Data visualization

Data Science Workshop - Session 2

Data Carpentry contributors & Montana State University R Workshops Team

---

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.
- Export publication ready graphics using ggsave().

Data Viz Introduction

`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.

Packages in `R` are basically sets of additional functions that let you do more stuff. The functions we've used in the previous session, like `str()` or `mean()`, come built into `R`; packages give you access to more of them. Before you use a package for the first time you need to install it on your machine, and then you should import it in every subsequent `R` session when you need it.

> Install the **tidyverse** package by going to `Packages > Install` and typing tidyverse into the dialog box. Keep "Install dependencies" checked. You can also run `install.packages("tidyverse")` from the console.

This is an "umbrella-package" that installs several packages useful for data analysis which work together well such as **`tidyr`**, **`dplyr`**, **`ggplot2`**, **`readr`**, **`forcats`**, etc.

The **`tidyverse`** package tries to address common issues that arise when doing data analysis with some of the functions that come with `R`.

1. The `tidyverse` solves complex problems by combining many simple pieces.

   "No matter how complex and polished the individual operations are, it is often the quality of the glue that most directly determines the power of the system."

   — Hal Abelson

2. The `tidyverse` is written for people to read!

   "Computer efficiency is a secondary concern because the bottleneck in most data analysis is thinking time, not computing time."

   — Hadley Wickham

In this workshop, we have already installed the `tidyverse` using `install.packages("tidyverse")`. It is important to note that there's no need to re-install packages every time we run the script.

Then, to load the package include code in your work with:

```r
## load the tidyverse packages
library(tidyverse)
```

Working with packages was discussed in more detail in the "Introduction to R" workshop. We will proceed through the remaining work with the `tidyverse` package installed and loaded.

To learn more about **`ggplot2`** after the workshop, you may want to check out this **`ggplot2`** reference website (link) and this handy cheatsheet on **`ggplot2`** (link).

Presentation of the Survey Data

The data used in this workshop are a time-series for a small mammal community in southern Arizona. This is part of a project studying the effects of rodents and ants on the plant community that has been running for almost 40 years, but we will focus on the years 1996 to 2002 (n=11332 observations). The rodents are sampled on a series of 24 plots, with different experimental manipulations controlling which rodents are allowed to access which plots. This is simplified version of the full data set that has been used in over 100 publications and was provided by the Data Carpentries (https://datacarpentry.org/ecology-workshop/data/). We are going to focus on animal species diversity and weights in this workshop. The dataset is stored as a comma separated value (CSV) file.

Each row holds information for a single animal, and the columns represent (along with some others we will not use):

Column	Description
record_id	Unique id for the observation
month	month of observation
day	day of observation
year	year of observation
plot_id	ID of a particular plot
species_id	2-letter code
sex	sex of animal ("M", "F")
hindfoot_length	length of the hindfoot in mm
weight	weight of the animal in grams

We'll read in our data using the `read_csv()` function, from the tidyverse package **`readr`**, instead of `read.csv()`.

```r
surveys <- read_csv("data/surveys2.csv")
```

```
## Rows: 11332 Columns: 15
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (7): species_id, sex, day_of_week, plot_type, genus, species, taxa
## dbl  (7): record_id, month, day, year, plot_id, hindfoot_length, weight
## date (1): date
##
## 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.
```

You will see the message `Parsed with column specification`, followed by each column name and its data type. When you execute `read_csv` on a data file, it looks through the first 1000 rows of each column and guesses the data type for each column as it reads it into `R`. For example, in this dataset, `read_csv` reads `weight` as `col_double` (a numeric data type), and `species` as `col_character`.

```r
## inspect the data
str(surveys)
```

```
## spc_tbl_ [11,332 x 15] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ record_id      : num [1:11332] 23215 23216 23217 23218 23220 ...
##  $ month          : num [1:11332] 1 1 1 1 1 1 1 1 1 1 ...
##  $ day            : num [1:11332] 27 27 27 27 27 27 27 27 27 27 ...
##  $ year           : num [1:11332] 1996 1996 1996 1996 1996 ...
##  $ plot_id        : num [1:11332] 21 1 17 17 2 18 1 2 17 2 ...
##  $ species_id     : chr [1:11332] "PF" "DM" "DM" "DM" ...
##  $ sex            : chr [1:11332] "F" "M" "M" "M" ...
```

```
##  $ hindfoot_length: num [1:11332] 16 36 36 37 36 16 34 37 39 40 ...
##  $ weight         : num [1:11332] 7 27 25 25 47 9 27 66 49 54 ...
##  $ date           : Date[1:11332], format: "1996-01-27" "1996-01-27" ...
##  $ day_of_week    : chr [1:11332] "Sat" "Sat" "Sat" "Sat" ...
##  $ plot_type      : chr [1:11332] "Long-term Krat Exclosure" "Spectab exclosure" "Control" "Control"
##  $ genus          : chr [1:11332] "Perognathus" "Dipodomys" "Dipodomys" "Dipodomys" ...
##  $ species        : chr [1:11332] "flavus" "merriami" "merriami" "merriami" ...
##  $ taxa           : chr [1:11332] "Rodent" "Rodent" "Rodent" "Rodent" ...
##  - attr(*, "spec")=
##   .. 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()
##   .. )
##  - attr(*, "problems")=<externalptr>
```

```
## Preview the data
View(surveys)
```

```
## # A tibble: 11,332 x 15
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1     23215     1    27  1996      21 PF         F                  16      7
##  2     23216     1    27  1996       1 DM         M                  36     27
##  3     23217     1    27  1996      17 DM         M                  36     25
##  4     23218     1    27  1996      17 DM         M                  37     25
##  5     23220     1    27  1996       2 DM         F                  36     47
##  6     23221     1    27  1996      18 PF         F                  16      9
##  7     23222     1    27  1996       1 DM         M                  34     27
##  8     23223     1    27  1996       2 DO         M                  37     66
##  9     23224     1    27  1996      17 DM         F                  39     49
## 10     23225     1    27  1996       2 DM         M                  40     54
```

```
## # i 11,322 more rows
## # i 6 more variables: date <date>, day_of_week <chr>, plot_type <chr>,
## #   genus <chr>, species <chr>, taxa <chr>
```

At the top of the `str()` output, notice that the class of the data is a tibble. Tibbles tweak some of the behaviors of the data frame objects we introduced in the previous workshop. The data structure is very similar to a data frame, so for our purposes the only differences are that:

1. In addition to displaying the data type of each column under its name, it only prints the first few rows of data and only as many columns as fit on one screen.
2. Columns of class `character` are never converted into factors.

Plotting with ggplot2

`ggplot2` functions like data in the 'long' format, i.e., a column for every dimension, and a row for every observation. There are other data formats, which we will discuss in the *Data Wrangling in* **R** workshop, as well as how to convert from one data format to another. Well-structured data will save you lots of time when making figures with `ggplot2` and when working in `R`!

`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(<VARIABLE MAPPINGS>)) +  <GEOM_FUNCTION>()
```

Let's go through this step by step!

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

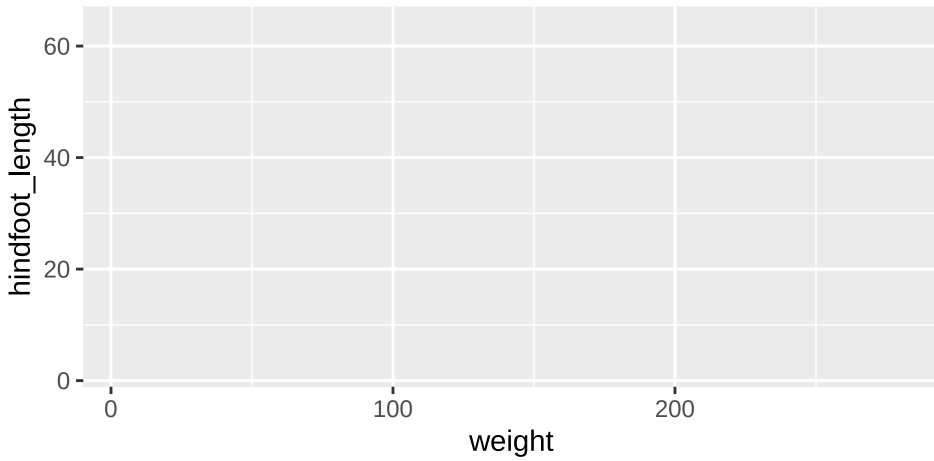
   ```r
   ggplot(data = surveys)
   ```



   ```r
   ## Creates a blank ggplot(), referencing the surveys dataset
   ```

2. 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,
       mapping = aes(x = weight, y = hindfoot_length))
```



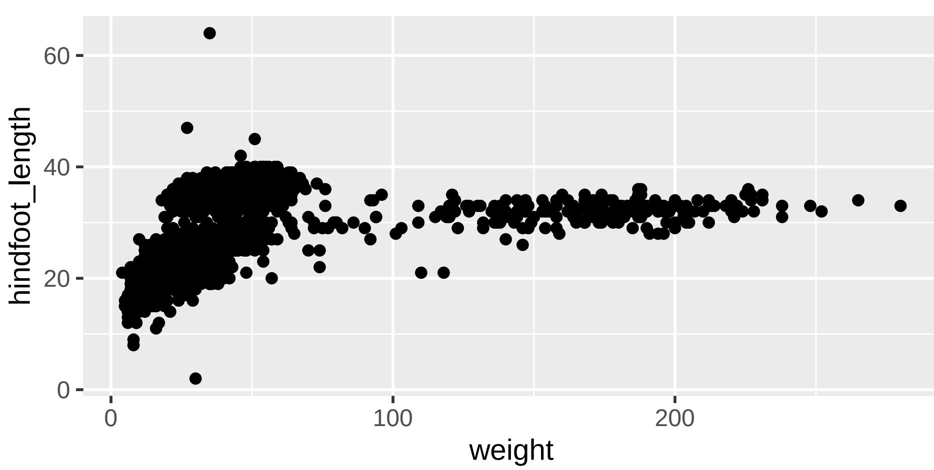

```
#
# Creates a blank ggplot(), with the variables mapped to the x- and y-axis
# ggplot() knows where the variables live, since you have defined the data to use
```

3. 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:

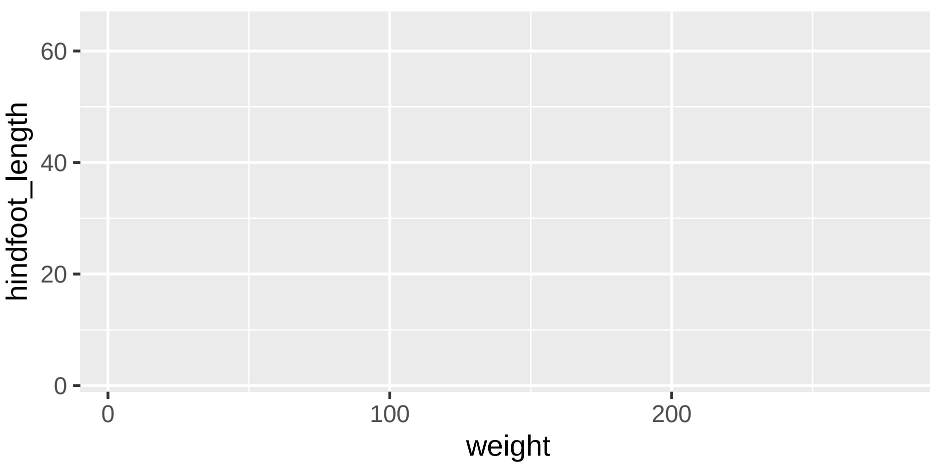
```
ggplot(data = surveys,
       mapping = aes(x = weight, y = hindfoot_length)) +
  geom_point()
```



```r
# Adds a point for each row (observation) in the data
```

You can think of the **+** sign as adding layers to the plot. Each **+** sign 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.

```r
# This will not add the new layer and will return an error message
ggplot(data = surveys,
       mapping = aes(x = weight, y = hindfoot_length))
```



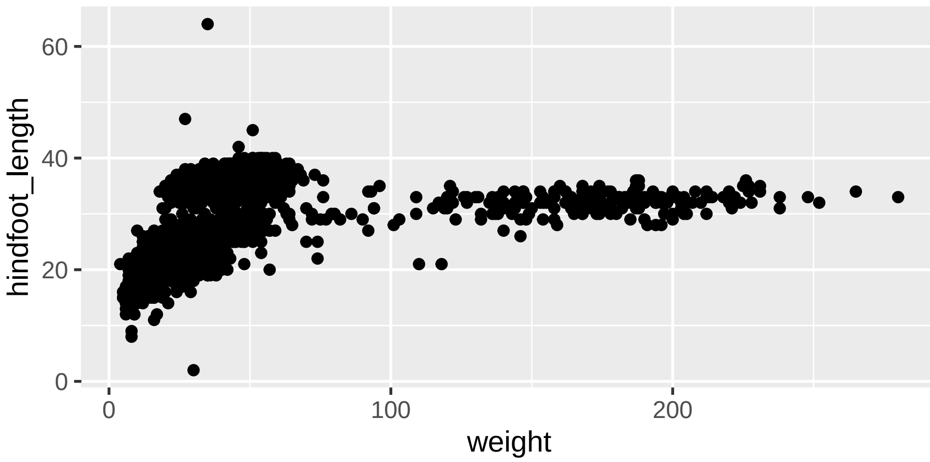

```r
  + geom_point()
```

```
## Error:
## ! Cannot use `+` with a single argument.
## i Did you accidentally put `+` on a new line?
```

Building Plots Iteratively

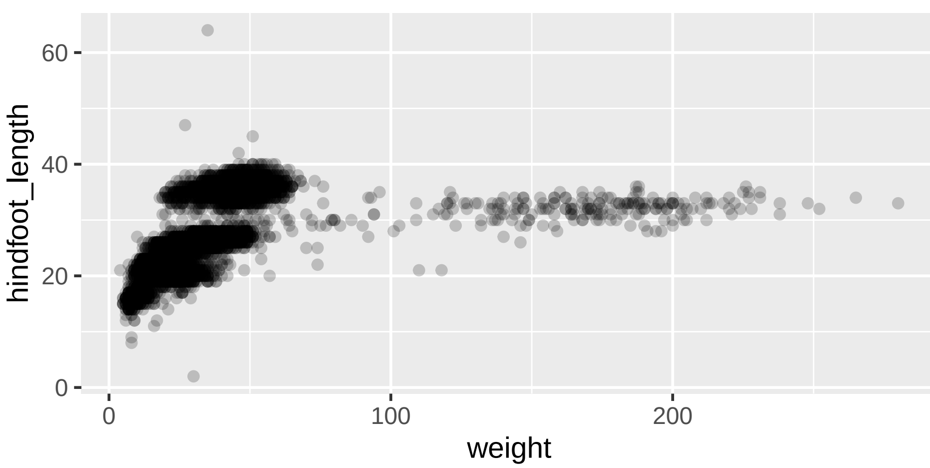
Building plots with **ggplot2** is typically an iterative process. We start by defining the dataset we'll use, lay out the axes, and choose a geom:

```r
ggplot(data = surveys,
       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 the points, to avoid overplotting:

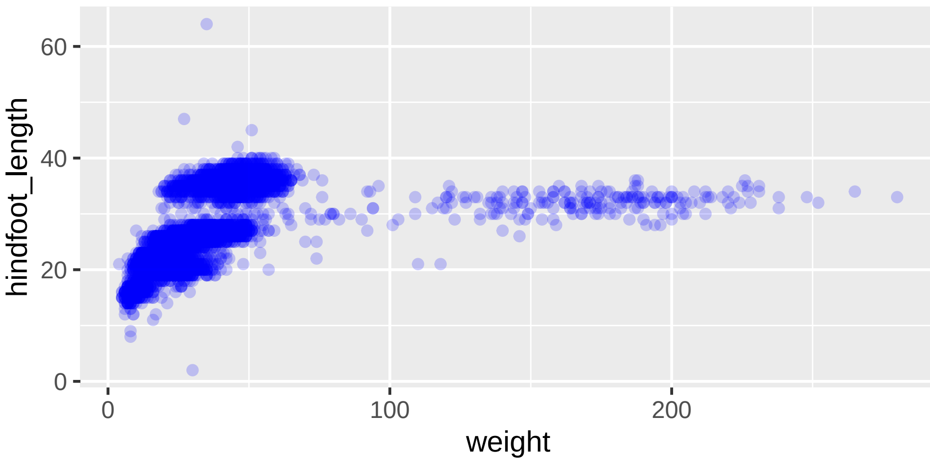
```
ggplot(data = surveys,
       mapping = aes(x = weight, y = hindfoot_length)) +
    geom_point(alpha = 0.2)
```




```
    ## 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,
       mapping = aes(x = weight, y = hindfoot_length)) +
    geom_point(alpha = 0.2, color = "blue")
```



`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

Challenge 1

> Copy and paste the code from the previous code chunk and modify it to assign one of these aesthetics to the `geom_point` aspect of your plot.
>
> What happened?

