))
```



This should give you a rough idea of how you can combine `hjust`, `vjust`, and `angle` together in your text specification to align your axis labels to not overlap!

Let's give it a try.

```
yearly_sex_counts %>%
  ggplot(mapping = aes(x = year, y = n, color = sex)) +
    geom_line() +
    facet_wrap(vars(species_id)) +
    labs(title = "Observed Genera Through Time",
        x = "Year of Observation",
        y = "Number of Rodents",
        color = "Sex") +
    theme_bw() +
    theme(axis.text.x = element_text(size = 12, angle = 60,
                                     hjust = 0.5, vjust = 0.5),
          ## theme specific to x-axis text
          axis.text.y = element_text(colour = "grey20", size = 12),
          ## theme specific to y-axis text
          text = element_text(size = 16))
```

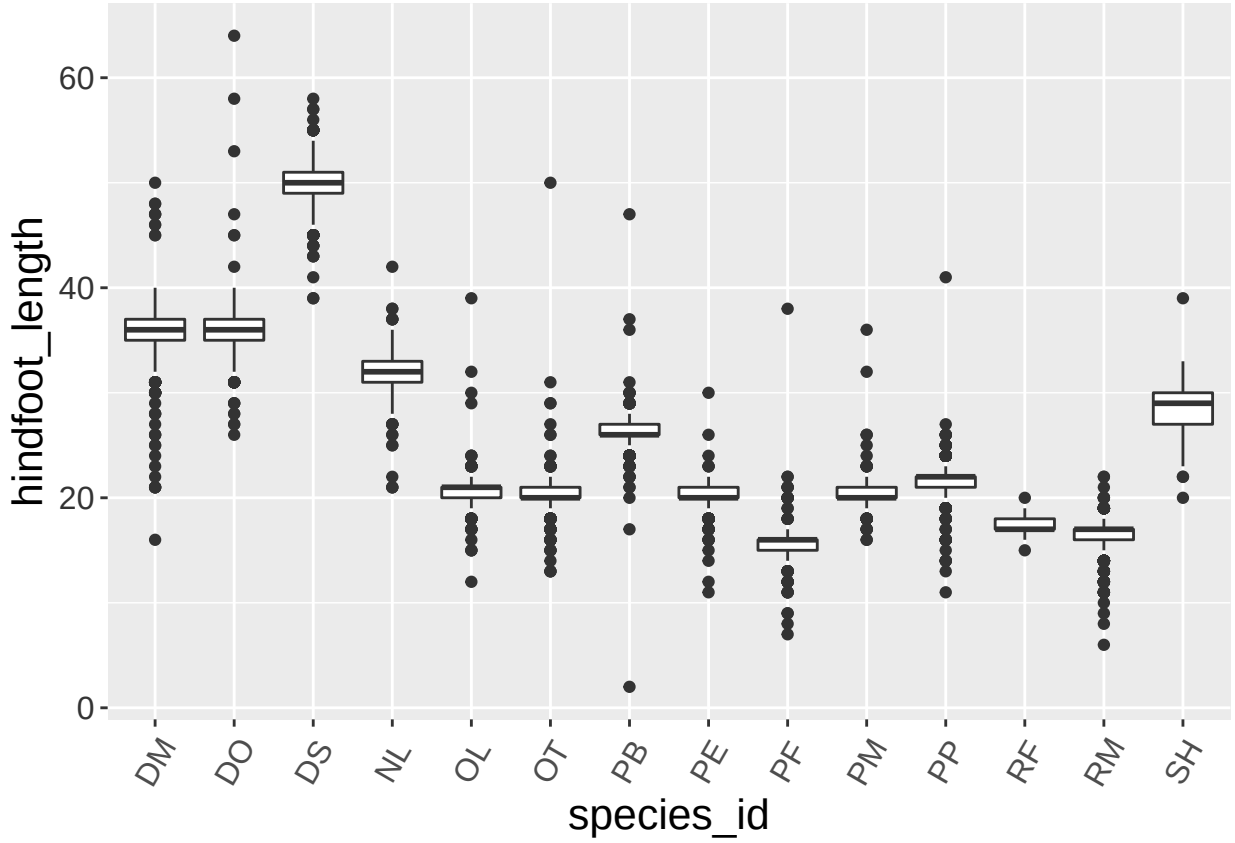



```
          ## theme for all other text on plot (title, legend, facets, etc.)
```

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 and store in an object to be added to ggplots
grey_theme <- theme(axis.text.x = element_text(size = 12, angle = 60,
```

```
                                                hjust = 0.5, vjust = 0.5),
                    axis.text.y = element_text(colour = "grey20", size = 12),
                    text = element_text(size = 16))

# create a boxplot and adds theme object to the plot
surveys_complete %>%
  ggplot(aes(x = species_id, y = hindfoot_length)) +
    geom_boxplot() +
    grey_theme
```



Changing Colors

The built in `ggplot` color scheme may not be what you were looking for, but don't worry! There are many other color palettes available to use!

