R with ggplot2: Data Visualization

Stacey Borrego

1/22/2023

The goal for this workshop is to build on the fundamental R skills introduced in previous workshops and focus on data visualization. We will predominately use packages from tidyverse including the powerful and popular ggplot2.

Helpful Links

- Data Carpentry: Data Analysis and Visualization in R for Ecologists
- Tidyverse: R packages for Data Science
- R for Data Science by Hadley Wickham and Garret Grolemund
- Advanced R by Hadley Wickham
- ggplot2: Elegant graphics for data analysis by Hadley Wickham
- Posit Cheatsheets by Posit (formerly RStudio)
- The R Gallery by Kyle W. Brown
- Introduction to R by Douglas, Roos, Mancini, Cuoto, & Lusseau

---

Setting up a Project

Whether you are using RStudio Cloud or have downloaded the RStudio IDE on your own computer, we will be working with a project. Projects are a helpful way to keep all of your related documents organized as well as a convenient method to access all documents locally.

RStudio Cloud

- Log in to RStudio Cloud and click on the large blue button on the far right side that says **New Project**
- Select **New RStudio Project**
- Your workspace is now in a Project setting!
- Rename your Project at the top by clicking on *Untitled Project*

RStudio IDE on your computer:

- Open RStudio on your computer
- Click on **File > New Project**
- Click on **New Directory**

- Select **New Project**
- Provide a name for folder/directory your Project will live in, ie *demo-project*
- Select a location on your computer to create your Project – *Desktop* is fine for this workshop
- Click **Create Project**

Now, let's make sure we are all set! If you are in a Project you will have a distinct path when look at where we currently are. . . or our working directory.

```
getwd()
```

If you are on the cloud, your working directory should return: `"/cloud/project"`

If you are on your computer, your working directory should return something that looks like mine: `"/Users/stacey/Desktop/demo-project"`

Getting Data

Now that we are ready with our file system let's get some data to play with! First, let's stay organized and make a directory for the raw data. This can be done one of two ways.

- The first way is to use the buttons on the *Files* pane. Click *New Folder*, name it *data_raw*, hit *OK* and ta-da!
- The second way is by scripting in your *Console* pane using `dir.create()`.

```
dir.create("data_raw")
```

You should now see a directory listed in your *Files* pane name "data_raw".

It is imperative that this is setup correctly before moving on to the next step!

Downloading Data

To save our work for future reference, let's begin working in a script:

- Click on **File > New File > R Script**.
  - You can begin writing in this text document.
- To run the code you have written, hit the *RUN* button when your cursor is at the end of a segment of code that you wish to execute.
- Alternatively, you can type *Command/Control* (depending on Mac or Windows, respectively) + *ENTER*.

The data we will be using for this workshop investigates the animal species diversity and weights found within a study site. The file is a comma separated value (CSV) file.

To download the file, we will use the function `download.file()` and we will provide the arguments `url` - where to load the file from and `destfile` - where to save the file on your computer.

```
download.file(url = "https://ndownloader.figshare.com/files/2292169",
              destfile = "data_raw/portal_data_joined.csv")
```

If you click on the "data_raw" directory, you should now see a new file name "portal_data_joined.csv".

Reading the data

We now have the data but we can't access it yet! We need to load the data set in memory so we can start to analyze and visualize! There are many ways to do this but we will be using tools from the tidyverse package.

A package in R is a set of functions that allows us to expand the ability of R. Base R functions exist that come with the installation of the language. To get these extra tools, we will have to install the packages that we want using the function `install.packages()`.

The tidyverse package comes with a ton of subpackages including ggplot2. For installation, type `install.packages("tidyverse")` into your console.

```
install.packages("tidyverse")
```

Load the tidyverse package

```
library(tidyverse)
```

Now it is time to load the data! We will use the function from tidyverse `read_csv()` to load the data as a tibble, a modified version of a data frame. You can learn more about tibbles HERE.

```
mydata <- read_csv("data_raw/portal_data_joined.csv")
```

Let's take a look at what we are working with. To see the top 6 rows of the data use the function `head()`. To see the last rows, use `tail()`.

```
head(mydata)
```

```
## # A tibble: 6 x 13
##   record_id month   day  year plot_id speci~1 sex   hindf~2 weight genus species
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>   <chr>   <dbl>  <dbl> <chr> <chr>
## 1         1     7    16  1977       2 NL      M          32     NA Neot~ albigu~
## 2        72     8    19  1977       2 NL      M          31     NA Neot~ albigu~
## 3       224     9    13  1977       2 NL      <NA>       NA     NA Neot~ albigu~
## 4       266    10    16  1977       2 NL      <NA>       NA     NA Neot~ albigu~
## 5       349    11    12  1977       2 NL      <NA>       NA     NA Neot~ albigu~
## 6       363    11    12  1977       2 NL      <NA>       NA     NA Neot~ albigu~
## # ... with 2 more variables: taxa <chr>, plot_type <chr>, and abbreviated
## #   variable names 1: species_id, 2: hindfoot_length
```

```
tail(mydata)
```

```
## # A tibble: 6 x 13
##   record_id month   day  year plot_id speci~1 sex   hindf~2 weight genus species
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>   <chr>   <dbl>  <dbl> <chr> <chr>
## 1     26787     9    27  1997       7 PL      F          21     16 Pero~ leucop~
## 2     26966    10    25  1997       7 PL      M          20     16 Pero~ leucop~
## 3     27185    11    22  1997       7 PL      F          21     22 Pero~ leucop~
## 4     27792     5     2  1998       7 PL      F          20      8 Pero~ leucop~
## 5     28806    11    21  1998       7 PX      <NA>       NA     NA Chae~ sp.
## 6     30986     7     1  2000       7 PX      <NA>       NA     NA Chae~ sp.
## # ... with 2 more variables: taxa <chr>, plot_type <chr>, and abbreviated
## #   variable names 1: species_id, 2: hindfoot_length
```

When you first load the data it gives a short description of the data. You can also look at the dimensions of the data using the function `dim()`.

```
dim(mydata)
```

```
## [1] 34786    13
```

To open the data in the RStudio Data viewer, use the function `view()`. This will open up a new tab with the data in a setting that you can scroll through. **Note**: `view()` is only available when tidyverse is loaded. `View()` with a capital 'V' is available as a base R function.

```
view(mydata)
```

It is also helpful to see the names of the rows or columns of your data. Use either `rownames()` or `names()` to retrieve a list of these names.

```
rownames(mydata)
```

```
names(mydata)
```

Indexing and Subetting Data

We can select specific data from our data frame by subsetting. We need to specify rows first followed by columns in the form of `data_frame[row_index, column_index]`.

```
# Subset first row, first column
mydata[1, 1]

# Subset first row, all columns
mydata[1, ]

# Subset all rows, first column
mydata[, 1]
```

```
# Shortcut to get all rows, first column
mydata[1]

# Subset multiple rows and columns using vectors
mydata[c(1, 2, 3), c(5, 6)]

# Shortcut to subset vectors
mydata[1:3, 5:6]

# Single brackets return a data frame
mydata[1]

# Double brackets return a vector
mydata[[1]]

# Subset a column by its name
mydata["species_id"]

# Subset a column by it name and $ - returns a vector
mydata$species_id
```

Factors

Factors are so important for plotting and is usually the issue that makes or breaks your plot!