```
## Your ggplot code to answer the challenge goes here!
```

Piping 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. The pipe operator sends the output of one function directly into the next function, which is useful when you need to do many things to the same dataset. Since the dataset we wish to use is the first argument to `ggplot()`, we can use the pipe operator to pipe the data into the `ggplot()` function!

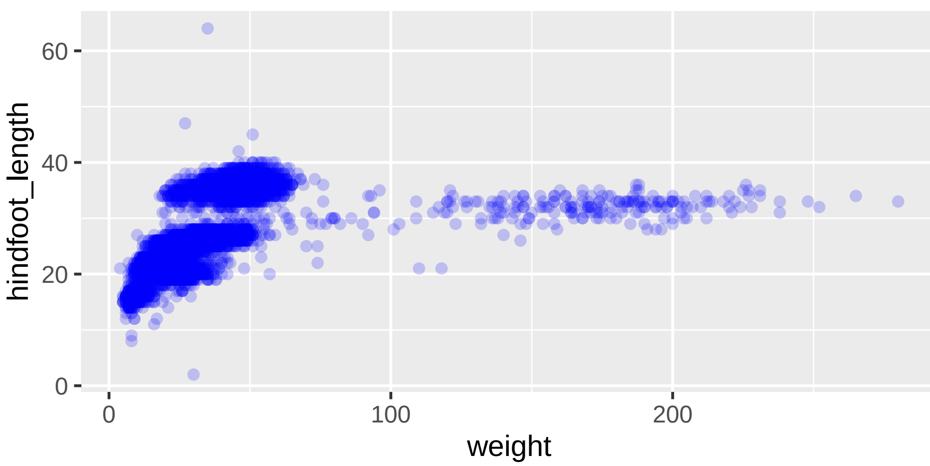
Pipes in `R` look like `%>%` and are made available via the **`magrittr`** package, installed automatically with the **`tidyverse`**. If you use RStudio, you can type the pipe with `Ctrl + Shift + M` if you have a PC or `Cmd + Shift + M` if you have a Mac.

> **Note:** There is now (as of R 4.1.0) a native R pipe `|>` that works similar to the `%>%` pipe operator with minor differences that you may encounter, but since we are working in the `tidyverse` we will stick with their pipe operator (`%>%`). If you want to switch which pipe operator is used with

> the shortcut keys, you can go to `Tools > Global Options... > Code` and check (or uncheck) the option "Use native pipe operator, |> (requires R 4.1+)".

This would instead look like this:

```r
surveys %>%
  ## data to be used in the ggplot
  ggplot(mapping = aes(x = weight, y = hindfoot_length)) +
  ## uses the data piped in as the first argument to ggplot()
    geom_point(alpha = 0.2, color = "blue")
```



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

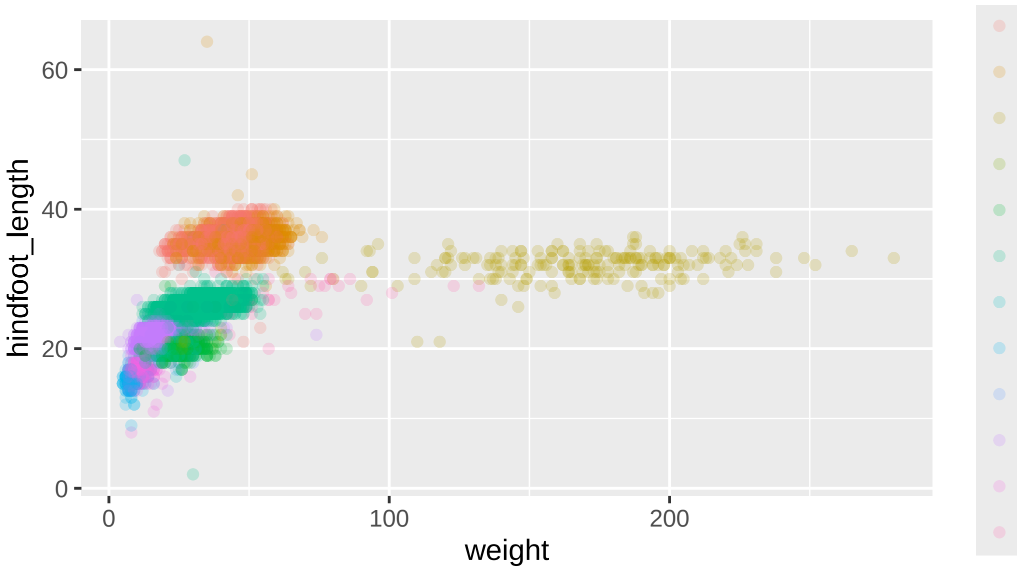
```
mapping = aes(<VARIABLES>)
```

When we pipe our data in, the first argument then becomes this `mapping` argument.

Assigning More Variables to Aesthetics

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 %>%
  ggplot(mapping = aes(x = weight, y = hindfoot_length)) +
    geom_point(alpha = 0.2, aes(color = species_id))
```



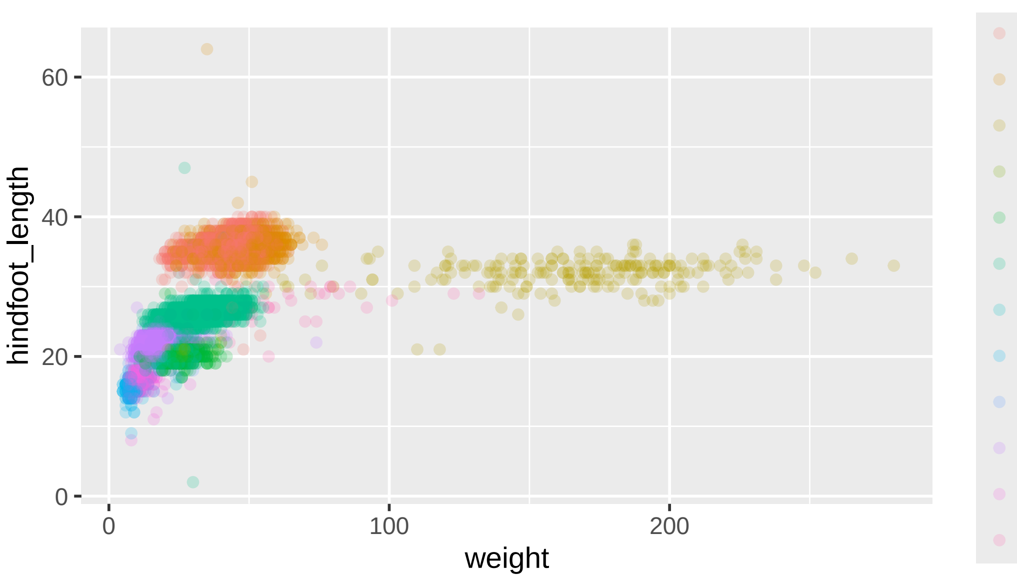
Note: When specifying an `alpha` for a scatterplot, it automatically uses that **same** `alpha` in the legend. To remedy this you can add:

```
guides(color = guide_legend(override.aes = list(alpha = 1)))
```

to your plot. This customizes the legend appearance, similar to what we will see in the customization section.

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 %>%
  ggplot(mapping = aes(x = weight, y = hindfoot_length, color = species_id)) +
    geom_point(alpha = 0.2)
```



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

Local Aesthetics versus Global Aesthetics

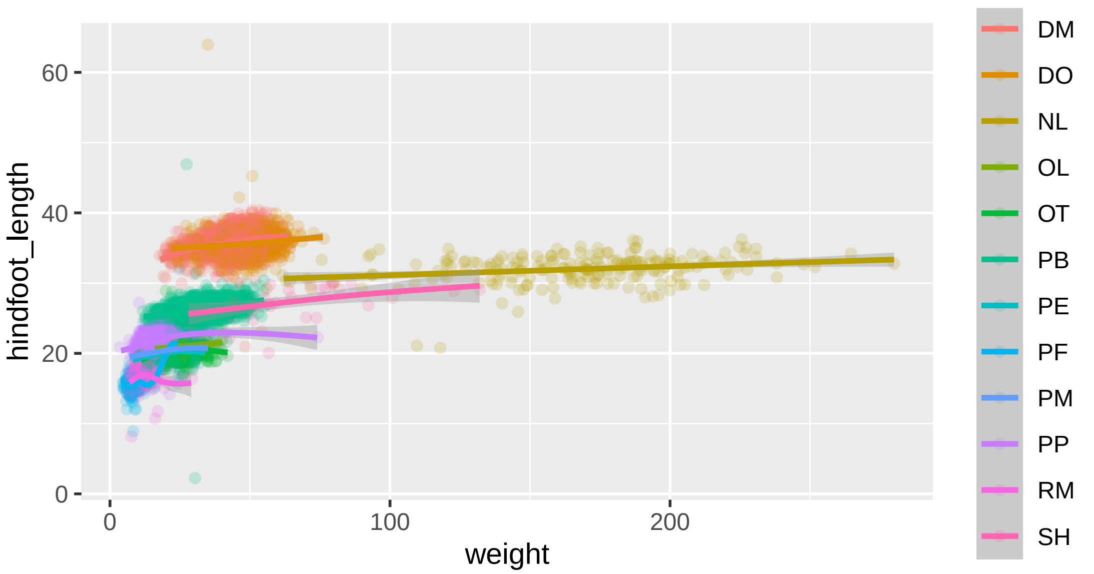
When you define aesthetics in the `ggplot()` function, those mappings hold for **every** aspect of your plot.

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

Globally

```
surveys %>%
  ggplot(mapping = aes(x = weight, y = hindfoot_length, color = species_id)) +
  geom_jitter(alpha = 0.2) +
  geom_smooth()
```

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

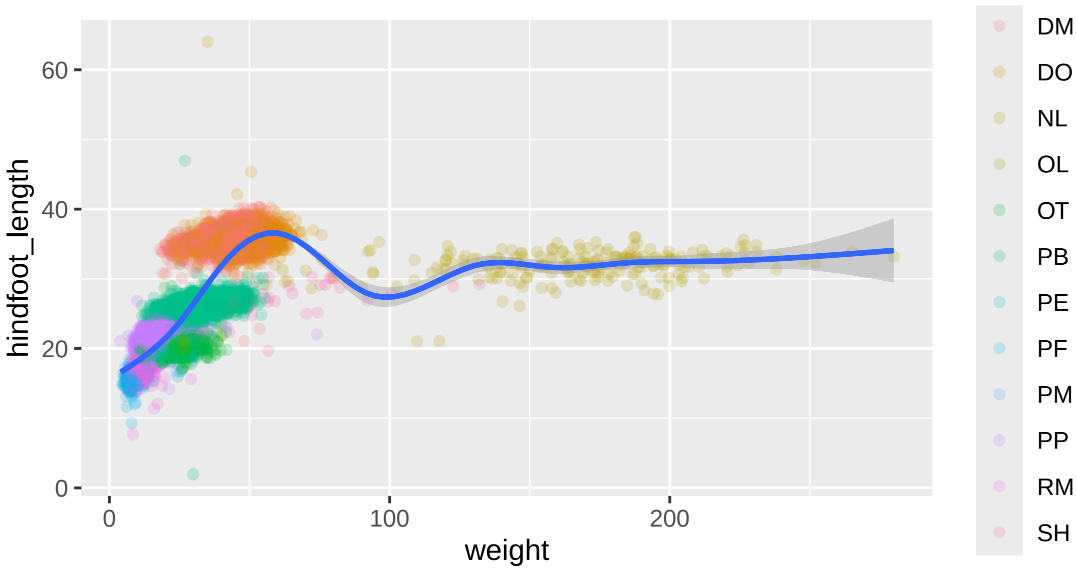



```
  ## smoothing line for each species_id -- because color is defined globally
```

Locally

```
surveys %>%
  ggplot(mapping = aes(x = weight, y = hindfoot_length)) +
  geom_jitter(aes(color = species_id), alpha = 0.2) +
  geom_smooth()
```

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




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

Challenge 2 (Part 1)

> Inspect the `geom_point` help file (either go to https://ggplot2.tidyverse.org/reference/geom_point.html or run `?geom_point`) 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?

