Data Analysis and Visualization

Stacey Borrego

11/3/2021

Helpful Links

- Data Carpentry: Data Analysis and Visualization in R for Ecologists
- Tidyverse: R packages for Data Science
- Advanced R by Hadley Wickham
- R packages by Hadley Wickham
- ggplot2: Elegant graphics for data analysis by Hadley Wickham

Data Carpentry: Introduction to R

https://datacarpentry.org/R-ecology-lesson/01-intro-to-r.html

Simple math In the console section of Rstudio, type in the following expressions. Hit `ENTER` after each expression.

```
3 + 5
12 / 7
3 * 4
8 ^ 2
log(8)
```

Creating objects in R

Working with variables/objects Assign values to a variable/object

- Assignment operator (<-)
- Cannot start with numbers
- Case sensitive
- Avoid function names
- Avoid dots (.)
- Be consistent in the styling of your code
  - R Style Guide
  - Styles can include "lower_snake", "UPPER_SNAKE", "lowerCamelCase", "UpperCamelCase", etc
  - We will use "lower_snake" for readibility during this workshop

```
weight_kg <- 55
```

Print the value of an object

```
(weight_kg <- 55)
weight_kg
```

Simple math with objects

```
2.2 * weight_kg
```

Reassign object's value

```
weight_kg <- 57.5
weight_kg
2.2 * weight_kg
```

Assign a new object a value containing another object

```
weight_lb <- 2.2 * weight_kg
weight_lb
```

Saving code in a script

- New script
  - File > New File > R Script (`Ctrl` + `Shift` + `N`)

- Save the file
  - File > Save (`Ctrl` + `S`)

Running code in a script

- Select `Run` button at the top of the script window
- Select code or move cursor to the line to be run and type `Ctrl` + `ENTER`

```
2 + 2
```

Add comments in a script The comment character in R is #. Anything to the right of a # in a script will be ignored by R Comment out a section by selecting and pressing `Ctrl` + `Shift` + `C`

```
#This is a comment.
#The sum of 2 + 2
2 + 2
```

Functions and arguments

- Functions are scripts that automate more commands
- Functions are available in base R, by importing packages, or by making them yourself
- Usually require one or more inputs called **arguments**
- Functions typically return a **value**

```
#This is a function call
sqrt(10)
weight_kg <- sqrt(10)
weight_kg
```

- Information about functions can be found in the `Help` window by typing `?` and the function name. Example: `?round`
- You can also use the function `args()` to see the arguments of a function

```
round(3.14159)
args(round)
round(3.14159, digits = 2)
```

Vectors and data types

- A vector is composed by a series of values, which can be either numbers or characters
  - Quotes must surround characters.
  - In a vector, all of the elements are the same type of data
- We can assign a series of values to a vector using the `c()` function.

```
#Numerical vector
weight_g <- c(50, 60, 65, 82)
weight_g

#Character vector
animals <- c("mouse", "rat", "dog")
animals
```

Explore the content of vector There are many functions that allow you to inspect the content of a vector. * `length()` tells you how many elements are in a particular vector * `class()` indicates what kind of object you are working with * `str()` provides an overview of the structure of an object and its elements.

```
#Length
length(weight_g)
length(animals)

#Class
class(weight_g)
class(animals)

#Structure
str(weight_g)
str(animals)
```

```
# Add to the end of the vector
weight_g <- c(weight_g, 90)
# Add to the beginning of the vector
weight_g <- c(30, weight_g)
# Inspect the modified vector
weight_g
```

Add elements to a vector

Data Types An atomic vector is the simplest R data type and is a linear vector of a single type.

There are four common types of atomic vectors: character, numeric or double, logical, and integer. There are two rare types that will not be discussed: complex and raw.

- `character`
- `numeric` or `double`
- `logical` for TRUE and FALSE (the boolean data type)
- `integer` for integer numbers (e.g., 2L, the L indicates to R that it's an integer)

- `complex` to represent complex numbers with real and imaginary parts (e.g., 1 + 4i)
- `raw` for bitstreams

```
typeof(weight_g)
typeof(animals)
```

Data Structures Vectors are one of the many data structures that R uses.