The data we are using today has several columns that are identified as integers or characters. Some of the character data is categorical which means that it can only take on a limited number of values. Categorical data are call factors and contain a pre-defined set of values called levels. Factors are stored as integers associated with labels and can be ordered or unordered. Many times categorical data will not be immediately coerced into factors, we must do this ourselves.

As an example, we will use the data in the "sex" column.

```
# Look at the first 6 rows of the data frame.
head(mydata)
```

```
## # A tibble: 6 x 13
##   record_id month   day  year plot_id speci~1 sex   hindf~2 weight genus species
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>   <chr>   <dbl>  <dbl> <chr> <chr>
## 1         1     7    16  1977       2 NL      M          32     NA Neot~ albigu~
## 2        72     8    19  1977       2 NL      M          31     NA Neot~ albigu~
## 3       224     9    13  1977       2 NL      <NA>       NA     NA Neot~ albigu~
## 4       266    10    16  1977       2 NL      <NA>       NA     NA Neot~ albigu~
## 5       349    11    12  1977       2 NL      <NA>       NA     NA Neot~ albigu~
## 6       363    11    12  1977       2 NL      <NA>       NA     NA Neot~ albigu~
```

```
## # ... with 2 more variables: taxa <chr>, plot_type <chr>, and abbreviated
## #   variable names 1: species_id, 2: hindfoot_length
```

```
# Determine what class the column "sex" currently is.
class(mydata$sex)
```

```
## [1] "character"
```

```
# Factor the values in the column "sex" and write over the values
# currently in the data frame.
mydata$sex <- factor(mydata$sex)

# Look at the first 6 rows of the data frame.
head(mydata)
```

```
## # A tibble: 6 x 13
##   record_id month   day  year plot_id speci~1 sex   hindf~2 weight genus species
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>   <fct>   <dbl>  <dbl> <chr> <chr>
## 1         1     7    16  1977       2 NL      M          32     NA Neot~ albigu~
## 2        72     8    19  1977       2 NL      M          31     NA Neot~ albigu~
## 3       224     9    13  1977       2 NL      <NA>       NA     NA Neot~ albigu~
## 4       266    10    16  1977       2 NL      <NA>       NA     NA Neot~ albigu~
## 5       349    11    12  1977       2 NL      <NA>       NA     NA Neot~ albigu~
## 6       363    11    12  1977       2 NL      <NA>       NA     NA Neot~ albigu~
## # ... with 2 more variables: taxa <chr>, plot_type <chr>, and abbreviated
## #   variable names 1: species_id, 2: hindfoot_length
```

```
# Use `summary()` to identify the different values in the column "sex".
summary(mydata$sex)
```

```
##     F     M  NA's
## 15690 17348  1748
```

```
# Determine the levels of the factored values.
# The levels are automatically ordered alphabetically.
levels(mydata$sex)
```

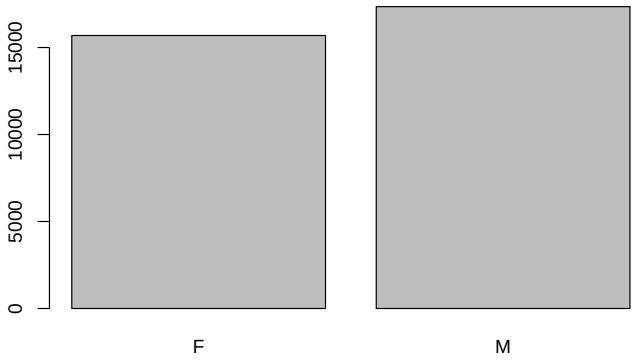
```
## [1] "F" "M"
```

```
nlevels(mydata$sex)
```

```
## [1] 2
```

When we plot the values we see the two factors listed, F and M. However, there was a good portion of the data without values. First, we'll make a copy of the data to play with. Then we will the missing data by using the function `addNA()`.

```
plot(mydata$sex)
```



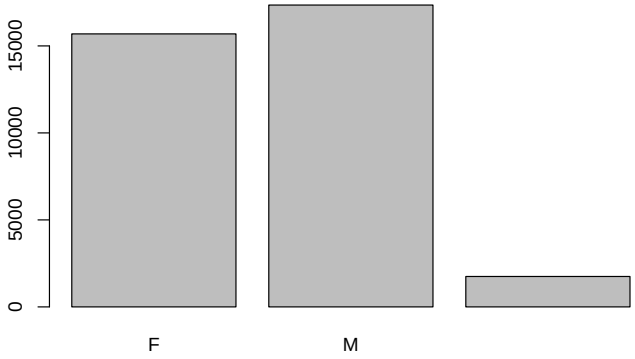

```
sex <- mydata$sex

levels(sex)
```

```
## [1] "F" "M"
```

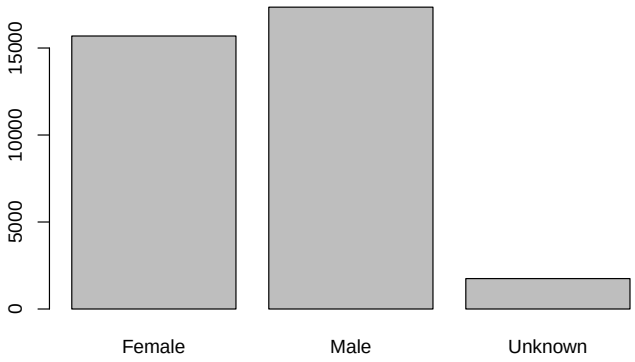
```
sex <- addNA(sex)
levels(sex)
```

```
## [1] "F" "M" NA
```

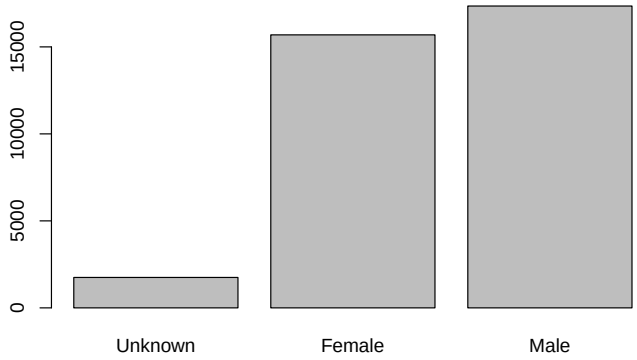
```
plot(sex)
```




```
levels(sex) <- c("Female", "Male", "Unknown")

plot(sex)
```



```r
ordered <- factor(sex,
                  levels = c("Unknown", "Female", "Male"))

plot(ordered)
```



Making Clean Data

The tidyverse contains the package dplyr, which contains tools to help you get your data ready for plotting. Here are a few common tools that you should know:

- `select()`: subset columns
- `filter()`: subset rows on conditions
- `mutate()`: create new columns by using information from other columns
- `group_by()` and `summarize()`: create summary statistics on grouped data
- `arrange()`: sort results
- `count()`: count discrete values

```r
# Select columns by name
select(mydata,
       species_id, sex, weight)
```

```
## # A tibble: 34,786 x 3
##    species_id sex   weight
##    <chr>      <fct>  <dbl>
##  1 NL         M         NA
##  2 NL         M         NA
##  3 NL         <NA>      NA
##  4 NL         <NA>      NA
##  5 NL         <NA>      NA
##  6 NL         <NA>      NA
##  7 NL         <NA>      NA
##  8 NL         <NA>      NA
##  9 NL         M        218
## 10 NL         <NA>      NA
## # ... with 34,776 more rows
```

```r
# Filter rows by values in the column "weight"
filter(mydata,
       weight < 5)
```

```
## # A tibble: 17 x 13
##    record~1 month   day  year plot_id speci~2 sex   hindf~3 weight genus species
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>   <fct>   <dbl>  <dbl> <chr> <chr>
##  1     4052     4     5  1981       3 PF      F          15      4 Pero~ flavus
##  2     7084    11    22  1982       3 PF      F          16      4 Pero~ flavus
##  3    28126     6    28  1998      15 PF      M          NA      4 Pero~ flavus
##  4     9909     1    20  1985      15 RM      F          15      4 Reit~ megalo~
##  5     9853     1    19  1985      17 RM      M          16      4 Reit~ megalo~
##  6     4290     4     6  1981       4 PF      <NA>       NA      4 Pero~ flavus
##  7    29906    10    10  1999       4 PP      M          21      4 Chae~ penici~
##  8     8736    12     8  1983      19 RM      M          17      4 Reit~ megalo~
##  9     9799     1    19  1985      19 RM      M          16      4 Reit~ megalo~
## 10     9794     1    19  1985      24 RM      M          16      4 Reit~ megalo~
## 11      218     9    13  1977       1 PF      M          13      4 Pero~ flavus
## 12     5346     2    22  1982      21 PF      F          14      4 Pero~ flavus
## 13     9937     2    16  1985      21 RM      M          16      4 Reit~ megalo~
## 14    10119     3    17  1985      10 RM      M          16      4 Reit~ megalo~
## 15     9790     1    19  1985      16 RM      F          16      4 Reit~ megalo~
## 16     9823     1    19  1985      23 RM      M          16      4 Reit~ megalo~
## 17    10439     5    24  1985       7 RM      M          16      4 Reit~ megalo~
## # ... with 2 more variables: taxa <chr>, plot_type <chr>, and abbreviated
## #   variable names 1: record_id, 2: species_id, 3: hindfoot_length
```

Pipes Pipes are a handy tool that allows you to combine functions and reduce intermediate objects. They take the output of one function and send it to the next function, processing the same data set in a single block of code. Pipes are achieved using this symbol: `%>%`. A shortcut to place this symbol in your code is **CONTROL/COMMAND + SHIFT + M**.

```r
# Filters data by weight and selects columns, prints to console
mydata %>%
  filter(weight < 5) %>%
  select(species_id, sex, weight)
```

```
## # A tibble: 17 x 3
##    species_id sex   weight
##    <chr>      <fct>  <dbl>
##  1 PF         F          4
##  2 PF         F          4
##  3 PF         M          4
##  4 RM         F          4
##  5 RM         M          4
```

```
##  6 PF         <NA>       4
##  7 PP         M          4
##  8 RM         M          4
##  9 RM         M          4
## 10 RM         M          4
## 11 PF         M          4
## 12 PF         F          4
## 13 RM         M          4
## 14 RM         M          4
## 15 RM         F          4
## 16 RM         M          4
## 17 RM         M          4
```

```
# Filters data by weight and selects columns, creates a new object
# called "data_small"
data_small <- mydata %>%
  filter(weight < 5) %>%
  select(species_id, sex, weight)

data_small
```

```
## # A tibble: 17 x 3
##    species_id sex   weight
##    <chr>      <fct>  <dbl>
##  1 PF         F          4
##  2 PF         F          4
##  3 PF         M          4
##  4 RM         F          4
##  5 RM         M          4
##  6 PF         <NA>       4
##  7 PP         M          4
##  8 RM         M          4
##  9 RM         M          4
## 10 RM         M          4
## 11 PF         M          4
## 12 PF         F          4
## 13 RM         M          4
## 14 RM         M          4
## 15 RM         F          4
## 16 RM         M          4
## 17 RM         M          4
```

```
# Filter out the missing values in the columns: weight, hindfoot_length, and sex
data_complete <- mydata %>%
  filter(!is.na(weight),
         !is.na(hindfoot_length),
```

```
         !is.na(sex))

head(data_complete)
```

```
## # A tibble: 6 x 13
##   record_id month   day  year plot_id speci~1 sex   hindf~2 weight genus species
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>   <fct>   <dbl>  <dbl> <chr> <chr>
## 1       845     5     6  1978       2 NL      M          32    204 Neot~ albigu~
## 2      1164     8     5  1978       2 NL      M          34    199 Neot~ albigu~
## 3      1261     9     4  1978       2 NL      M          32    197 Neot~ albigu~
## 4      1756     4    29  1979       2 NL      M          33    166 Neot~ albigu~
## 5      1818     5    30  1979       2 NL      M          32    184 Neot~ albigu~
## 6      1882     7     4  1979       2 NL      M          32    206 Neot~ albigu~
## # ... with 2 more variables: taxa <chr>, plot_type <chr>, and abbreviated
## #   variable names 1: species_id, 2: hindfoot_length
```

```
dim(data_complete)
```

```
## [1] 30676    13
```

```
names(data_complete )
```

```
##  [1] "record_id"       "month"           "day"             "year"
##  [5] "plot_id"         "species_id"      "sex"             "hindfoot_length"
##  [9] "weight"          "genus"           "species"         "taxa"
## [13] "plot_type"
```

We are interested in plotting changes in species abundances over time so we need to modify the data for common species only. To do this, we will first count all the species and remove those with observations less than 50. Second we will filter our data set to keep only the common species.

```
# First: count each observation of each species and remove those under 50
species_counts <- data_complete %>%
  count(species_id) %>%
  filter(n > 50)

species_counts
```

```
## # A tibble: 14 x 2
##    species_id     n
##    <chr>      <int>
##  1 DM          9727
##  2 DO          2790
##  3 DS          2023
##  4 NL          1045
##  5 OL           905
```

```
##  6 OT           2081
##  7 PB           2803
##  8 PE           1198
##  9 PF           1469
## 10 PM            835
## 11 PP           2969
## 12 RF             73
## 13 RM           2417
## 14 SH            128
```

```
# Second: Extract the observations of the common species
data_complete <- data_complete %>%
  filter(species_id %in% species_counts$species_id)

# If done correctly the dimensions of the data frame is 30463 rows by 13 columns
dim(data_complete)
```

```
## [1] 30463    13
```

After all your hard work preparing your data, you may want to save it. You can write it to a CSV file and save it using the function `write_csv()`

```
# First: create a new directory called "data"
dir.create("data")

# Second: save the csv in the new directory
write_csv(data_complete,
          file = "data/data_complete.csv")
```

Plotting

We will be using ggplot2 to create beautiful plots from our data. Graphics in ggplot2 are built in layers, allowing for customization and flexibility.