```
## Your ggplot() code for the challenge goes here!
```

Challenge 2 (Part 2)

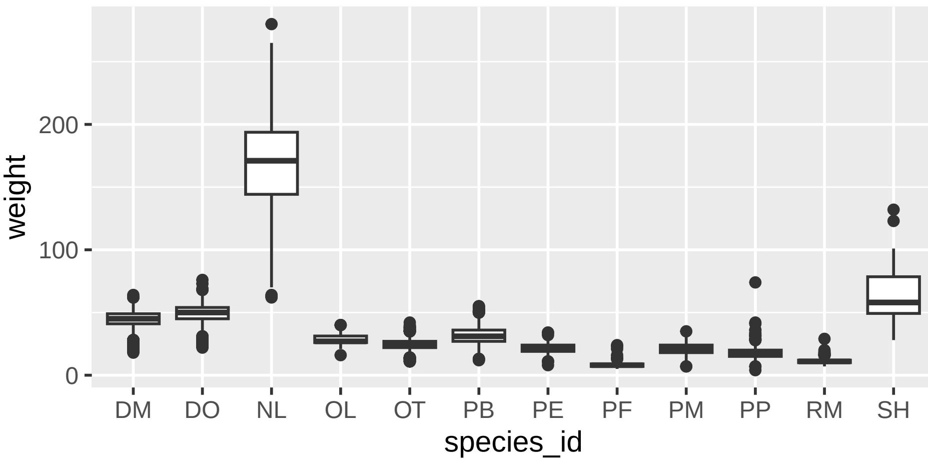
> Use what you just learned to create a scatter plot of `weight` over `plot_id` with data from different plot types being showed in different colors. Is this a good way to show this type of data?

```
## Your ggplot() code for the challenge goes here!
```

Boxplots & Violin Plots

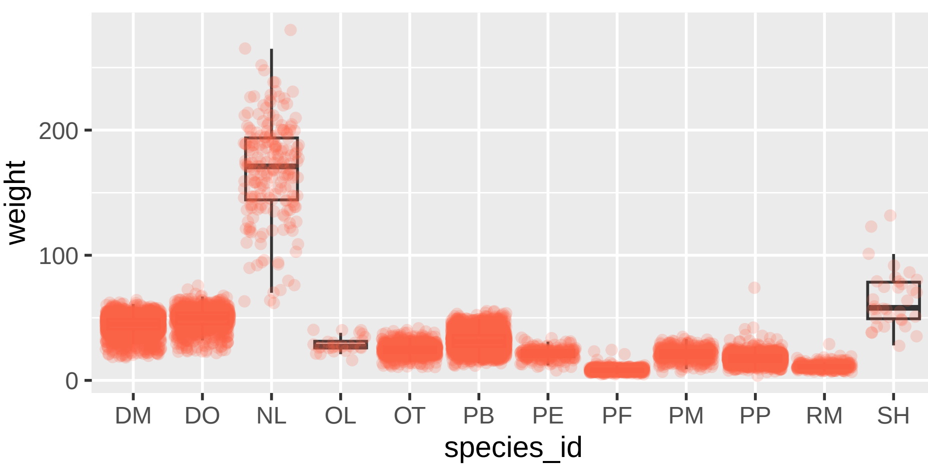
Boxplots provide a visualization of a quantitative variables across different levels of a categorical (grouping) variable. For example, we can use boxplots to visualize the distribution of weight within each species:

```
surveys %>%
  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 %>%
  ggplot(mapping = aes(x = species_id, y = weight)) +
    geom_boxplot(alpha = 0) +
    ## alpha = 0 eliminates the black (possible outlier) points, so they're not plotted twice
    geom_jitter(alpha = 0.2, color = "tomato")
```



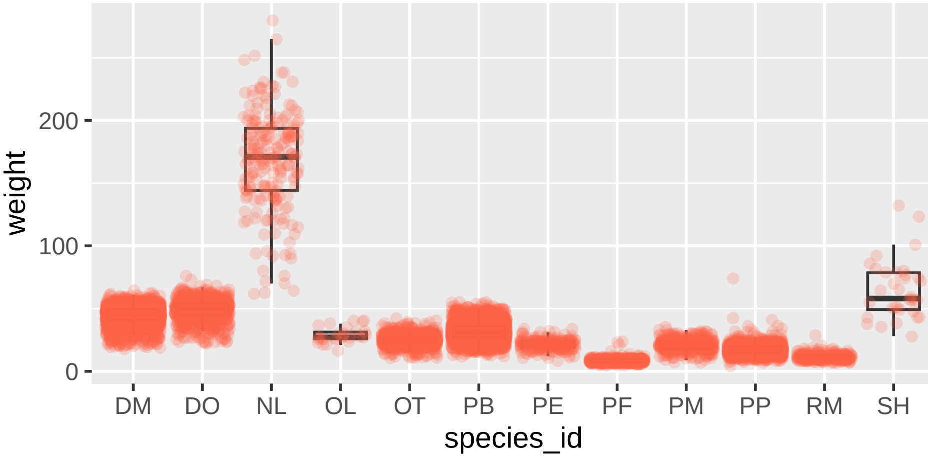

```
    ## alpha = 0.2 decreases the opacity of the points, to not be too busy
```

> Notice how the boxplot layer is behind the jitter layer? What would you change in the code to put the boxplot in front of the points?

Challenge 3 (Part 1)

> Boxplots are useful summaries, but hide details of 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. For help see `geom_violin()`. Start with the boxplot we created:

```
ggplot(data = surveys, mapping = aes(x = species_id, y = weight)) +
  geom_boxplot(alpha = 0) +
  geom_jitter(alpha = 0.2, color = "tomato")
```




```
##  Start with the boxplot we created
##  1. Replace the boxplot with a violin plot. For help, see geom_violin().
##  You might need to decrease opacity even more to see the violins (try 0.03)
```

Challenge 3 (Part 2)

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

Create a boxplot for `hindfoot_length`. This time overlay the boxplot layer over a jitter layer that shows the actual measurements.

```
##  First: create boxplot for hindfoot_length` overlaid on a jitter layer.
```

Challenge 3 (Part 3)

Now, 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`. If `plot_id` was a character instead, how would the graph be different?

```
## Next: 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`. If plot_id was a character instead,
## how would the graph be different?
```

Bonus violin plot example (DatasauRus)

> The previous example doesn't fully illustrate the power of violin plots. This example from the `datasauRus` package (https://www.autodeskresearch.com/publications/samestats) shows five different distributions that have exactly the same summary statistics and boxplots but very different shapes:

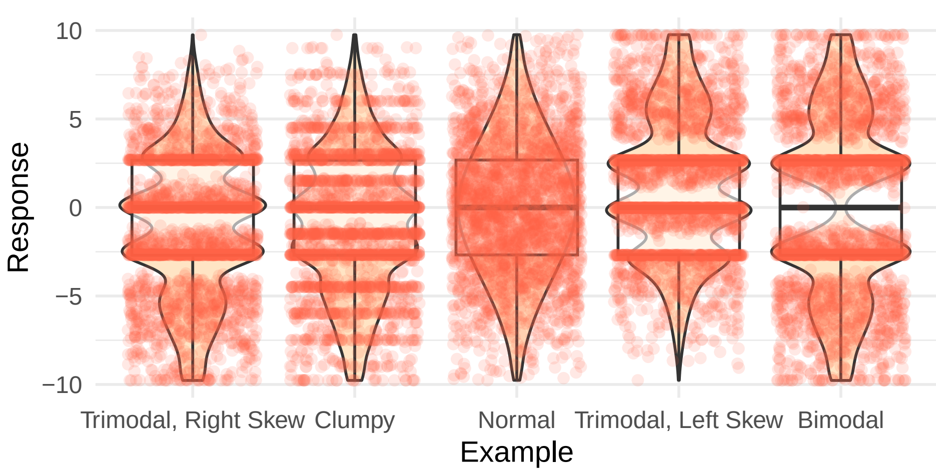
```
library(datasauRus)
data(box_plots)

box_plots_long <- box_plots %>%
  pivot_longer(cols=1:5)

box_plots_long <- box_plots_long %>%
  rename(Example=name, Response=value)

box_plots_long <- box_plots_long %>%
  mutate(Example = factor(Example))

ggplot(box_plots_long, aes(x=Example, y=Response)) +
  geom_violin(fill="bisque") +
  geom_boxplot(alpha=.6) +
  geom_jitter(alpha=0.15, col="tomato") +
  theme_minimal() +
  scale_x_discrete(labels=c('Trimodal, Right Skew', 'Clumpy', 'Normal',
                            'Trimodal, Left Skew',  'Bimodal'))
```



Plotting Single Variables

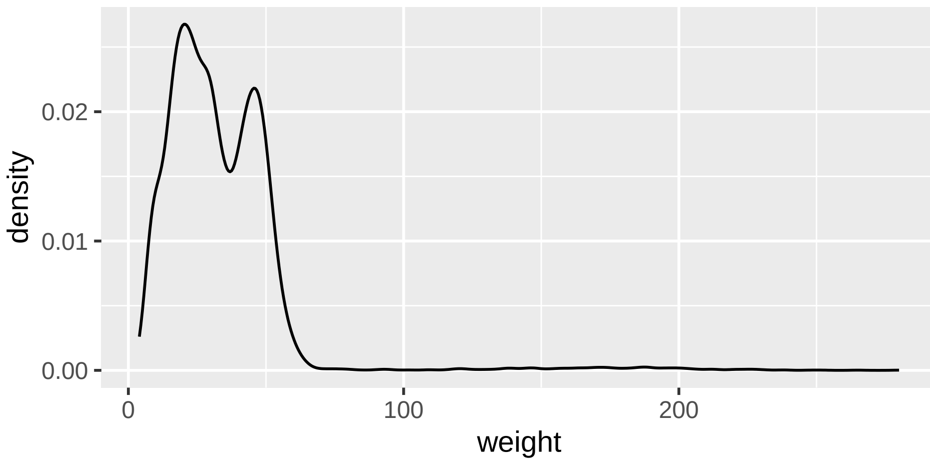
Distribution Plots (Quantitative Variables)

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.

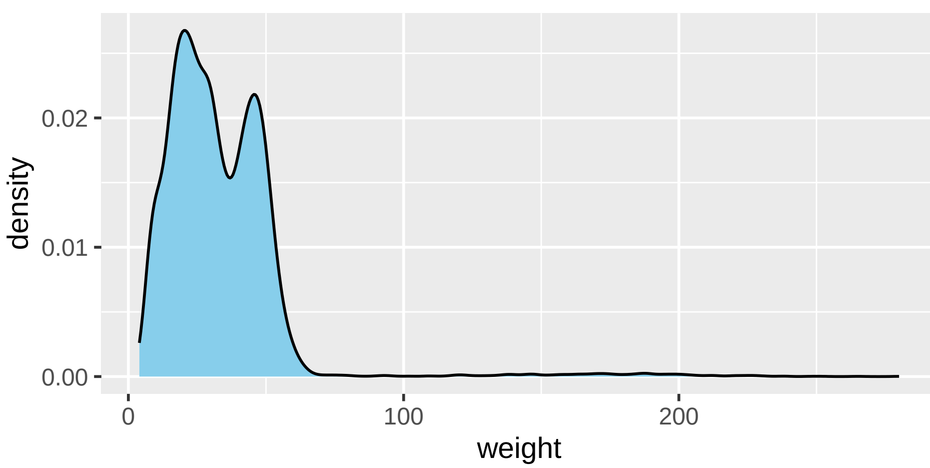
To visualize the distribution of rodent weights we could aggregate over all species, years, plots, etc. and produce a single density plot:

```
surveys %>%
  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:

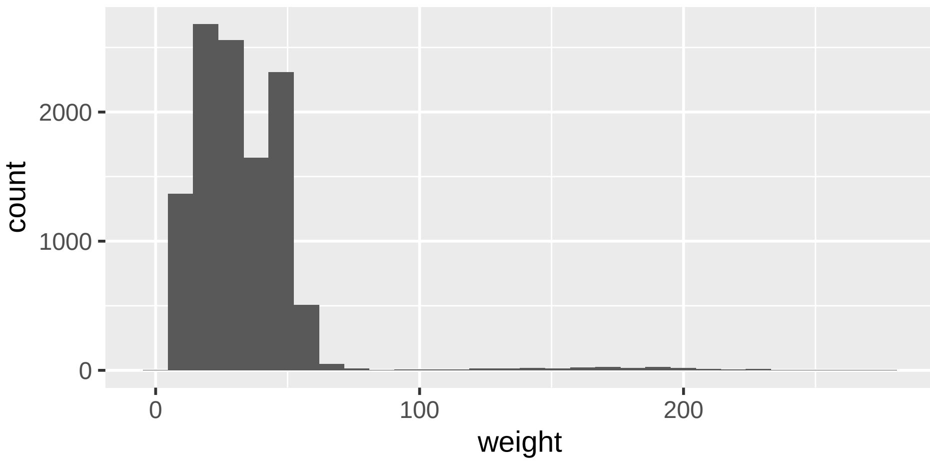
```
surveys %>%
  ggplot(mapping = aes(x = weight)) +
  geom_density(fill = "sky blue")
```



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.

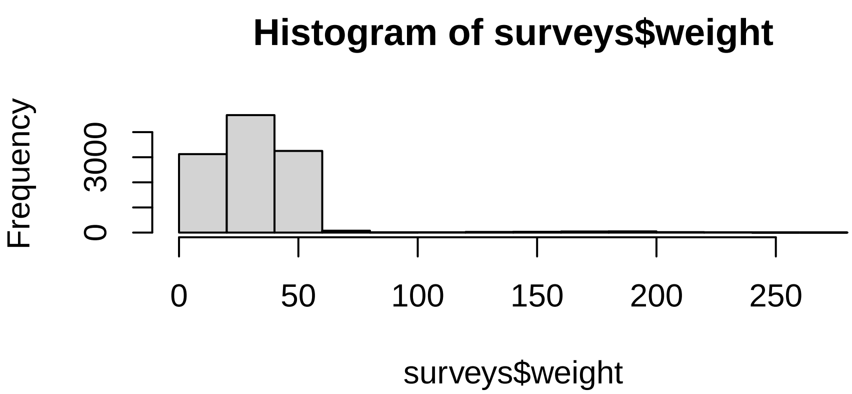
```
surveys %>%
  ggplot(mapping = aes(x = weight)) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



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

```
hist(surveys$weight)
```



There is no single right answer for the number of bins. There are some "plug-in" choices for number of bins that can be used, but you are always welcome to explore different numbers of bins to see if features you are seeing persist when you choose more or less bins.

Challenge 4

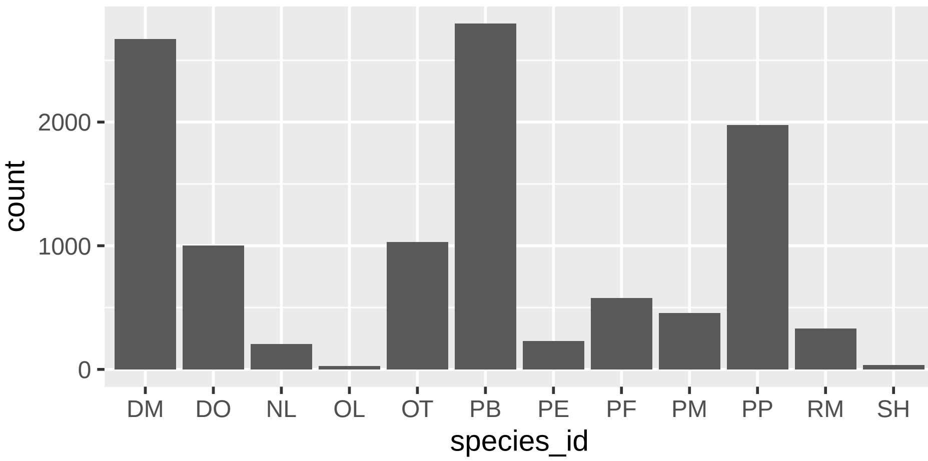
> Use the `bins` argument in `geom_histogram()` to play around with the number of bins in your histogram. Try different numbers of bins to explore how that changes the results!