- lists (list)
- matrices (matrix)
- data frames (data.frame)
- factors (factor)
- arrays (array)

Coercion All elements of an atomic vector must be the same type, so when you attempt to combine different types they will be coerced to the most flexible type. Types from least to most flexible are: logical, integer, double, and character.

```
num_char <- c(1, 2, 3, "a")
typeof(num_char)
str(num_char)

num_logical <- c(1, 2, 3, TRUE, FALSE)
typeof(num_logical)
str(num_logical)

char_logical <- c("a", "b", "c", TRUE)
typeof(char_logical)
str(char_logical)

tricky <- c(1, 2, 3, "4")
typeof(tricky)
str(tricky)
```

Conditional subsetting

To extract one or several values from a vector, provide one or several indices in square brackets.

```
animals <- c("mouse", "rat", "dog", "cat")

animals[2]

animals[c(3, 1)]

animals[1:3]

more_animals <- animals[c(1, 4, 1, 3, 1, 2)]
more_animals
```

Subset by using a logical vector. TRUE will select the element with the same index, while FALSE will not.

```
weight_g <- c(21, 34, 39, 54, 55)

weight_g[c(TRUE, FALSE, FALSE, TRUE, TRUE)]

weight_g > 50

weight_g[weight_g > 50]
```

Combine multiple tests using `&` (both conditions are true, "AND") or | (at least one of the conditions is true, "OR").

Helpful operators

- `>` greater than
- `<` less than

- `<=` less than or equal to
- `==` equal to, test for numerical equality between the left and right hand sides

```
weight_g

weight_g[weight_g > 30 & weight_g < 50]

weight_g[weight_g <= 30 | weight_g == 55]

weight_g[weight_g >= 30 & weight_g == 21]
```

Search for strings in a vector

```
animals <- c("mouse", "rat", "dog", "cat", "cat")
animals

animals[animals == "cat" | animals == "rat"]

animals %in% c("rat", "cat", "dog", "duck", "goat", "bird", "fish")

animals[animals %in% c("rat", "cat", "dog", "duck", "goat", "bird", "fish")]
```

Missing data

- Missing data are represented in vectors as NA.
- You can add the argument `na.rm = TRUE` to calculate the result as if the missing values were removed (rm stands for **rem**ove)
- `is.na()` identifies missing elements
- `na.omit()` returns the object with incomplete cases removed
- `complete.cases()` returns a logical vector indicating which cases are complete

```
heights <- c(2, 4, 4, NA, 6)

mean(heights)
max(heights)

mean(heights, na.rm = TRUE)
max(heights, na.rm = TRUE)
```

```
heights

heights[is.na(heights)]
heights[!is.na(heights)]

na.omit(heights)

heights[complete.cases(heights)]
```

Data Carpentry: Starting with Data

https://datacarpentry.org/R-ecology-lesson/02-starting-with-data.html

Downloading the data

This is what the data will look like:

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 experimental plot of land
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
genus	genus of animal
species	species of animal
taxon	e.g. Rodent, Reptile, Bird, Rabbit

Before downloading data, ensure you are in the right place and directories are set up properly

```
# Returns the working directory
getwd()

# Provide a path as an argument to set the new working directory
setwd("/cloud/project")

# Provide a path and name for a new directory - absolute path
dir.create("/cloud/project/data_raw")
# Or if your working directory is already in the correct location you can provide the new directory nam
dir.create("data_raw")
```

- `download.file()` downloads the CSV file that contains the survey data

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

- `read.csv` is a base R function but today we will be using functions from the collection of R packages bundled up as one large package – Tidyverse
- `read_csv()` loads the content of the CSV file into R

```
# First load Tidyverse into your session
# if you have not installed it, use the following code:
# install.packages("tidyverse")
library(tidyverse)

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

Open the dataset in RStudio's Data Viewer

```
view(surveys)
```

Data Frames

https://datacarpentry.org/R-ecology-lesson/02-starting-with-data.html#What_are_data_frames

When we load the data into R, it is stored as an object of class tibble, which is a special kind of data frame - learn more about tibbles HERE.

A data frame is the representation of data in the format of a table where the columns are vectors that all have the same length. Because columns are vectors, each column must contain a single type of data (e.g., characters, integers, factors).

For example, here is an example data frame comprising a numeric, character, and logical vector.