The basic template to build a plot is as follows:

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

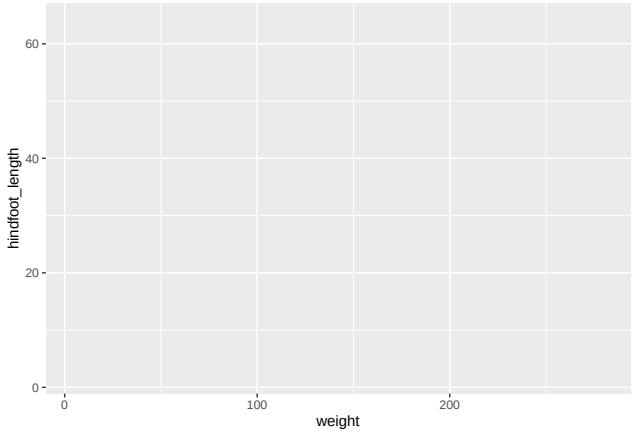
The `ggplot()` function is first directed to the data frame that it will be referring to.

```
ggplot(data = data_complete)
```

The aesthetic function `aes()` selects the variables to be plotted and specifies how to present them in the graph – as x and y positions or characteristics such as size, shape, color.

Once the `ggplot()` function has been completed, you can save it as an object and refer to it later. For demonstaration purposes I will continue to show the full contents of the `ggplot()` function.

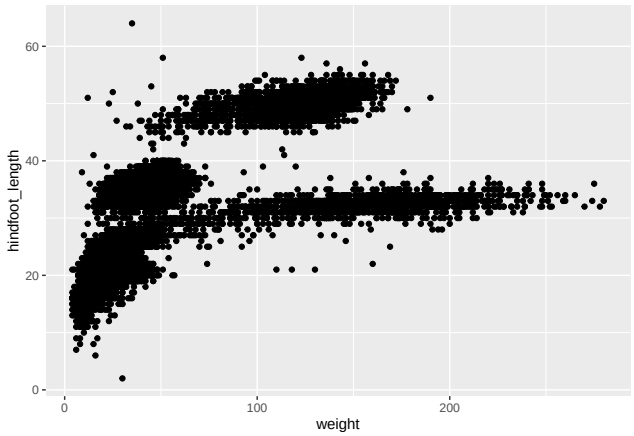
```
ggplot(data = data_complete,
       mapping = aes(x = weight, y = hindfoot_length))
```



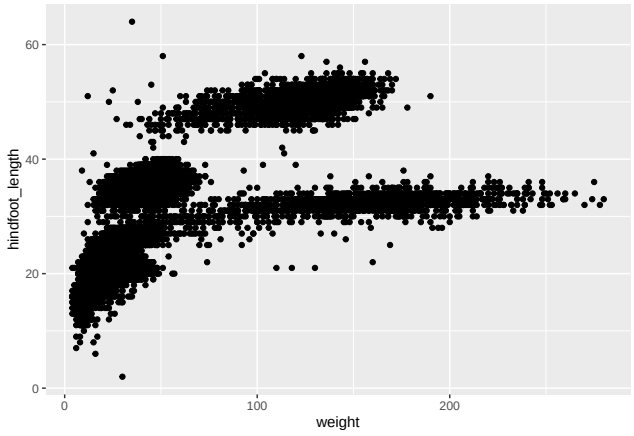

```
# Save the ggplot2 object
data_plot <- ggplot(data = data_complete,
       mapping = aes(x = weight, y = hindfoot_length))
```

Lastly, geoms are the way the graphs are represented such as line, bar, and scatter plots.

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




```
# Referring to the ggplot() object
data_plot +
  geom_point()
```



Each geom understands certain aesthetics. To learn more about a particular geom, you can use `?geom_<TYPE>` to learn more. To learn more about available aesthetics, HERE is a good article.

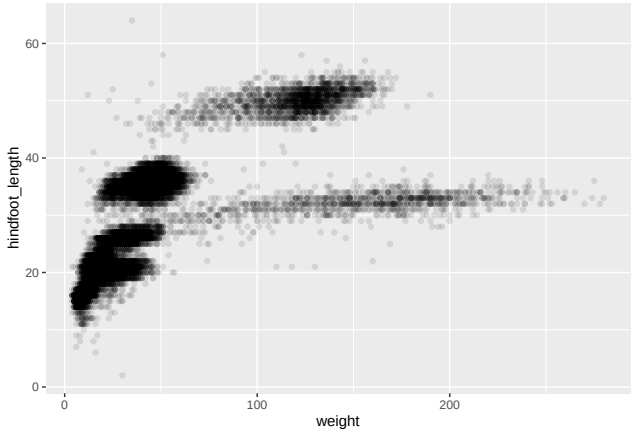
Geom_point can be modified with the following aesthetics:

- alpha - the opacity of a geom. Values of alpha range from 0 to 1, with lower values corresponding to more transparent colors.

- color - colors the line or stroke of each point
- fill - fills the point
- group - controls which rows of the data get grouped together
- shape - controls the shape of the point
- size - controls the size of the filled part of point
- stroke - the thickness of each point on a scatterplot

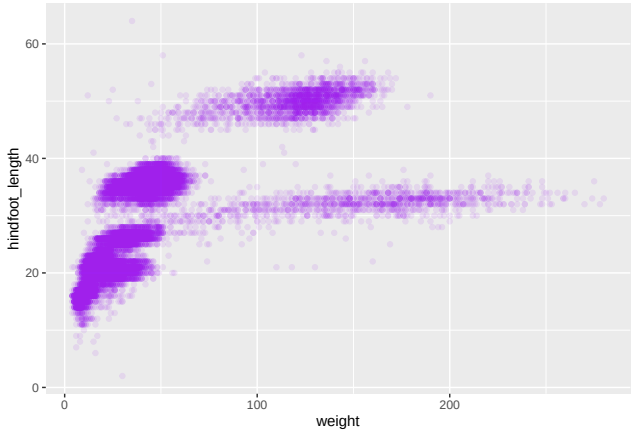
Sometimes it helps to understand the data by reducing the opacity of each individual point. Use `alpha` within the `geom_point()` function to adjust the opacity with a number between 0 and 1.

```
# Reducing the opacity with alpha
ggplot(data = data_complete,
       mapping = aes(x = weight, y = hindfoot_length)) +
  geom_point(alpha = 0.1)
```



Here we can change the color of the data points using `color` within the `geom_point()` function.