```
## Your code to answer the challenge goes here!
```

Bar Charts (Categorical Variables)

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` dataset, grouped by their species ID.

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



The x-axis displays the levels of `species_id`, a variable in the `surveys` 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, quartiles, median) and plot those summaries in a tidy box, with "potential outliers" (data over 1.5*IQR from Q1 or Q3) plotted as points.
- Smoothers (as used in `geom_smooth`) fit a model to your data (you can specify, but we used the `gam` (generalized additive model from the `mgcv` package) default) and then plot the predicted means from that model (with associated 95% 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

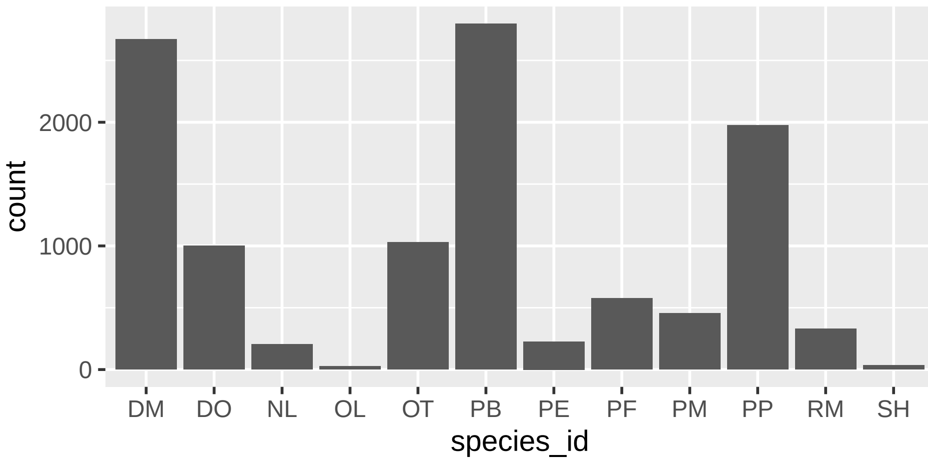
2. `geom_bar` then transforms the data using the `count` statistic

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

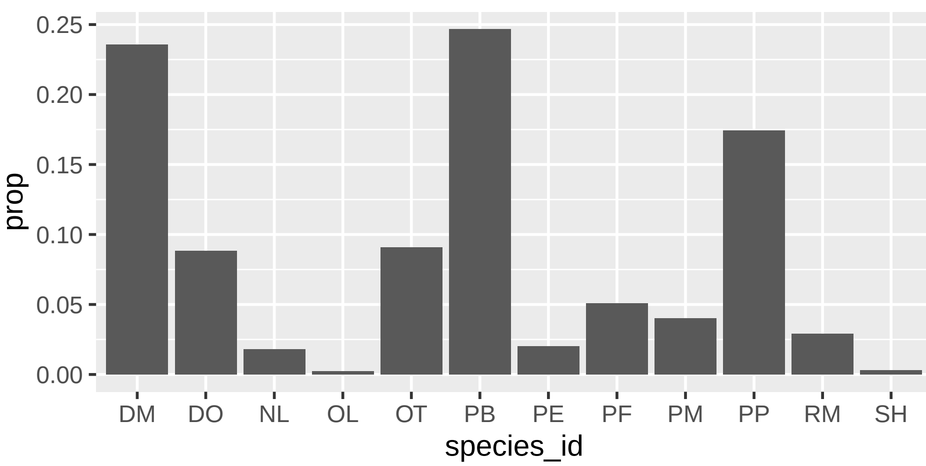
Generally, you can use `geoms` and `stats` interchangeably. This is because every `geom` has a default `stat` and vice versa. For example, the following code produces the same output as above:

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

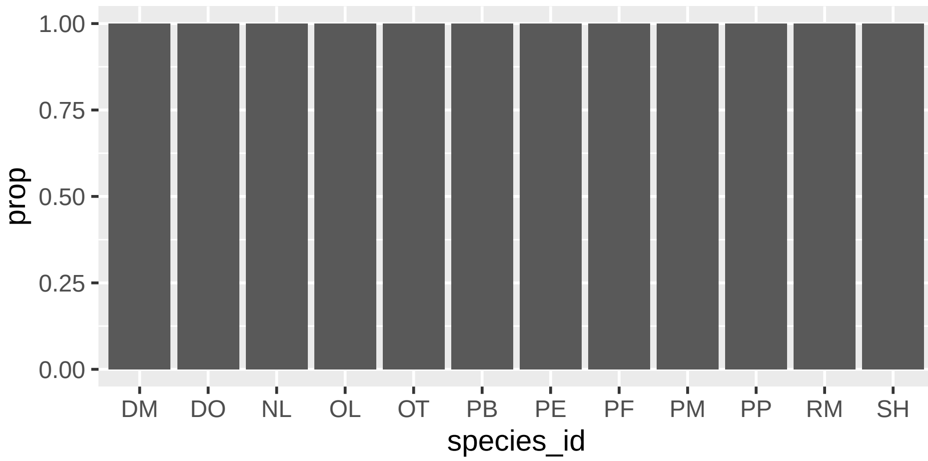


Challenge 5

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

```
## What is wrong with this plot?
surveys %>%
  ggplot(mapping = aes(x = species_id)) +
```

```
  geom_bar(aes(y = after_stat(prop)))
```

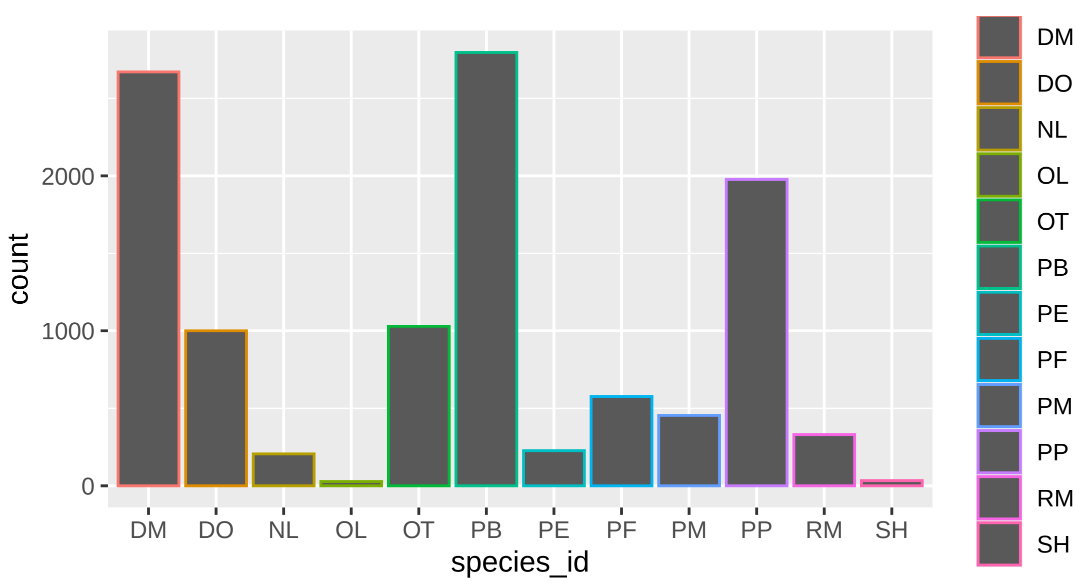


Colored and/or Stacked Bar Charts

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.

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

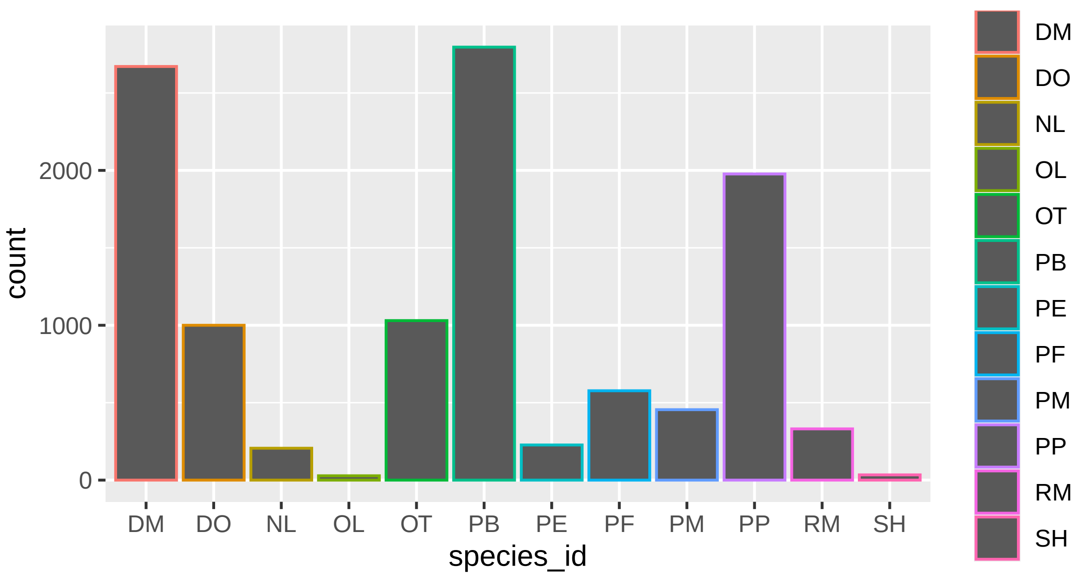


We notice that the color only appears in the outline of the bars. For a bar chart, the aesthetic that we are interested in is the `fill` of the bars.

Challenge 6

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

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

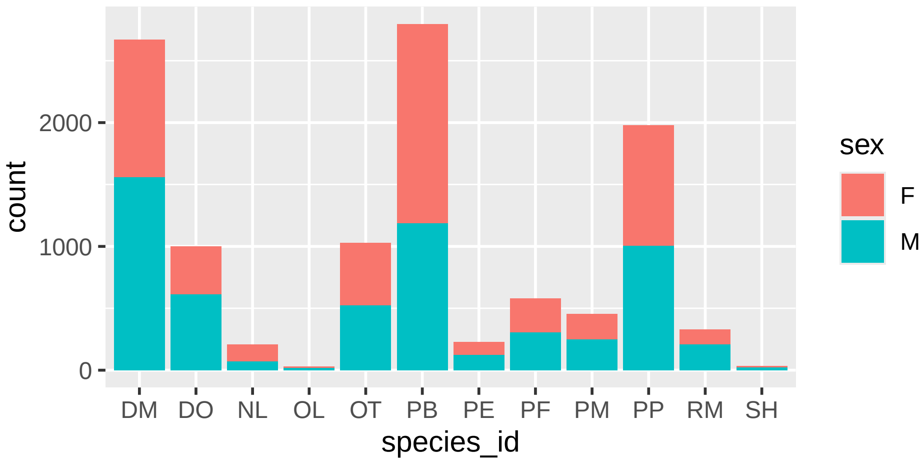


Stacking Bars Now suppose you are interested in whether the number of male and female rodents captured differs by species. This would require for you to create a bar plot with two categorical variables. You have two options:

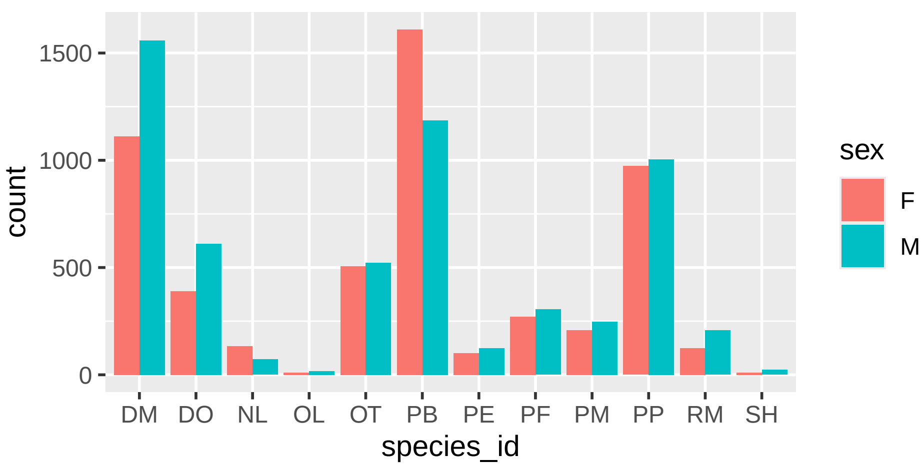
1. each of the bars for sex could be stacked within a species *OR*
2. the bars for sex could be side-by-side within a species

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



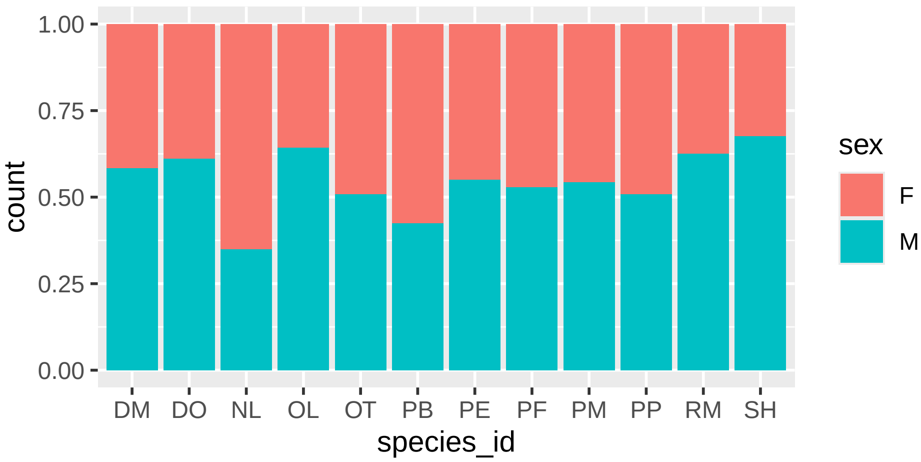