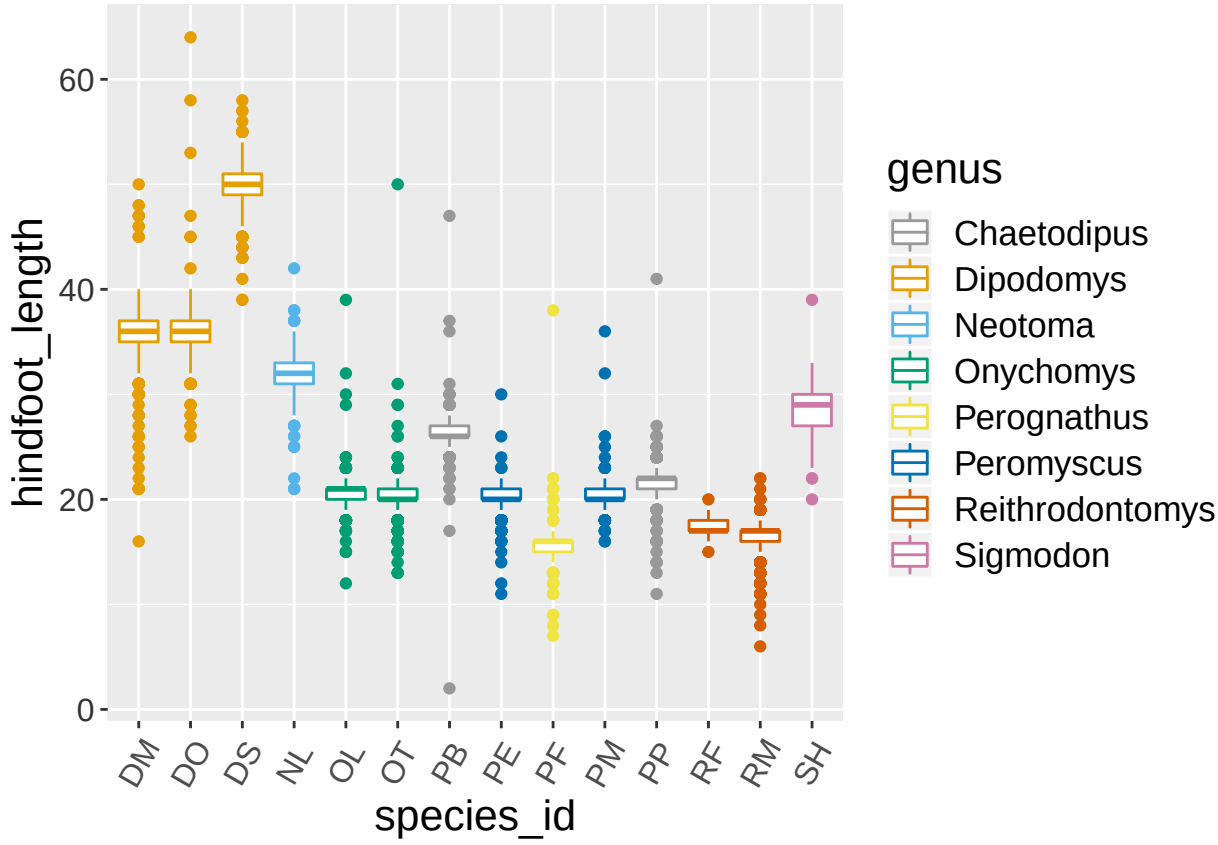
You can change the colors used by `ggplot()` a few different ways.

1. Add the `scale_color_manual()` or `scale_fill_manual()` functions to your plot and directly specify the colors you want to use, by:

- defining a vector of colors right there (e.g. c("blue", "black", "red", "green"))
- creating a vector of colors and storing it in an object and calling on it (see below)

```
# The palette with grey:
cbPalette_grey <- c("#999999", "#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55E00", "#CC79
```

```
# The palette with black:
cbbPalette_blk <- c("#000000", "#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55E00", "#CC79

surveys_complete %>%
  ggplot(aes(x = species_id, y = hindfoot_length, color = genus)) +
  geom_boxplot() +
  grey_theme +
  scale_color_manual(values = cbPalette_grey)
```



2. Install a package and use it's available color scales. Popular options include:

- `RColorBrewer`: using `scale_fill_brewer()` or `scale_colour_brewer()`
- `viridis`: using `scale_colour_viridis_d()` for discrete data, `scale_colour_viridis_c()` for continuous data, with an inside argument of `option = <COLOR>` for your chosen color scheme
- `ggsci`: using `scale_color_<PALNAME>()` or `scale_fill_<PALNAME>()`, where you specify the name of the palette you wish to use (e.g. `scale_color_aaas()`)

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 color palette (see http://www.cookbook-r.com/Graphs/Colors_(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 **gridExtra** package allows us to combine separate ggplots into a single figure using `grid.arrange()`:

```
library(gridExtra)

spp_weight_boxplot <- surveys_complete %>%
  ggplot(aes(x = genus, y = weight)) +
  geom_violin() +
  geom_point(color = "tomato") +
  scale_y_log10() +
  labs(x = "Genus", y = "Weight (g)") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))

spp_count_plot <- yearly_counts %>%
  ggplot(aes(x = year, y = n, color = genus)) +
  geom_line() +
  labs(x = "Year", y = "Abundance")

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


```
## nrow and ncol specify how many rows/columns you want the arranged plots to be in
## widths specify what proportion of the overall plotting area each plot takes up
```

In addition to the `ncol` and `nrow` arguments, used to make simple arrangements, there are tools for constructing more complex layouts.

For more assistance arranging plots with `grid.arrage()` I find the following vignette **very** helpful!

https://cran.r-project.org/web/packages/egg/vignettes/Ecosystem.html

Exporting Plots

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` and `height`: adjust the total plot size in units ("in", "cm", or "mm")
  - If units are not specified, default is inches.
- `dpi`: adjusts the plot resolution. This accepts a string or numeric input:
  - "retina" (320)
  - "print" (300)
  - "screen" (72)

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

```
my_plot <- ggplot(data = yearly_sex_counts,
                  mapping = aes(x = year, y = n, color = sex)) +
              geom_line() +
              facet_wrap(vars(species_id)) +
              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)

# This also works for grid.arrange() plots
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)
```

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

Final 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: https://www.rstudio.com/wp-content/uploads/2015/08/ggplot2-cheatsheet.pdf

Workshop build on: 2019-11-12 13:03:32