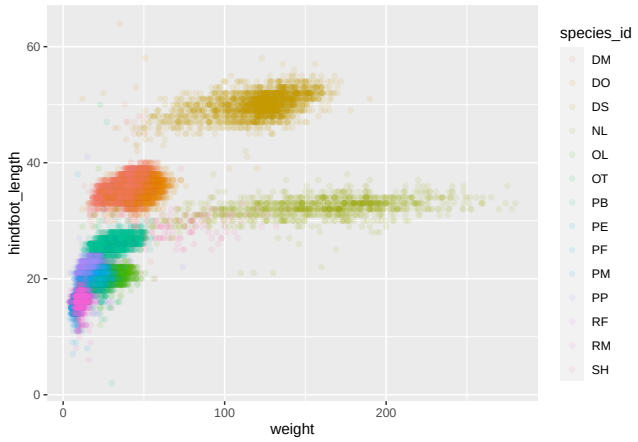
```
# Changing the color and opacity
ggplot(data = data_complete,
       mapping = aes(x = weight, y = hindfoot_length)) +
  geom_point(alpha = 0.1,
             color = "purple")
```



It can helpful to separate groups within the dataset with different colors. Using the `aes()` function within `geom_point()`, we can assign default colors to each group identified within "species_id". A legend is automatically placed in the plotting area.

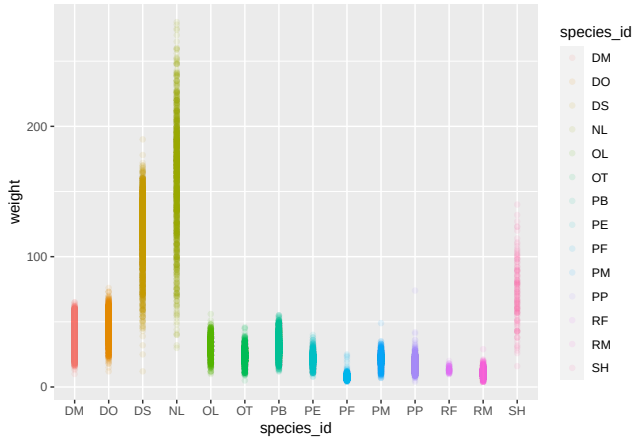
```
# Changing the color by "species_id"
ggplot(data = data_complete,
       mapping = aes(x = weight, y = hindfoot_length)) +
  geom_point(alpha = 0.1,
```

```
            aes(color = species_id))
```



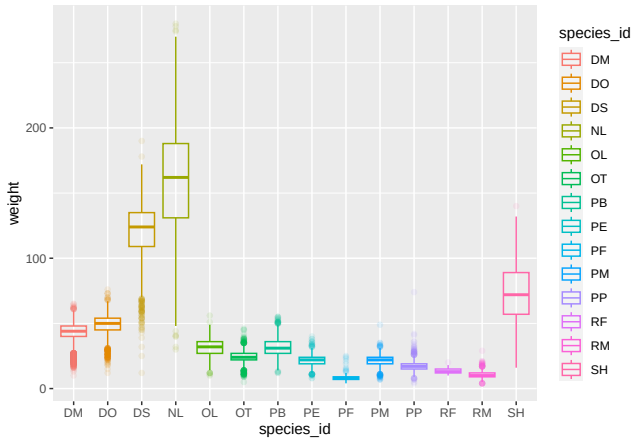
If we want to change the data in the plot, we can indicate a different column of data to use in the `ggplot()` function in the `mapping` argument. Let's change `x` to "species_id" and `y` to "weight".

```
# Changing the color by "species_id"
ggplot(data = data_complete,
       mapping = aes(x = species_id, y = weight)) +
  geom_point(alpha = 0.1,
             aes(color = species_id))
```



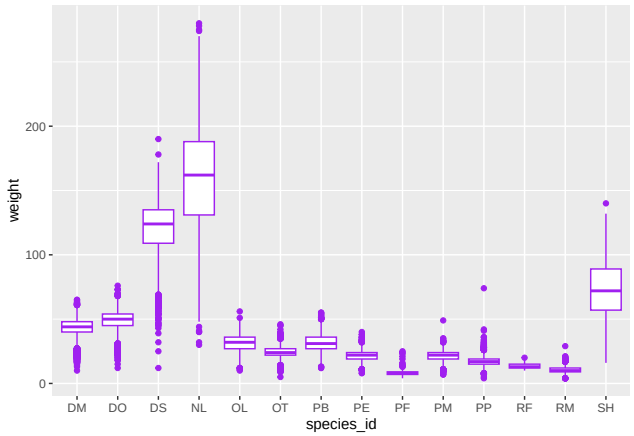
Let's change the plot to something more informative. A boxplot measures the points for each group and plots the median (middle line), the 25th percentile, the 75th percentile, and 1.5x the inter-quartile range (IQR).

```
# Changing type of plot by using `geom_boxplot()`
ggplot(data = data_complete,
       mapping = aes(x = species_id, y = weight)) +
  geom_boxplot(alpha = 0.1,
               aes(color = species_id))
```



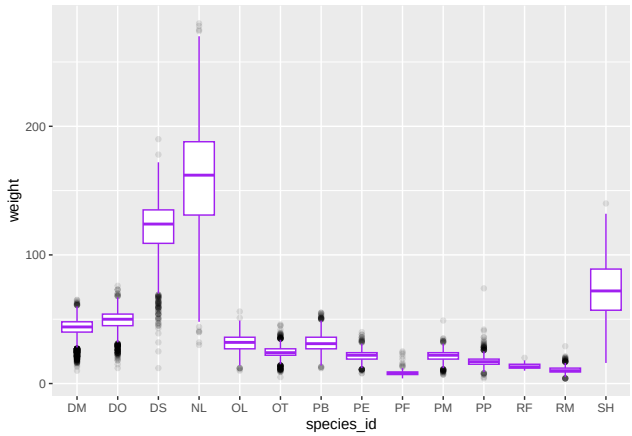
Since the species are labeled we don't need to color code the graph but you can if you want. Here we will make all boxplots purple with a white fill.

```
# Changing type of plot by using `geom_boxplot()`
ggplot(data = data_complete,
       mapping = aes(x = species_id, y = weight)) +
  geom_boxplot(color = "purple",
               fill = "white")
```



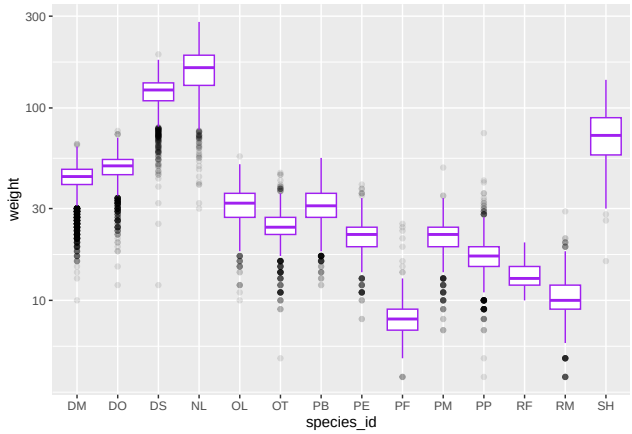
To distinguish the outliers, we can modify the opacity and color.

```
# Changing colors and characteristics of the outliers
# Use `?geom_boxplot` to learn more
ggplot(data = data_complete,
       mapping = aes(x = species_id, y = weight)) +
  geom_boxplot(color = "purple",
               fill = "white",
               outlier.color = "black",
               outlier.alpha = 0.1)
```



To help visualize the plot differently, we can modify the y-axis to a log10. This allows us to see the smaller changes and still acknowledge the larger values within the space of the plot.