```r
surveys %>%
  ggplot(mapping = aes(x = species_id, fill = sex)) +
  geom_bar(position = "dodge")
```



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?

Finally, we can also choose the `position` to be `fill` for the bars and to `fill` the bars based on `sex`:

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



Notice that the y-axis label still says "count" instead of "proportion". We will learn how to change labels later when we discuss **Customization**.

Time-series Data

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

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

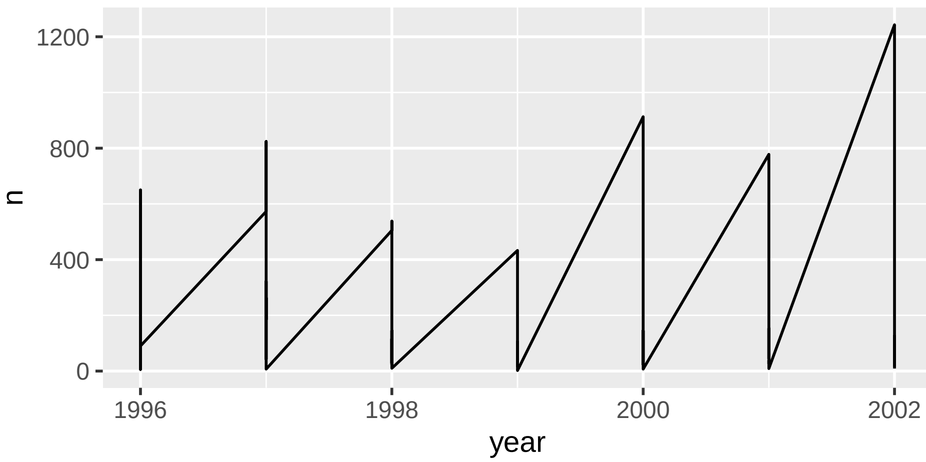
```
yearly_counts <- surveys %>%
  count(year, genus)
  ## counts the number of observations (rows) for each year, genus combination
  ## and creates a new variable "n" and creates a new variable "n"

yearly_counts
```

```
## # A tibble: 52 x 3
##     year genus               n
##    <dbl> <chr>           <int>
##  1  1996 Chaetodipus       328
##  2  1996 Dipodomys         650
##  3  1996 Neotoma             6
##  4  1996 Onychomys         121
##  5  1996 Perognathus       324
##  6  1996 Peromyscus         85
##  7  1996 Reithrodontomys    90
##  8  1997 Chaetodipus       573
##  9  1997 Dipodomys         824
## 10  1997 Neotoma            43
## # i 42 more rows
```

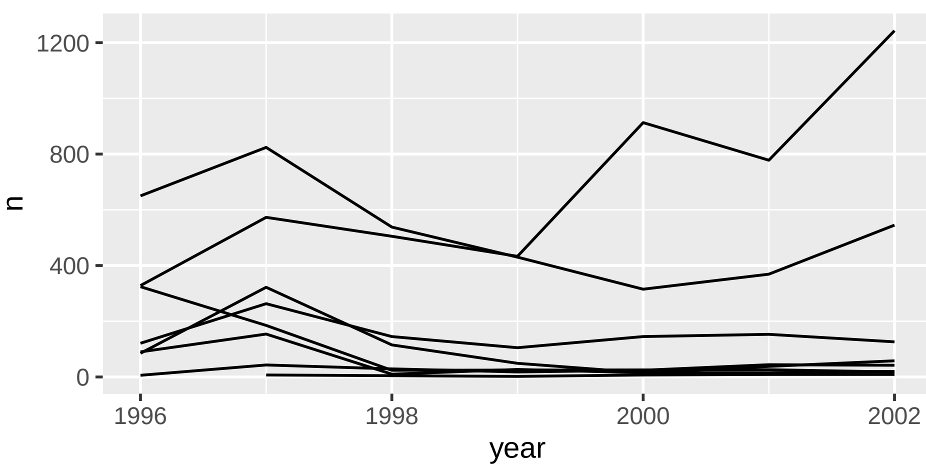
Time series data can be visualized as a line plot with years on the x-axis and counts on the y-axis:

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



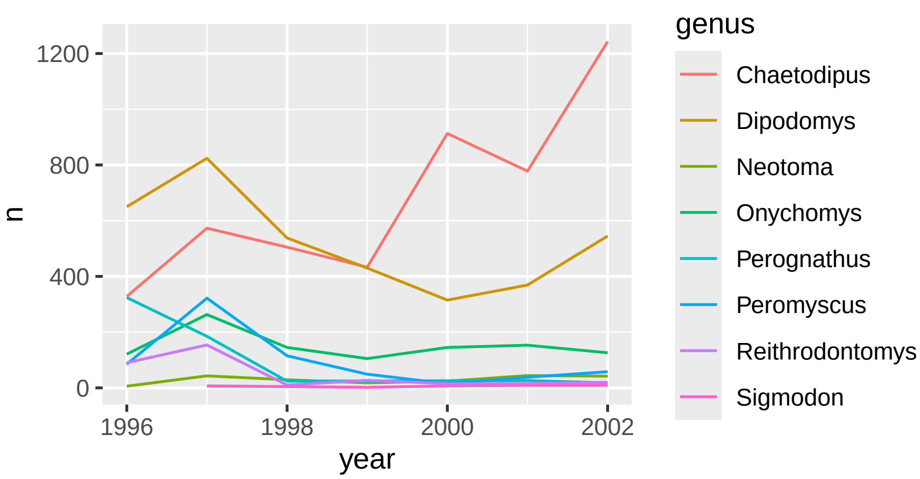
Unfortunately, this does not work because we plotted data for all the genera together. We need to tell `ggplot()` to draw a line for each genus by modifying the aesthetic function to include `group = genus`:

```
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()
```



> **Note:** When specifying the color for a line graph, you don't need to use both the `color = <VARIABLE>` argument and the `group = <VARIABLE>` argument. Both do the same grouping of observations! So you just need to specify the `color` argument.

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:

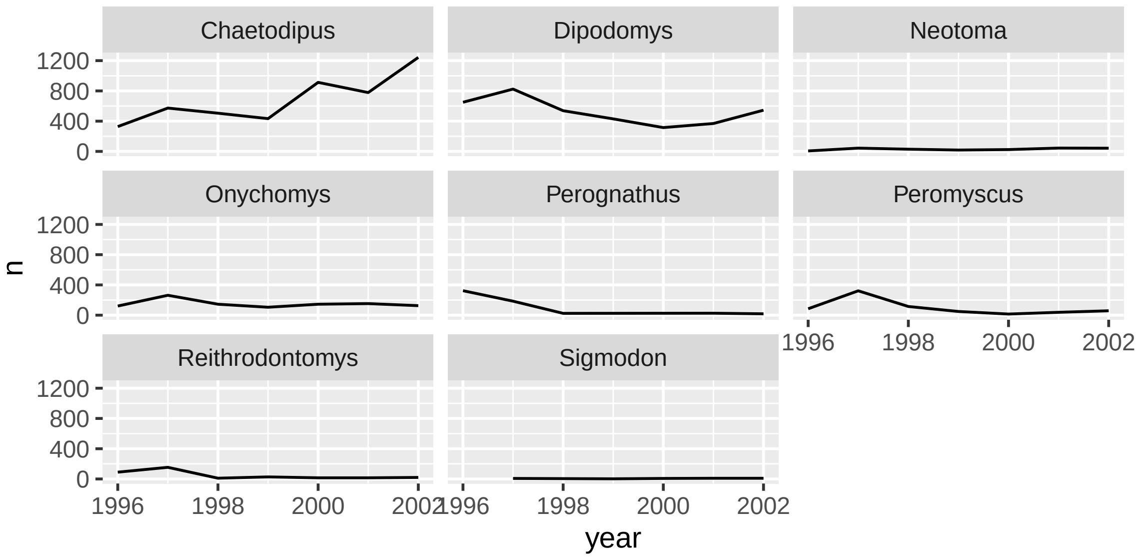
- `facet_wrap()` arranges a one-dimensional sequence of panels to allow them to cleanly fit on one page – used for one variable
- `facet_grid()` allows you to form a matrix of rows and columns of panels – used for two variables

Both geometries allow you to specify faceting variables using formula notation or the `vars()` function. We will use the formula notation, as you will use this notation when creating models (not covered in this workshop).

This looks like: `facet_wrap(facets = ~ facet_variable)` or `facet_grid(row_variable ~ 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 = ~ genus)
```



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

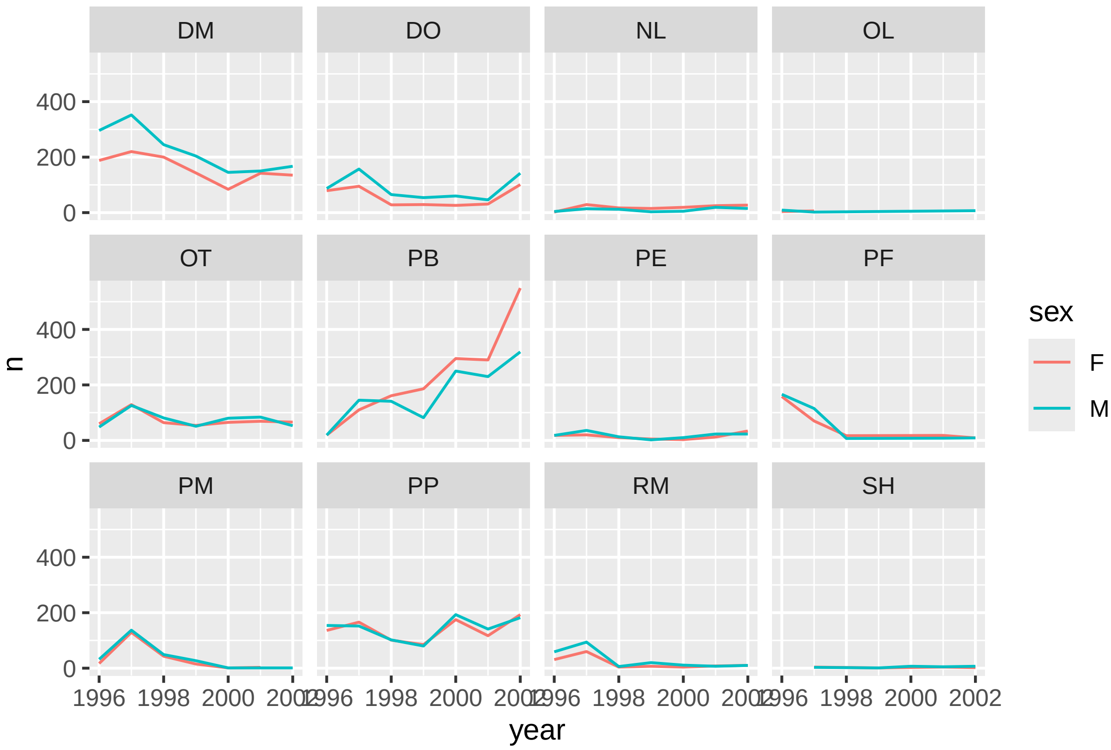
```
yearly_sex_counts <- surveys %>%
  count(year, species_id, sex)
## counts the number of observations (rows) for each year, species, sex combination

yearly_sex_counts
```

```
## # A tibble: 148 x 4
##     year species_id sex       n
##    <dbl> <chr>      <chr> <int>
##  1  1996 DM         F       188
##  2  1996 DM         M       296
##  3  1996 DO         F        79
##  4  1996 DO         M        87
##  5  1996 NL         F         2
##  6  1996 NL         M         4
##  7  1996 OL         F         4
##  8  1996 OL         M         9
##  9  1996 OT         F        60
## 10  1996 OT         M        48
## # i 138 more rows
```

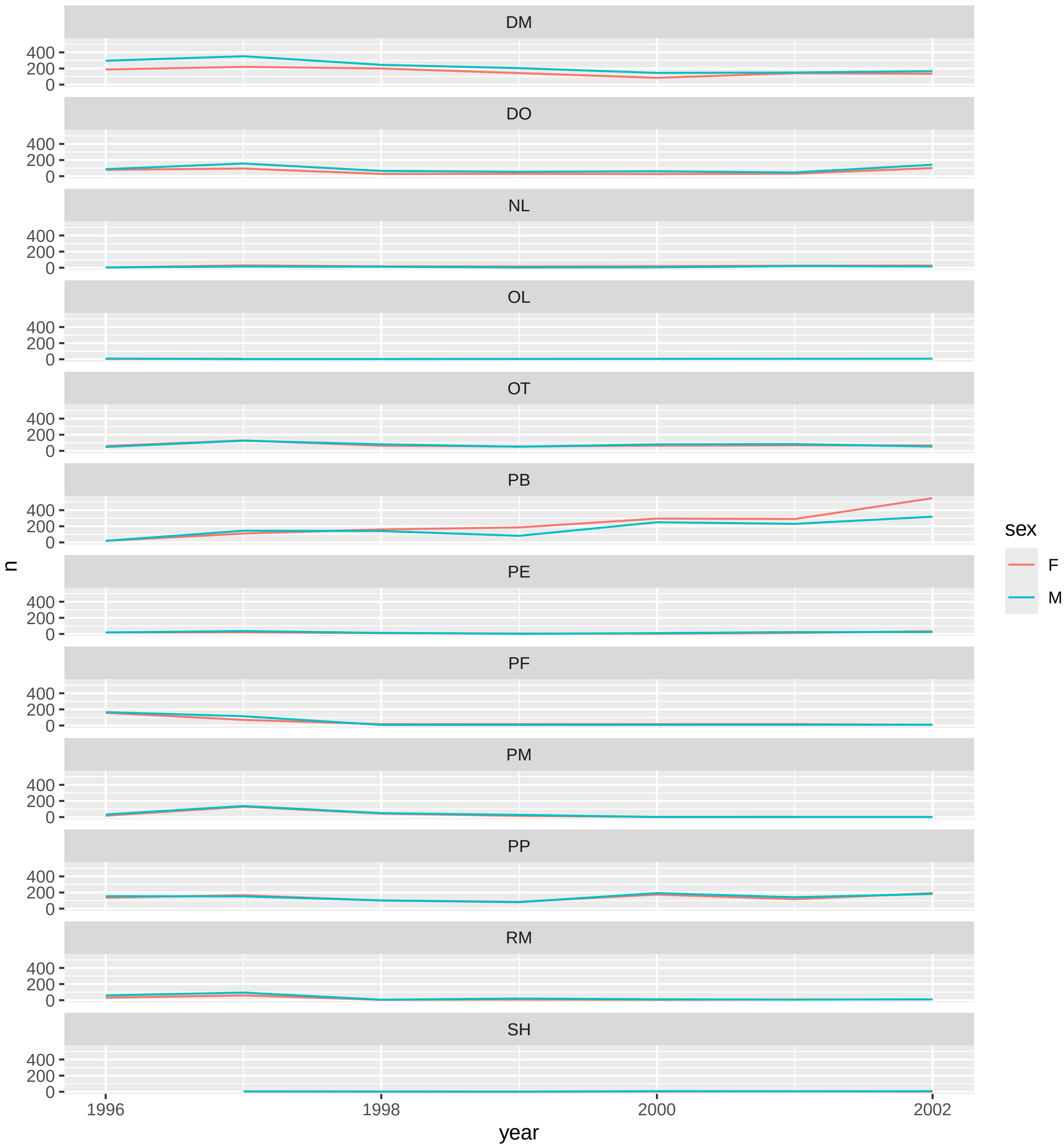
We can now make the faceted plot by splitting further by sex using `color` (within each panel):