```
##   numeric character logical
## 1       1         S    TRUE
## 2       7         A   FALSE
## 3       3         U    TRUE
```

Inspect the data

```
# See the first few rows of the data
head(surveys)
tail(surveys)

# See a specific number of rows of the data
head(surveys, n=10)
tail(surveys, n=10)
```

- Size:
  - `dim(surveys)` - returns a vector with the number of rows in the first element, and the number of columns as the second element (the dimensions of the object)
  - `nrow(surveys)` - returns the number of rows
  - `ncol(surveys)` - returns the number of columns
- Content:
  - `head(surveys)` - shows the first 6 rows
  - `tail(surveys)` - shows the last 6 rows
- Names:
  - `names(surveys)` - returns the column names (synonym of colnames() for data.frame objects)
  - `rownames(surveys)` - returns the row names
- Summary:
  - `str(surveys)` - structure of the object and information about the class, length and content of each column
  - `summary(surveys)` - summary statistics for each column

Indexing and subsetting data frames

A data frame has rows and columns in 2 dimensions. To extract specific data, specify the "coordinates" in `[]` indicating row numbers first followed by column numbers.

```
dim(surveys)

# Extract specific values by specifying row and column indices
# in the format:
# data_frame[row_index, column_index]

# Extract the first row and first column from surveys
surveys[1, 1]

# Extract the first row and sixth column
surveys[1, 6]
```

```r
# To select all columns, leave the column index blank
# Extract the first row and all columns
surveys[1, ]

# Extract all rows and the first column
surveys[, 1]

# An even shorter way to select first column across all rows:
surveys[1] # No comma!

# Select multiple rows or columns with vectors
# Extract the first three rows of the 5th and 6th column
surveys[c(1, 2, 3), c(5, 6)]

# We can use the : operator to create those vectors
surveys[1:3, 5:6]

# This is equivalent to head(surveys)
surveys[1:6, ]

# As we've seen, when working with tibbles
# subsetting with single square brackets ("[]") always returns a data frame.
# If you want a vector, use double square brackets ("[[]]")

# For instance, to get the first column as a vector:
surveys[[1]]

# To get the first value in our data frame:
surveys[[1, 1]]
```

```r
# The whole data frame, except the first column
surveys[, -1]

surveys[-(7:nrow(surveys)), ]
```

Data frames can be subset by calling their column names directly

```r
# Single brackets returns a data frame
surveys["species_id"]
surveys[, "species_id"]

# Double brackets returns a vector:
surveys[["species_id"]]

# Use the $ operator with column names to return a vector
surveys$species_id
```

Factors

https://datacarpentry.org/R-ecology-lesson/02-starting-with-data.html#Factors

When we did str(surveys) we saw that several of the columns consist of integers. The columns genus, species, sex, plot_type, ... however, are of the class character. Arguably, these columns contain categorical data, that is, they can only take on a limited number of values.

R has a special class for working with categorical data, called factors. Once created, factors can only contain

a pre-defined set of values, known as levels. Factors are stored as integers associated with labels and they can be ordered or unordered. While factors look (and often behave) like character vectors, they are actually treated as integer vectors by R. So you need to be very careful when treating them as strings.

When importing a data frame with read_csv(), the columns that contain text are not automatically coerced (=converted) into the factor data type, but once we have loaded the data we can do the conversion using the factor() function

```
head(surveys)
surveys$sex <- factor(surveys$sex)
head(surveys)

summary(surveys$sex)
```

By default, R always sorts levels in alphabetical order. In the example below, R will assign 1 to the level "female" and 2 to the level "male" (because f comes before m, even though the first element in this vector is "male").

```
sex <- factor(c("male", "female", "female", "male"))

# Different ways to see the levels of factored data
sex
levels(sex)
nlevels(sex)
```

Reorder factors

```
sex <- factor(sex, levels = c("male", "female"))

sex
levels(sex)
nlevels(sex)
```

Converting Factors Sometimes it is necessary to convert data from one type to another

```
sex

as.character(sex)
sex_char <- as.character(sex)
sex_char

as.numeric(sex)
```

```
year_fct <- factor(c(1990, 1983, 1977, 1998, 1990))
levels(year_fct)

as.numeric(year_fct)
as.character(year_fct)

as.numeric(as.character(year_fct))
as.numeric(levels(year_fct))[year_fct]
```

Notice that in the `levels()` approach, three important steps occur:

- Obtain all the factor levels using `levels(year_fct)`
- Convert these levels to numeric values using `as.numeric(levels(year_fct))`
- Access these numeric values using the underlying integers of the vector `year_fct` inside the square brackets

```
# Inspect the contents of the sex column in surveys
summary(surveys$sex)

# Plot the sex column of the data in surveys
plot(surveys$sex)
```

Renaming Factors There are about 1700 individuals for which the sex information has not been recorded. To show them in the plot, we can turn the missing values into a factor level with the `addNA()` function.