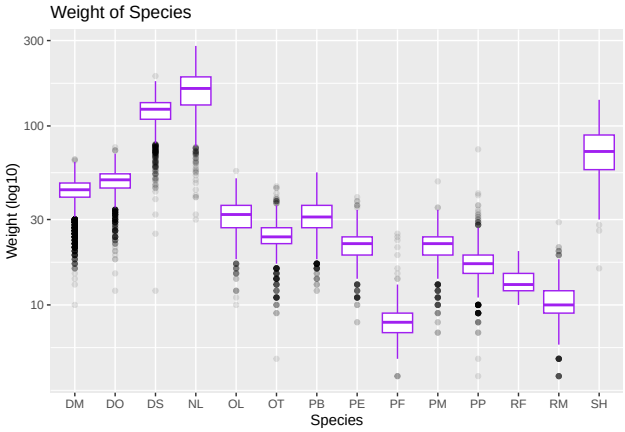
```
# Modifying the y-axis to be on a log10 scale
# Use `?geom_boxplot` to learn more
ggplot(data = data_complete,
       mapping = aes(x = species_id, y = weight)) +
  geom_boxplot(color = "purple",
               fill = "white",
               outlier.color = "black",
               outlier.alpha = 0.1) +
  scale_y_log10()
```



Next, we can add labels, titles, and subtitles to our plot using the `labs()` function.

```
# Changing the labels
# Use `?geom_boxplot` to learn more
ggplot(data = data_complete,
       mapping = aes(x = species_id, y = weight)) +
  geom_boxplot(color = "purple",
               fill = "white",
               outlier.color = "black",
```

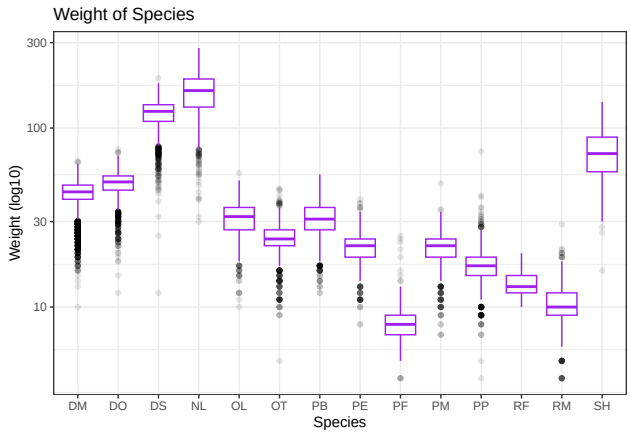
```
              outlier.alpha = 0.1) +
  scale_y_log10() +
  labs(title = "Weight of Species",
       y = "Weight (log10)",
       x = "Species")
```



We can change the look of our plot by using pre-loaded themes. You can find a list of available themes HERE. It is important that the theme is the last layer in your plot, otherwise it will be hidden and you cannot see it.

A common and simple theme is `theme_bw()`.

```
# Changing the theme to a simple white background
# Use `?geom_boxplot` to learn more
ggplot(data = data_complete,
       mapping = aes(x = species_id, y = weight)) +
  geom_boxplot(color = "purple",
               fill = "white",
               outlier.color = "black",
               outlier.alpha = 0.1) +
  scale_y_log10() +
  labs(title = "Weight of Species",
       y = "Weight (log10)",
       x = "Species") +
  theme_bw()
```



Time Series

For these plots, we will look at the counts per year of each genus. First, we group the data by year then by genus.

```
annual <- data_complete %>%
  count(year, genus)

head(annual)
```

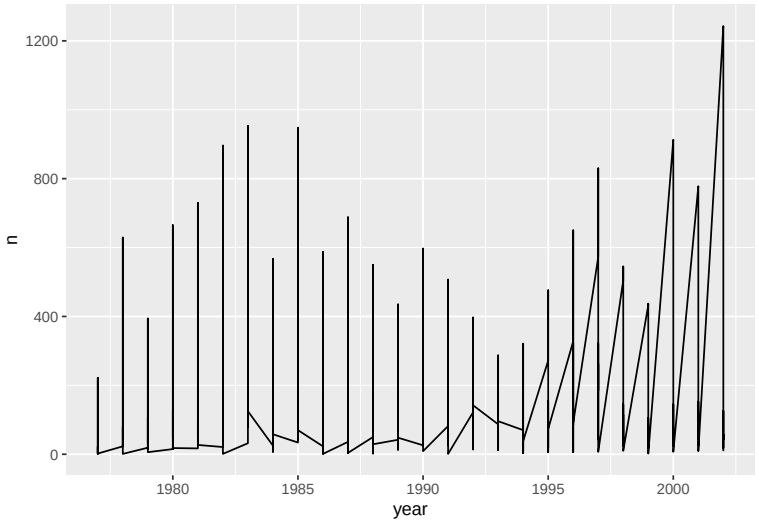
```
## # A tibble: 6 x 3
##    year genus              n
##   <dbl> <chr>          <int>
## 1  1977 Chaetodipus        3
## 2  1977 Dipodomys        222
## 3  1977 Onychomys          1
## 4  1977 Perognathus       22
## 5  1977 Peromyscus         2
## 6  1977 Reithrodontomys    2
```

```
tail(annual)
```

```
## # A tibble: 6 x 3
##    year genus              n
##   <dbl> <chr>          <int>
## 1  2002 Neotoma           42
## 2  2002 Onychomys        126
## 3  2002 Perognathus       18
## 4  2002 Peromyscus        58
## 5  2002 Reithrodontomys   20
## 6  2002 Sigmodon           9
```

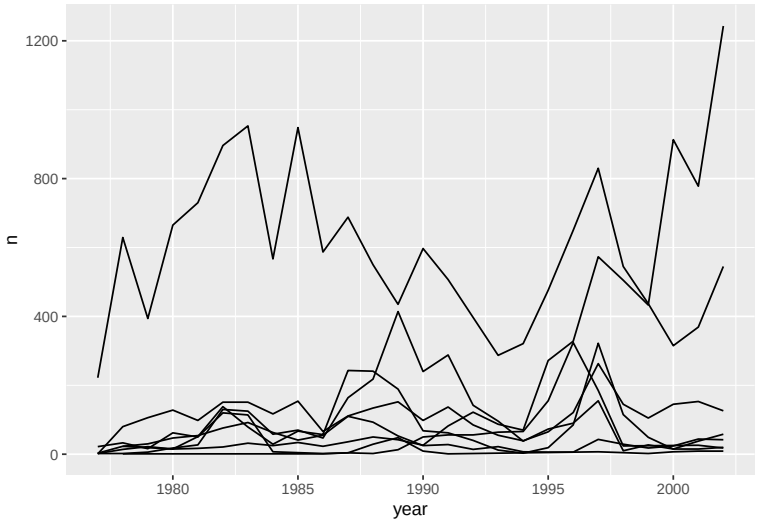
Now we can plot the data over time. However, the plot doesn't look as we hoped. Our goal is to see the trend over time for each genus. The current plot looks at the total abundance over time.

```
ggplot(data = annual,
       mapping = aes(x = year, y = n)) +
  geom_line()
```