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



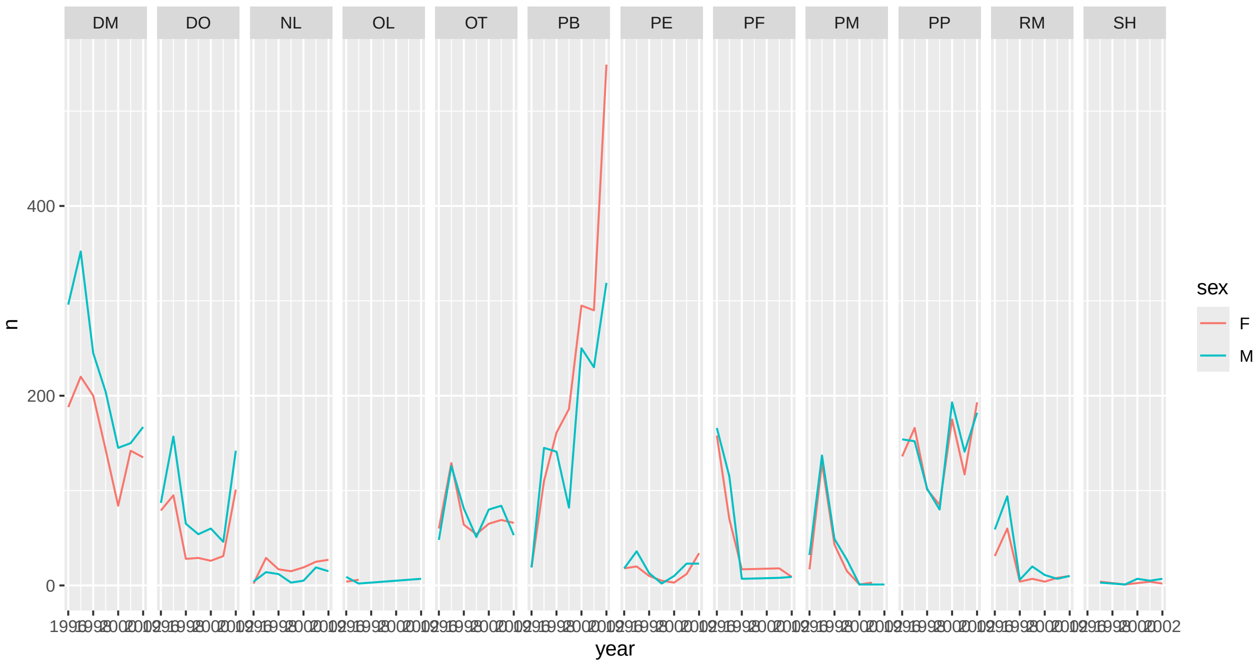
You can also organize the panels only by rows (or only by columns), using the optional `nrow` and `ncol` arguments:

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




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

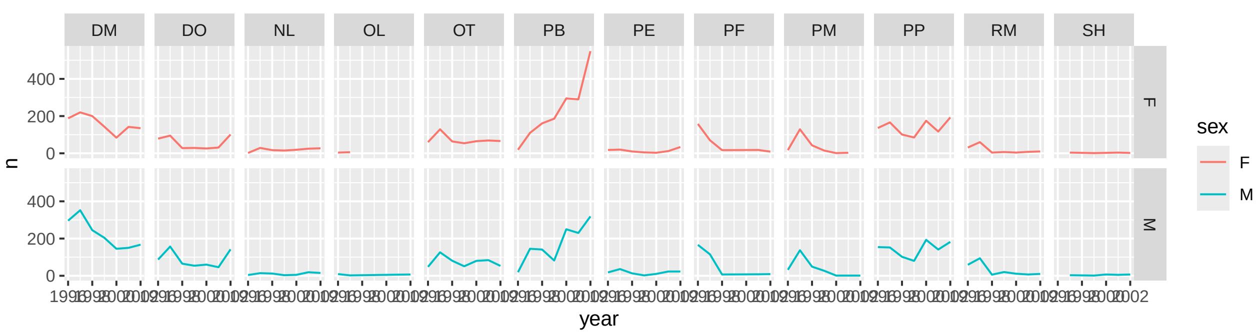
```
yearly_sex_counts %>%
  ggplot(mapping = aes(x = year, y = n, color = sex)) +
    geom_line() +
    facet_wrap(~ species_id, nrow = 1)
```




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

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

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



Challenge 7

> 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!

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

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

```
## Edit the following ggplot() code for the plot here:
yearly_species_weight %>%
  ggplot(mapping = aes(x = year, y = n, color = avg_weight)) +
    geom_line() +
    facet_wrap(facets =  ~ species_id)
```

```
## Don't know how to automatically pick scale for object of type <function>.
## Defaulting to continuous.
```

```
## Error in `geom_line()`:
## ! Problem while computing aesthetics.
## i Error occurred in the 1st layer.
## Caused by error in `compute_aesthetics()`:
## ! Aesthetics are not valid data columns.
## x The following aesthetics are invalid:
## x `y = n`
## i Did you mistype the name of a data column or forget to add `after_stat()`?
```

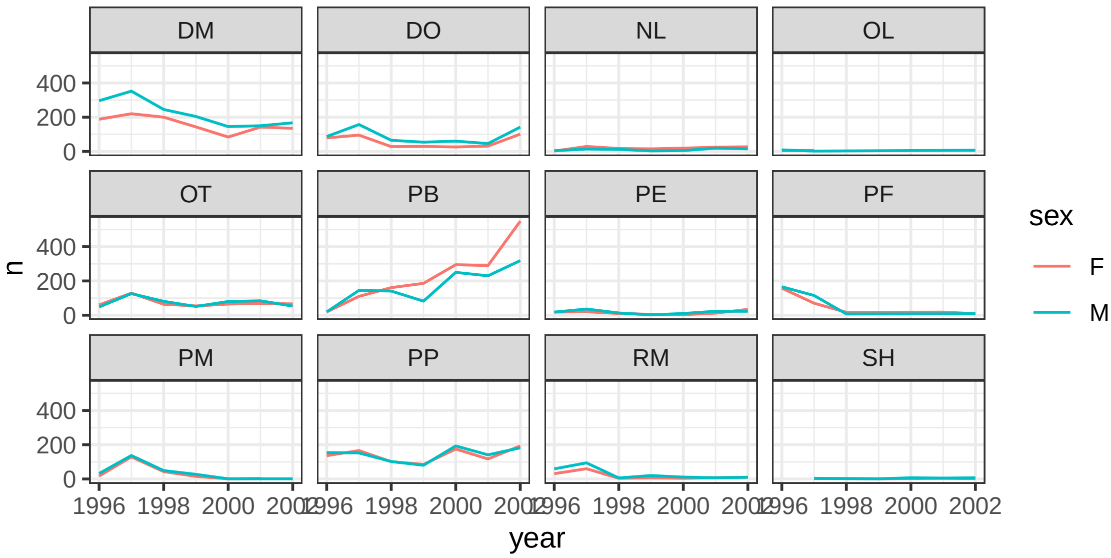
```
## Your ggplot() code for the plot goes here!
```

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(~ 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 8

> Use what you just learned to add the plotting background theme of your choosing to the plot you made in Challenge 7!

```
## Your ggplot() code for the plot goes here!
```

Customization

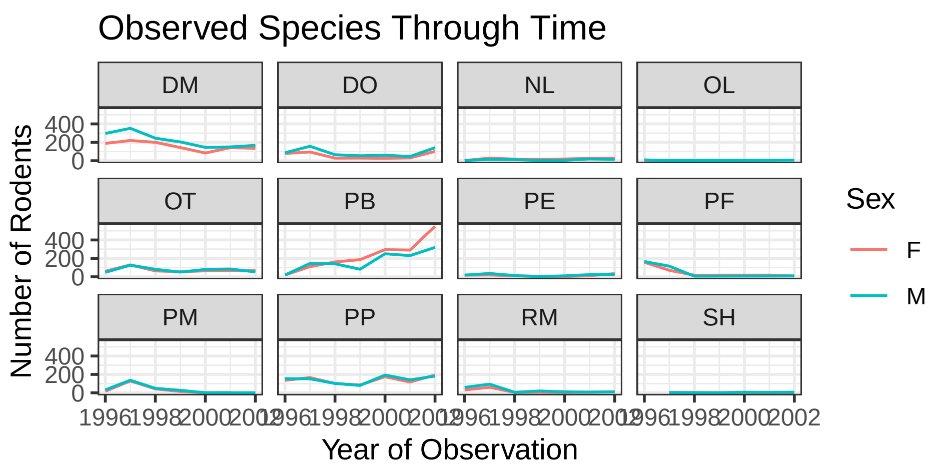
Take a look at the **`ggplot2`** cheat sheet, and think of ways you could improve the previous plot.

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(~ species_id) +
  theme_bw() +
  labs(title = "Observed Species Through Time",
       x = "Year of Observation",
```

```
       y = "Number of Rodents",
       color = "Sex")
```



Tip: Wrapping Titles

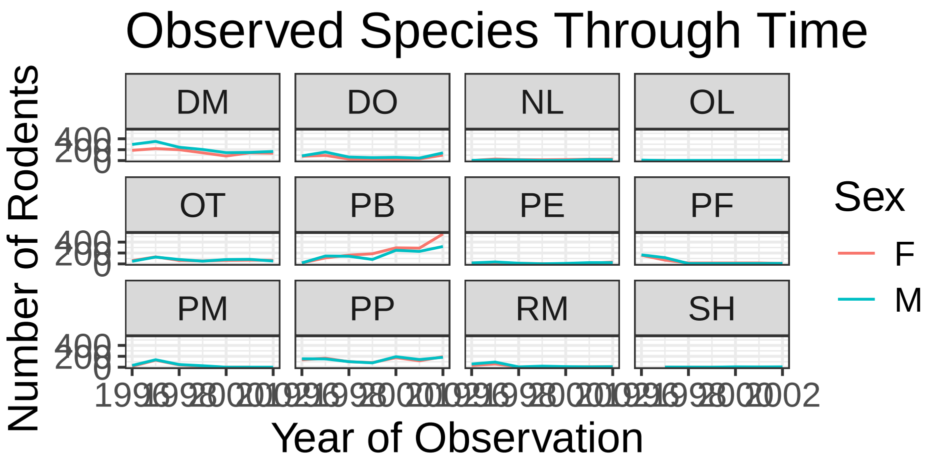
Sometimes the titles we wish to have for our plots are longer than the space originally allotted. If you create a title and the text is running off the plot you can add a `\n` inside your title to force a line break (`\n` stands for new line).

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.

In the last plot, 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(~ species_id) +
  theme_bw() +
  labs(title = "Observed Species Through Time",
      x = "Year of Observation",
      y = "Number of Rodents",
      color = "Sex") +
  theme(text = element_text(size = 16))
```

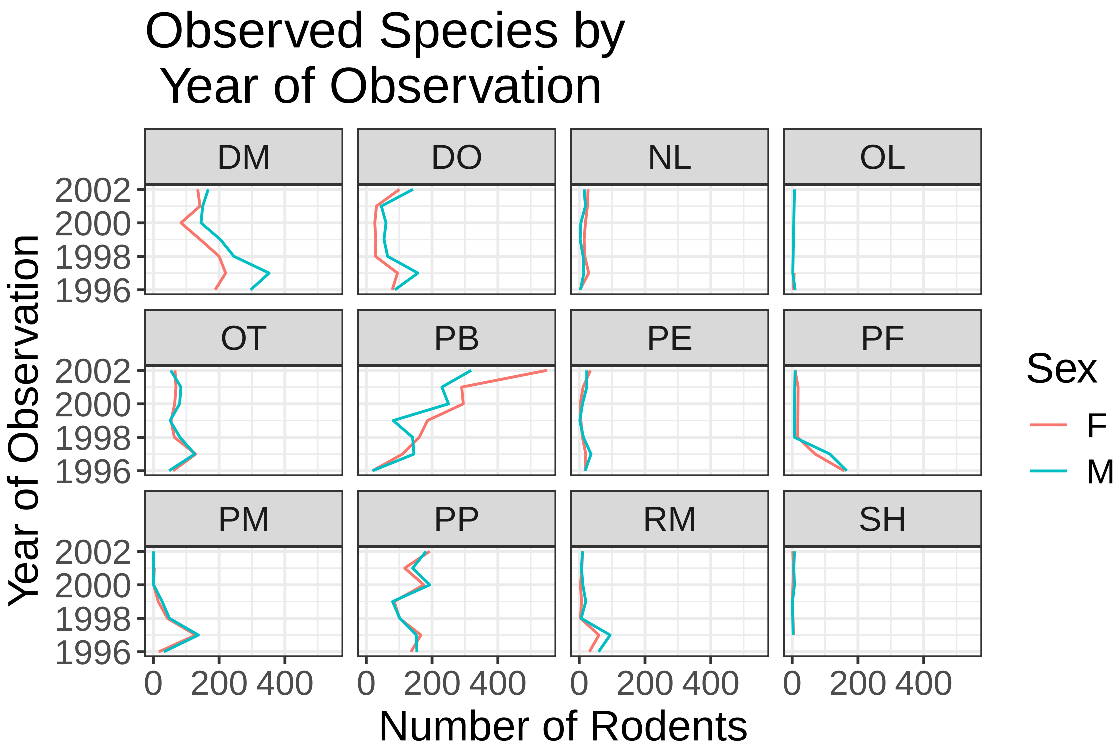



```
## sets ALL the text on the plot to be size 16
```

Note:
`theme_bw()` is a function for a **specific** theme and `theme()` is a generic function for a **variety** of different themes!