```
sex <- surveys$sex
levels(sex)

# Add the NA data to be included in the data
sex <- addNA(sex)
levels(sex)

# Rename the NA level
levels(sex)[3]
levels(sex)[3] <- "undetermined"
levels(sex)

# Plot the modified data set
plot(sex)
```

Formatting Dates

https://datacarpentry.org/R-ecology-lesson/02-starting-with-data.html#Formatting_dates

Will not be covered today.

Data Carpentry: Manipulating, analyzing and exporting data with tidyverse

https://datacarpentry.org/R-ecology-lesson/03-dplyr.html

Data manipulation using dplyr and tidyr

dplyr is a package for helping with tabular data manipulation. It pairs nicely with tidyr which enables you to swiftly convert between different data formats for plotting and analysis.

The tidyverse package is an "umbrella-package" that installs tidyr, dplyr, and several other useful packages for data analysis, such as ggplot2, tibble, etc.

Read in our data using the read_csv() function from the tidyverse package `readr`

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

# Inspect the data
str(surveys)

# Preview the data
view(surveys)
```

dplyr functions

- `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

Selecting columns and filtering rows

- `select()` subset columns
  - The first argument is the data frame (surveys), and the subsequent arguments are the columns to keep.
- `filter()` subset rows on conditions

```
select(surveys,
       plot_id, species_id, weight)
```

Select all columns except certain ones by using a "-" in front of the variable to exclude it

```
select(surveys,
       -record_id, -species_id)
```

Choose rows based on specific criterion with `filter()`

```
filter(surveys, year == 1995)
```

Pipes

What if you want to select and filter at the same time? There are three ways to do this: use intermediate steps, nested functions, or pipes.

Pipes let you take the output of one function and send it directly to the next, which is useful when you need to do many things to the same dataset. Pipes in R look like `%>%` and are made available via the magrittr package, installed automatically with dplyr. 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.

```
filter(surveys,
       weight < 5)
select(surveys,
       species_id, sex, weight)

surveys %>%
  filter(weight < 5) %>%
  select(species_id, sex, weight)
```

Create a new object with this smaller version of the data by assigning it a new name

```
surveys_sml <- surveys %>%
  filter(weight < 5) %>%
  select(species_id, sex, weight)

surveys_sml
```

Mutate Create new columns based on the values in existing columns using `mutate()`

```
surveys %>%
  mutate(weight_kg = weight / 1000)
```

```
surveys %>%
  mutate(weight_kg = weight / 1000,
         weight_lb = weight_kg * 2.2)
```

The first few rows of the output contain NAs. To remove those we can insert a filter() in the chain of functions combined with `is.na()` to identify all NAs and the `!` operator to negate the result

```
surveys %>%
  filter(!is.na(weight)) %>%
  mutate(weight_kg = weight / 1000) %>%
  head()
```

Split-apply-combine data analysis and the `summarize()` function Many data analysis tasks can be approached using the split-apply-combine paradigm.

- Split the data into groups
- Apply some analysis to each group
- Combine the results

Key functions of dplyr for this workflow are `group_by()` and `summarize()`. `group_by()` is often used together with `summarize()`, which collapses each group into a single-row summary of that group.

```
surveys %>%
  group_by(sex) %>%
  summarize(mean_weight = mean(weight, na.rm = TRUE))
```

Once the data are grouped, you can also summarize multiple variables at the same time. Add a column indicating the minimum weight for each species for each sex

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight),
            min_weight = min(weight))
```

Sort the results by decreasing order of mean weight using the `desc()` function.

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight),
            min_weight = min(weight)) %>%
  arrange(desc(mean_weight))
```

Counting We often want to know the number of observations found for each factor or combination of factors. For this task, dplyr provides `count()`.

```
# Groups by a variable and summarizes it by counting the number of observations in that group
surveys %>%
    group_by(sex) %>%
    summarise(count = n())

# Same as above using the function count()
surveys %>%
    count(sex)

# Sorts results of count()
```

```
surveys %>%
    count(sex, sort = TRUE)
```

```
# Count combination of factors
surveys %>%
  count(sex, species)

# Sort the table according to a number of criteria
surveys %>%
  count(sex, species) %>%
  arrange(species, desc(n))
```

Reshaping with gather and spread Four rules defining a tidy dataset

- Each variable has its own column
- Each observation has its own row
- Each value must have its own cell
- Each type of observational unit forms a table

Here we examine the fourth rule: Each type of observational unit forms a table.