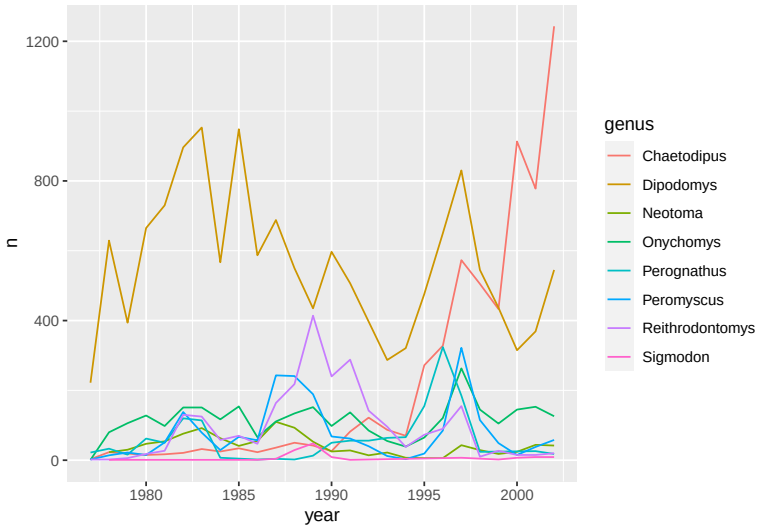
To group the data by genus so we can see the data plotted over time, we have to assign the `group` argument within the `aes()` function in our ggplot. Now you will see several lines changing over time but without any color or legend this plot is useless.

```
ggplot(data = annual,
       mapping = aes(x = year, y = n, group = genus)) +
  geom_line()
```



Another option to group the data but also assign colors is to use the argument `color` within `aes()`. This will automatically print a legend in the plot area.

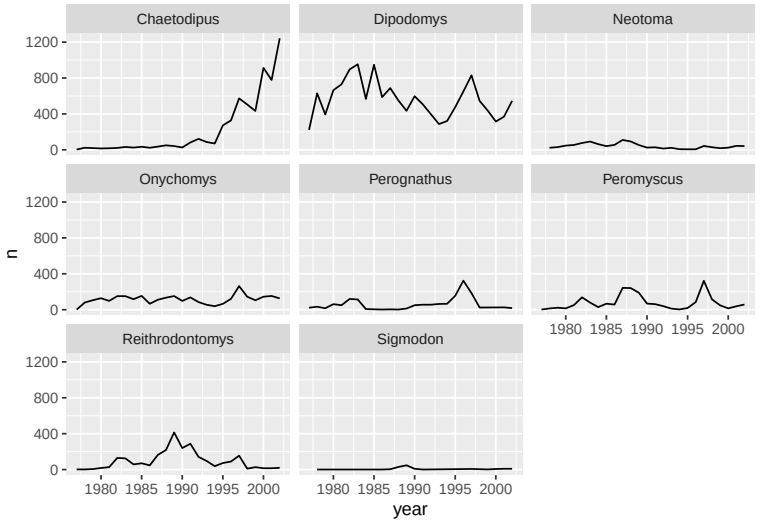
```
ggplot(data = annual,
       mapping = aes(x = year, y = n, color = genus)) +
  geom_line()
```



Faceting

Sometime the data can look too complex and needs to be separated for clarity. Faceting is a helpful technique that splits one plot into multiple plots based on a factor. To use faceting, we add on an additional layer to our plot and separate the data by "genus".

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



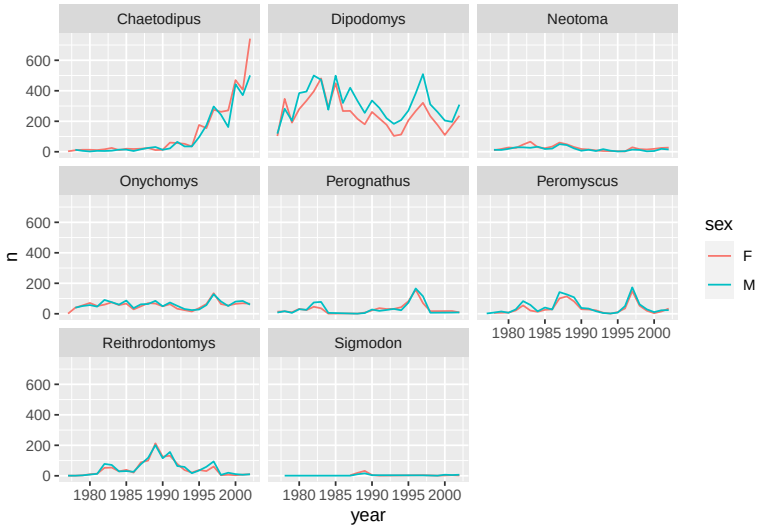
To gain more insight about the abunance change over time, we can group our data by sex in addition to year and genus. To do this, we just add our additional group to the function `count()`.

```
annual_sex <- data_complete %>%
  count(year, genus, sex)

head(annual_sex)
```

```
## # A tibble: 6 x 4
##    year genus       sex       n
##   <dbl> <chr>       <fct> <int>
## 1  1977 Chaetodipus F         3
## 2  1977 Dipodomys   F       103
## 3  1977 Dipodomys   M       119
## 4  1977 Onychomys   F         1
## 5  1977 Perognathus F        14
## 6  1977 Perognathus M         8
```

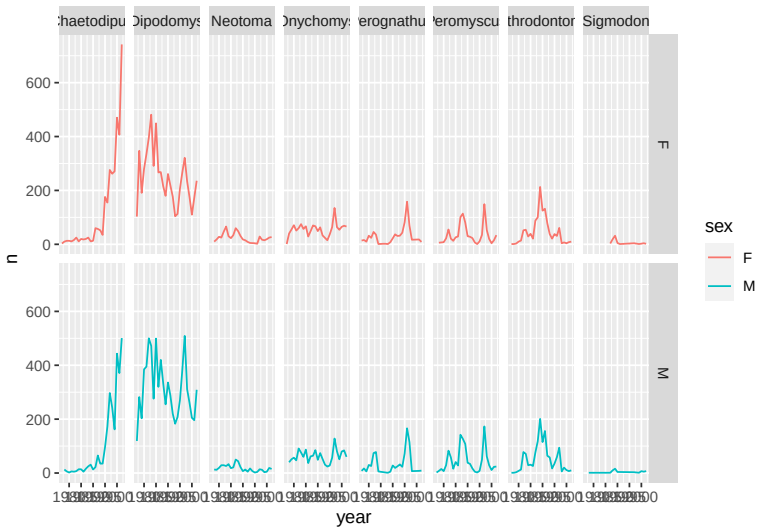
Now we reassign the `data` argument in the `ggplot()` function and add the argument `color` to the `aes()` function. This will produce a faceted plot with two different colored lines in each plot.

```
ggplot(data = annual_sex,
       mapping = aes(x = year, y = n, color = sex)) +
  geom_line() +
  facet_wrap(facets = vars(genus))
```