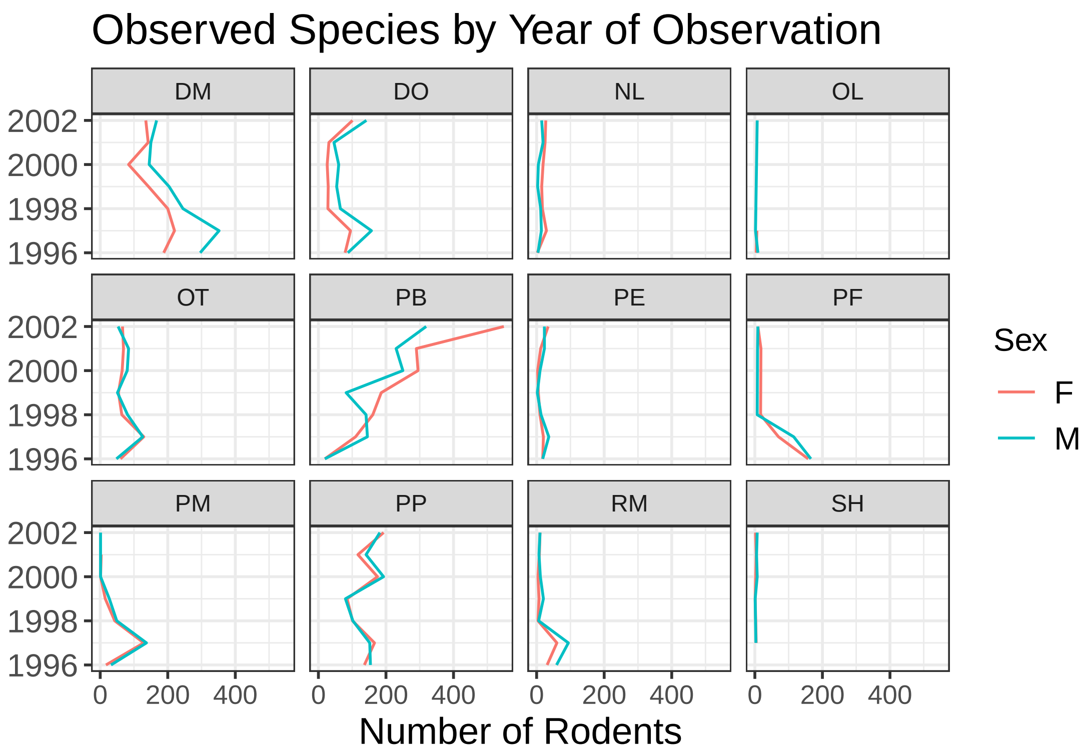
After our manipulations, you may notice that the values on the x-axis are still not properly readable. Let's swap the orientation of the labels, so the reader doesn't have to tilt their head when reading our plot! The `coord_flip()` function easily changes the x- and y-axis.

```
yearly_sex_counts %>%
  ggplot(mapping = aes(x = year, y = n, color = sex)) +
  geom_line() +
  facet_wrap(~ species_id) +
  theme_bw() +
  labs(title = "Observed Species by \n Year of Observation",
      x = "Year of Observation",
      y = "Number of Rodents",
      color = "Sex") +
  theme(text = element_text(size = 16)) +
  coord_flip()
```



This definitely makes the reader tilt their head less! But, the text on the x-axis is a bit too large to separate the numbers. We can specify the text size for each element of the plot independently, if we so wish. This would look something like this:

```
yearly_sex_counts %>%
  ggplot(mapping = aes(x = year, y = n, color = sex)) +
  geom_line() +
  facet_wrap(~ species_id) +
  theme_bw() +
  labs(title = "Observed Species by Year of Observation",
        x = "",
        y = "Number of Rodents",
        color = "Sex") +
  theme(axis.text.x = element_text(size = 10),
        axis.text.y = element_text(size = 12),
        axis.title.x = element_text(size = 14),
        legend.text = element_text(size = 12),
        legend.title = element_text(size = 12),
        plot.title = element_text(size = 16)) +
  coord_flip()
```

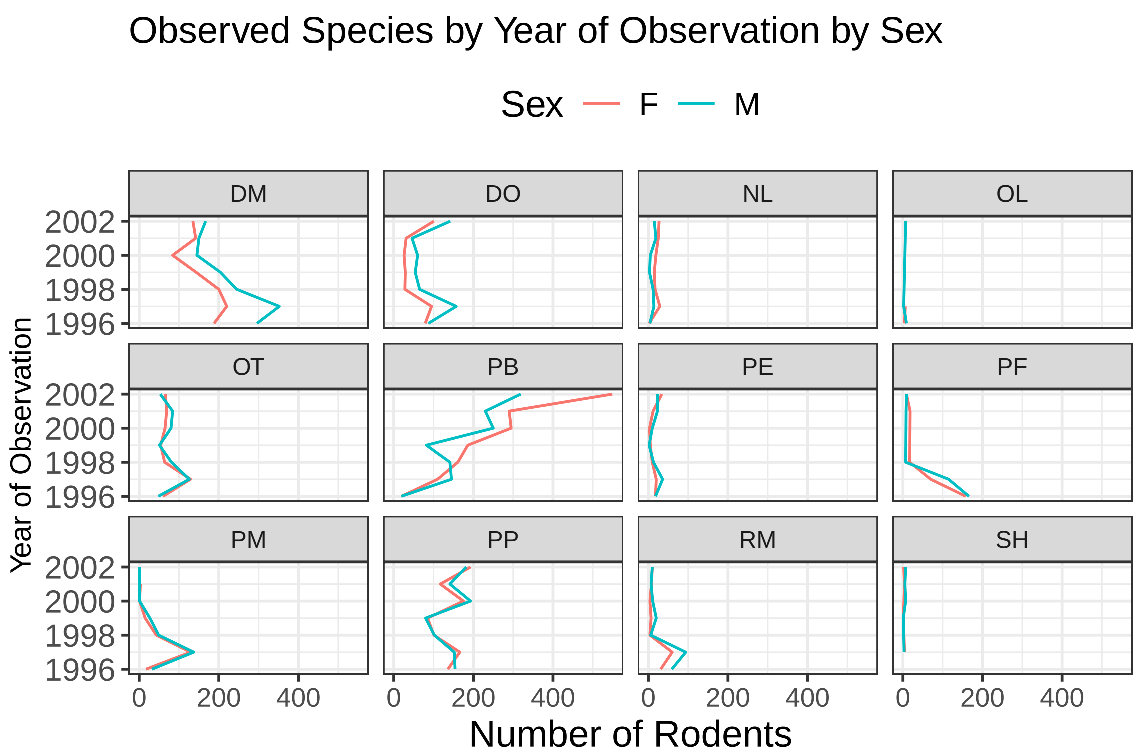


Legend Position

By default in `ggplot2` the legend is positioned on the right hand side. However, you are able to change the position of the legend to the left hand side, the top of the plot, or the bottom of the plot.

This is done by adding a `legend.position` theme to the plot's `theme()`'s.

```
yearly_sex_counts %>%
  ggplot(mapping = aes(x = year, y = n, color = sex)) +
    geom_line() +
    facet_wrap(~ species_id) +
    labs(title = "Observed Species by Year of Observation by Sex",
        x = "Year of Observation",
        y = "Number of Rodents",
        color = "Sex") +
  theme_bw() +
  theme(axis.text.x = element_text(size = 10),
        axis.text.y = element_text(size = 12),
        axis.title.x = element_text(size = 14),
        legend.text = element_text(size = 12),
        legend.title = element_text(size = 14),
        plot.title = element_text(size = 14),
        legend.position = "top") +
  coord_flip()
```



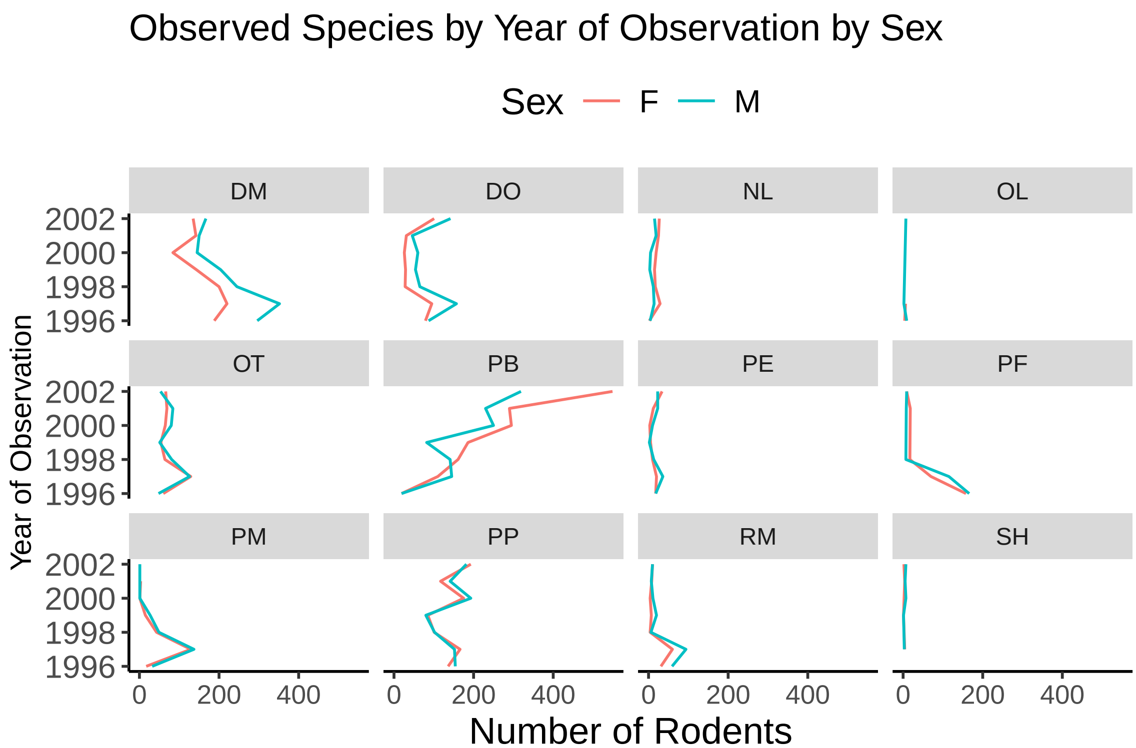
Removing Grid Lines

By default, the background of a `ggplot()` contains both minor and major gridlines. These can make the plot look a bit busy and sometimes difficult for the reader to follow. As you may have guessed, to remove these gridlines, we add another theme to our plot.

This looks like this:

```
yearly_sex_counts %>%
  ggplot(mapping = aes(x = year, y = n, color = sex)) +
  geom_line() +
  facet_wrap(~ species_id) +
  labs(title = "Observed Species by Year of Observation by Sex",
      x = "Year of Observation",
      y = "Number of Rodents",
      color = "Sex") +
  theme(axis.text.x = element_text(size = 10),
        axis.text.y = element_text(size = 12),
        axis.title.x = element_text(size = 14),
        legend.text = element_text(size = 12),
        legend.title = element_text(size = 14),
        plot.title = element_text(size = 14),
        legend.position = "top",
        ## New themes for the grid lines
        axis.line = element_line(color = "black"),
```

```
        ##
        panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(),
        panel.border = element_blank(),
        panel.background = element_blank()) +
  coord_flip()
```



Let's break these options down!

- The `axis.line` option declares what color the x- and y-axis lines should be. (Change it to a different color, if you don't believe me!)
- The `panel.grid.major` removes the major grid (the one associated with the ticks from the x- and y-axis).
- The `panel.grid.minor` removes the minor grid (the one *between* the x- and y-axis ticks).
- The `panel.border` removes the border around the plot.
- The `panel.background` performs a similar action to `theme_bw()`, but it keeps the border around the facet labels.

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.

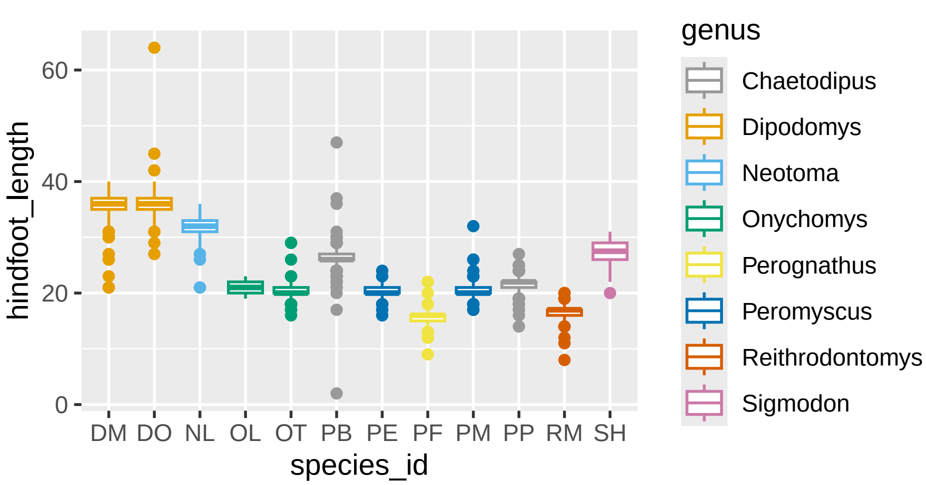
Manual Specification Add the `scale_color_manual()` or `scale_fill_manual()` functions to your plot and directly specify the colors you want to use. You can either:

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

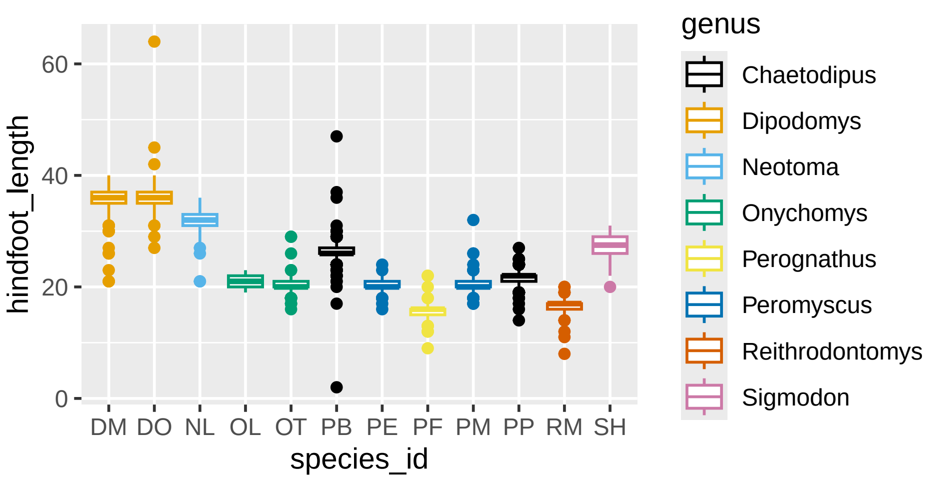
```
# A color deficient friendly palette with grey:
cbPalette_grey <- c("#999999", "#E69F00", "#56B4E9", "#009E73", "#F0E442",
                    "#0072B2", "#D55E00", "#CC79A7")

# A color deficient friendly palette with black:
cbPalette_blk <- c("#000000", "#E69F00", "#56B4E9", "#009E73", "#F0E442",
                   "#0072B2", "#D55E00", "#CC79A7")

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




```
surveys %>%
  ggplot(aes(x = species_id, y = hindfoot_length, color = genus)) +
  geom_boxplot() +
  scale_color_manual(values = cbPalette_blk)
```



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

- `RColorBrewer`: using `scale_fill_brewer()` or `scale_color_brewer()`
- `viridis`: using `scale_color_viridis_d()` for discrete data, `scale_color_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 9

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.
- Try using a different color palette
- Can you find a way to change the name of the legend? What about its labels? (see http://www.cookbook-r.com/Graphs/Colors_(ggplot2)/).