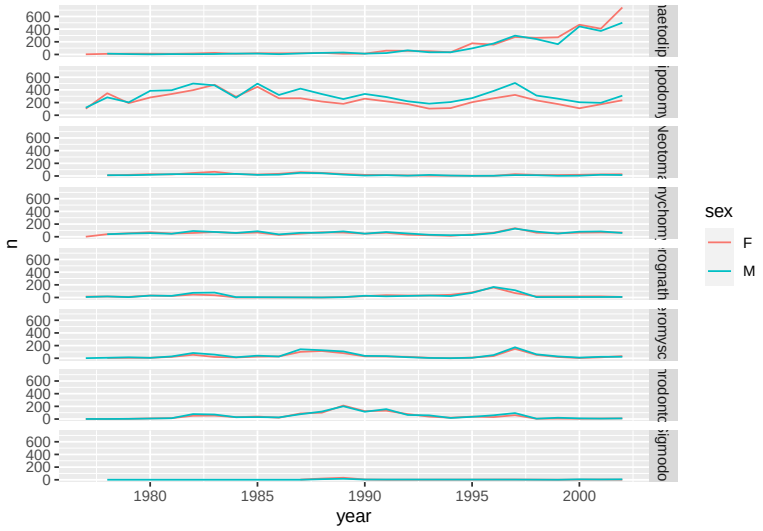
You can further customize the plot by indicating factors for rows and columns within the `facet_grid()` function. Here we will separate the data with the factor "sex" in each row and "genus" in the columns.

```
ggplot(data = annual_sex,
       mapping = aes(x = year, y = n, color = sex)) +
  geom_line() +
  facet_grid(row = vars(sex), cols = vars(genus))
```



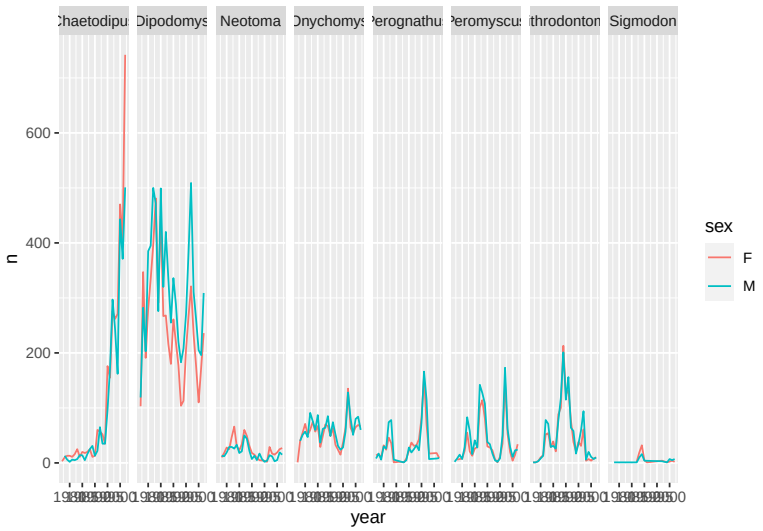
By only indicating a factor for rows, you can limit the look of your plot to be in rows.

```
ggplot(data = annual_sex,
       mapping = aes(x = year, y = n, color = sex)) +
  geom_line() +
  facet_grid(row = vars(genus))
```



Similarily, you can have your data faceted by columns only.

```
ggplot(data = annual_sex,
       mapping = aes(x = year, y = n, color = sex)) +
  geom_line() +
  facet_grid(cols = vars(genus))
```



Exporting Plots

We can easily save our plots using the `ggsave()` function. This will allow you to customize the dimensions (`width`, `height`) and resolution (`dpi`) of your plot with a few simple arguments.

You must save your plot as object so that it can be used as an argument to export.

```
myplot <- ggplot(data = annual_sex,
                 mapping = aes(x = year, y = n, color = sex)) +
  geom_line() +
  facet_grid(cols = vars(genus))
```

Create a new directory called "figures" and export your plot to the new directory.

```
dir.create("figures")

ggsave("figures/myplot_file.pdf", myplot, width = 15, height = 10)
```

Arranging Plots with Patchwork

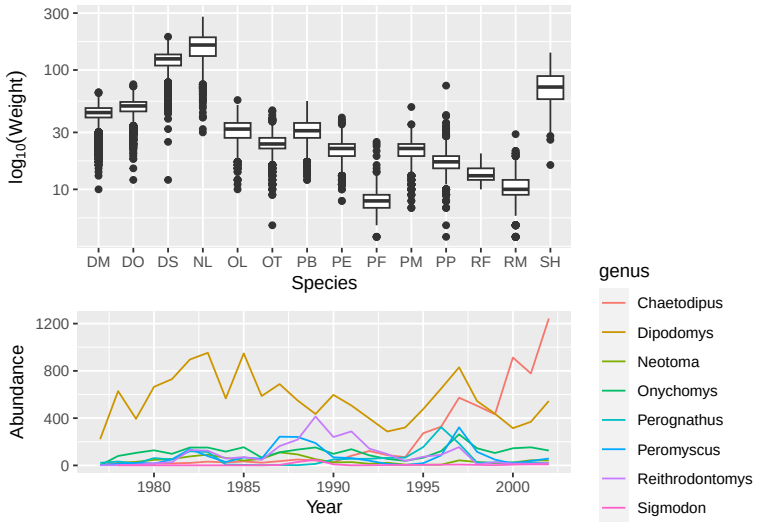
If you are publishing figures, it is helpful to combine multiple plots in a single figure. We can use the package `patchwork` to do this. First it must be installed and loaded into our session.

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

```
library(patchwork)
```

```
# Making the first plot and saving it as an object
plot_weight <- ggplot(data = data_complete,
                      aes(x = species_id, y = weight)) +
  geom_boxplot() +
  labs(x = "Species",
       y = expression(log[10](Weight))) +
```

```
  scale_y_log10()

# Making the second plot and saving it as an object
plot_count <- ggplot(data = annual,
                     aes(x = year, y = n, color = genus)) +
  geom_line() +
  labs(x = "Year",
       y = "Abundance")

# Calling both plots at the same time
plot_weight / plot_count + plot_layout(heights = c(3, 2))
```



These types of figures can be exported just as single plots. Just save the combined figure as an object and refer to it within the `ggsave()` function.

```
plot_combined <- plot_weight / plot_count + plot_layout(heights = c(3, 2))

ggsave("figures/plot_combined.png", plot_combined, width = 10, dpi = 300)
```