```
## your code for the challenge goes here!
```

Arranging 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()`** (make sure to scroll down in the window to see all the code):

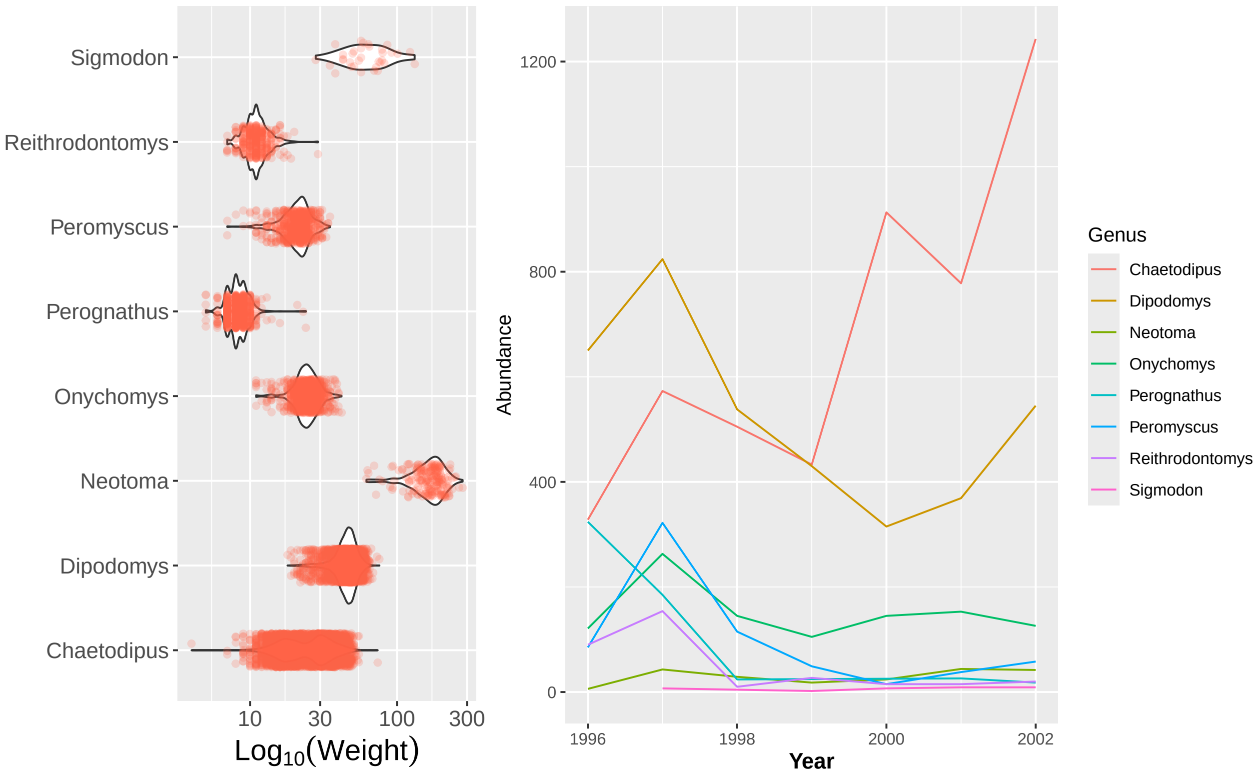
```
library(gridExtra)

spp_weight_boxplot <- surveys %>%
  ggplot(aes(x = genus, y = weight)) +
  geom_violin() +
```

```
  geom_jitter(color = "tomato", width = 0.2, alpha = 0.2) +
  scale_y_log10() +
  ## log (base 10) transforms the y-axis variable
  ## (helps to make the plot less skewed)
  labs(x = "",
       ## removes the y-axis label
       y = expression(Log[10](Weight))) +
       ## Expression creates a mathematical expression in the axis label
       ## the [10] refers to the subscript next to Log
  coord_flip() +
  theme(axis.text.y = element_text(size = 12),
        axis.text.x = element_text(size = 12),
        text = element_text(size = 16))

spp_count_plot <- yearly_counts %>%
  ggplot(aes(x = year, y = n, color = genus)) +
  geom_line() +
  labs(x = "Year",
       y = "Abundance",
       color = "Genus") +
  theme(axis.title.x = element_text(face = "bold", size = 12))
  ## To make your axis title boldface, this is what you need!

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



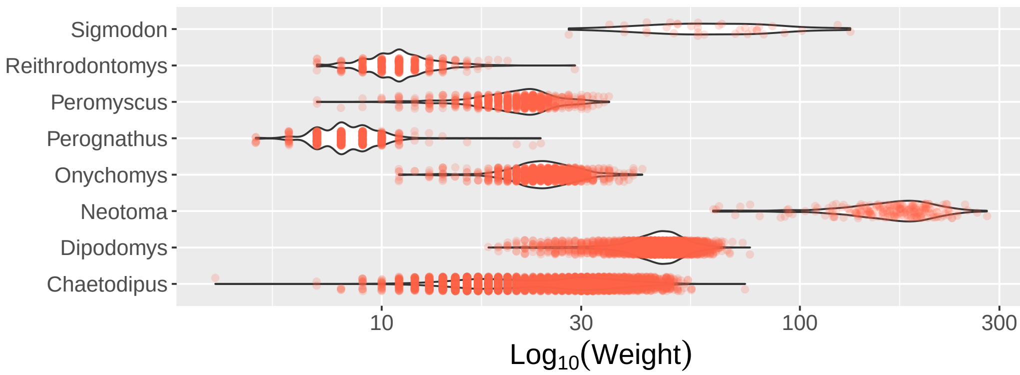

```
## nrow and ncol specify how many rows/columns you want the arranged plots to be in
## widths specify what proportion of the overall plotting area each plot takes up
```

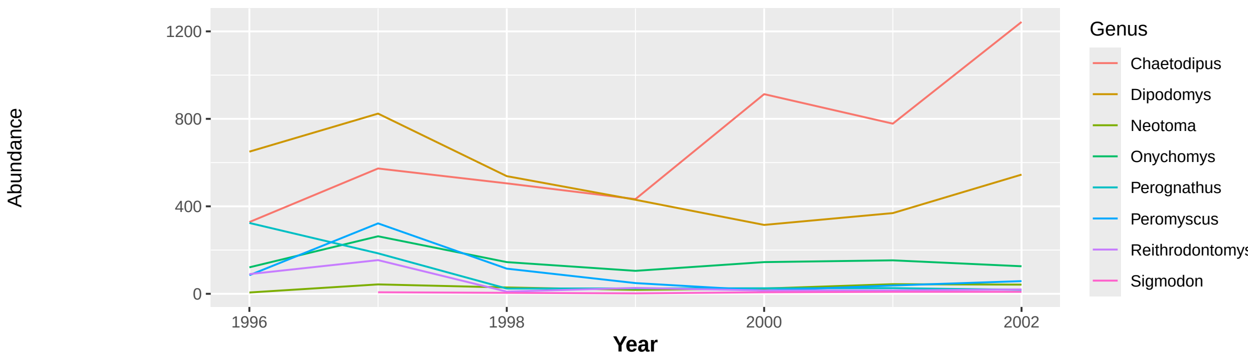
In addition to the `ncol` and `nrow` arguments, used to make simple arrangements, there are tools for constructing more complex layouts.

For more assistance arranging plots with `grid.arrage()`. We find the following vignette **very** helpful! https://cran.r-project.org/web/packages/egg/vignettes/Ecosystem.html

Another option for combining plots is the patchwork package. It uses a sort of formula interface for defining the layout of multiple plots. For example, you can get two plots side-by-side in a one row, two column array with `p1 + p2` and two plots stacked into two rows and one column with `p1 / p2`. This provides both a quick and powerful way to arrange ggplots you have created.

```
spp_weight_boxplot / spp_count_plot
```





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. The first line of code checks to see if that folder exists, and if not, creates it.

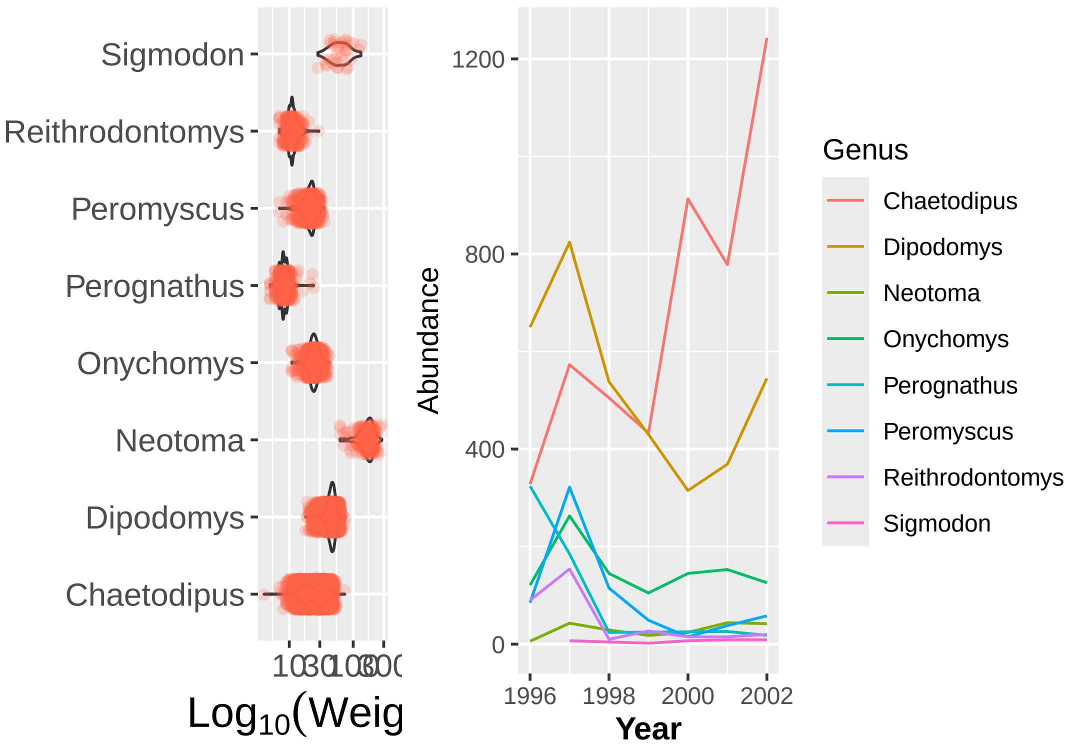
```
# Specify where you want figures to be saved relative to current directory
if(!dir.exists("fig")){dir.create("fig")}

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

```
              labs(title = "Observed genera through time",
                   x = "Year of observation",
                   y = "Number of individuals") +
              theme_bw() +
              coord_flip() +
              theme(axis.text.y = element_text(color = "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)
```

```
## Saving 10 x 4 in image
```

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

Suggestions for your own work

The goal of this workshop was to teach you to write code in `R` to learn data visualization using `ggplot2`. The first workshop in our series contains more information on how to get started working in `R` using RStudio (see http://www.montana.edu/datascience/training/). The code chunks in this interactive document mimic the code chunks you can use on your own projects in RMarkdown but you will need to download and install both

R and RStudio on your own computer.

Interactive Graphics (Bonus Material)

In certain situations, static displays can limit the sorts of information available and do not allow easy interrogation for information on individual aspects of plots. Obviously, most print journals do not have a way to have readers interact with the printed page, but in digital venues there are some possibilities. Of particular interest here are interactive graphics that can function on websites and in blog posts or even in certain presentation formats. One way to do this that leverages the previous work in making `ggplot`-style graphics is using the `ggplotly` function from the `plotly` R package (Sievert, 2020). You can access the 2020 book that goes into more detail on `plotly`-style graphics at https://plotly-r.com/.

To use `ggplotly`, we wrap that function around a `ggplot` object and it will render it in an interactive fashion when the viewer hovers over individual plot components. There are also ways of making `plotly` graphs directly using `plot_ly` and that may prove easier for some things, for example for making interactive three-dimensional graphs.

Here are two examples that we worked with earlier converted into `ggplotly` objects that allow further interrogation of the information displayed:

```r
library(plotly)

spp_weight_boxplot <- surveys %>%
  ggplot(aes(x = genus, y = weight)) +
  geom_violin() +
  geom_jitter(color = "tomato", width = 0.2, alpha = 0.2) +
  scale_y_log10() +
  ## log (base 10) transforms the y-axis variable
  ## (helps to make the plot less skewed)
  labs(x = "",
       ## removes the y-axis label
       y = "log10-Weight") +
  coord_flip() + #Switches x and y axes
  theme(axis.text.y = element_text(size = 12),
        axis.text.x = element_text(size = 12),
        text = element_text(size = 16)) +
  theme_bw()

spp_count_plot <- yearly_counts %>%
  ggplot(aes(x = year, y = n, color = genus)) +
  geom_line() +
  labs(x = "Year",
       y = "Abundance",
       color = "Genus") +
  theme(axis.title.x = element_text(face = "bold", size = 12)) +
  theme_bw()
```

```r
ggplotly(spp_weight_boxplot);ggplotly(spp_count_plot)
```

One note about using `plotly` graphics in R-markdown is that they will not knit into word or PDF formats, only into HTML (in word or PDF, you are stuck with static images but you could incorporate a link to a website containing the interactive version of the plot). You can also interact with plots when working in markdown or running code in the console in RStudio. For presentations, you can also record a video of interactions with plot to remove some challenges of live presentations using this format. But if you are looking to wow your viewers in a digital format, need to dig into some details of what is displayed in a plot quickly, or when static graphics are limiting your story-telling ability, remember that there might be another option!

Happy plotting!

Montana State University R Workshops Team

These materials were adapted from materials generated by the Data Carpentries (https://datacarpentry.org/) and were originally developed at MSU by Dr. Allison Theobold. The workshop series is co-organized by the Montana State University Library and Social Data Collection and Analysis Services (Social Data) which is an MSU Core Facility and also part of the Data Science Core for Montana INBRE. Social Data is supported by Montana INBRE (National Institutes of Health, Institute of General Medical Sciences Grant Number P20GM103474).

Research related to the development of these workshops appeared in:

- Allison S. Theobold, Stacey A. Hancock & Sara Mannheimer (2021) Designing Data Science Workshops for Data-Intensive Environmental Science Research, *Journal of Statistics and Data Science Education*, 29:sup1, S83-S94, DOI: 10.1080/10691898.2020.1854636

The workshops for 2025 involve modifications of materials and are being taught by:

Greta Linse

- Greta Linse is the Facility Manager of Social Data Collection and Analysis Services (https://www.montana.edu/socialdata/) among other on campus roles. Greta has been teaching, documenting and working with statistical software including R and RStudio for over 10 years.

Sally Slipher

- Sally Slipher is a research statistician for Social Data. She has taught statistics in the past and uses R extensively (and sometimes other coding languages) to explore data and put together analyses.

Sara Mannheimer

- Sara Mannheimer is an Associate Professor and Data Librarian at Montana State University, where she helps shape practices and theories for curation, publication, and preservation of data. Her research examines the social, ethical, and technical issues of a data-driven world. She is the project lead for the MSU Dataset Search and the Responsible AI in Libraries and Archives project. Her 2024 book, Scaling Up, explores how data curation can address epistemological, ethical, and legal issues in qualitative data reuse and big social research.

The materials have also been modified and improved by:

- Dr. Mark Greenwood
- Harley Clifton
- Eliot Liucci
- Dr. Allison Theobold