In surveys, the rows of surveys contain the values of variables associated with each record (the unit), values such as the weight or sex of each animal associated with each record. What if instead of comparing records, we wanted to compare the different mean weight of each genus between plots? (Ignoring plot_type for simplicity).

We'd need to create a new table where each row (the unit) is comprised of values of variables associated with each plot. In practical terms this means the values in genus would become the names of column variables and the cells would contain the values of the mean weight observed on each plot.

Having created a new table, it is therefore straightforward to explore the relationship between the weight of different genera within, and between, the plots. The key point here is that we are still following a tidy data structure, but we have reshaped the data according to the observations of interest: average genus weight per plot instead of recordings per date.

The opposite transformation would be to transform column names into values of a variable.

We can do both these of transformations with two tidyr functions, spread() and gather().

Spreading `spread()` takes three principal arguments:

- the data
- the key column variable whose values will become new column names
- the value column variable whose values will fill the new column variables

Let's use `spread()` to transform surveys to find the mean weight of each genus in each plot over the entire survey period. We use filter(), group_by() and summarise() to filter our observations and variables of interest, and create a new variable for the mean_weight.

```
surveys_gw <- surveys %>%
  filter(!is.na(weight)) %>%
  group_by(plot_id, genus) %>%
  summarize(mean_weight = mean(weight))

surveys_gw

str(surveys_gw)
```

This yields surveys_gw where the observations for each plot are spread across multiple rows, 196 observations of 3 variables. Using `spread()` to focus on genus with values from mean_weight this becomes 24 observations of 11 variables, one row for each plot.

```
surveys_spread <- surveys_gw %>%
  spread(key = genus, value = mean_weight)

surveys_spread

str(surveys_spread)
```

We could now plot comparisons between the weight of genera (one is called a genus, multiple are called genera) in different plots, although we may wish to fill in the missing values first.

```
surveys_gw %>%
  spread(key = genus, value = mean_weight, fill = 0) %>%
  head()
```

```
surveys_gw %>%
  spread(key = genus, value = mean_weight, fill = 0) %>%
  plot()
```

Gathering The opposing situation could occur if we had been provided with data in the form of surveys_spread, where the genus names are column names, but we wish to treat them as values of a genus variable instead.

In this situation we are gathering the column names and turning them into a pair of new variables. One variable represents the column names as values, and the other variable contains the values previously associated with the column names.

`gather()` takes four principal arguments:

- the data
- the key column variable we wish to create from column names.
- the values column variable we wish to create and fill with values associated with the key.
- the names of the columns we use to fill the key variable (or to drop).

To recreate surveys_gw from surveys_spread we create a key called **genus** and value called **mean_weight** and use all columns except **plot_id** for the key variable. Here we exclude plot_id from being gathered.

```
surveys_gather <-
  surveys_spread %>%
  gather(key = genus, value = mean_weight, -plot_id)

surveys_gather
str(surveys_gather)
```

Exporting Data

Before using write_csv(), we are going to create a new folder, `data`, in our working directory that will store this generated dataset. The data_raw folder should only contain the raw, unaltered data, and should be left alone to make sure we don't delete or modify it.

In preparation for our next lesson on plotting, we are going to prepare a cleaned up version of the data set that doesn't include any missing data.

Removing observations of animals for which weight and hindfoot_length are missing, or the sex has not been determined

```
surveys_complete <-
  surveys %>%
   filter(!is.na(weight),           # remove missing weight
          !is.na(hindfoot_length),  # remove missing hindfoot_length
          !is.na(sex))                # remove missing sex

surveys_complete

## Extract the most common species_id
species_counts <-
  surveys_complete %>%
   count(species_id) %>%
   filter(n >= 50)

species_counts

## Only keep the most common species
surveys_complete <-
  surveys_complete %>%
  filter(species_id %in% species_counts$species_id)

surveys_complete

# Check to see the length of the species count is the same as the file data frame surveys_complete
length(unique(surveys_complete$species_id))

# Check dimensions of final data frame
dim(surveys_complete)
```

Export data into new folder called “data”

```
# Uses relative path
dir.create("data")

write_csv(surveys_complete,
          file = "data/surveys_complete.csv")
```

Data Carpentry: Data visualization with ggplot2

https://datacarpentry.org/R-ecology-lesson/04-visualization-ggplot2.html

Prepare to plot

Load the packages and data for plotting

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

Plotting with `ggplot2`

ggplot2 is a plotting package that provides helpful commands to create complex plots from data in a data frame.

ggplot2 plots work best with data in the ‘long’ format, i.e., a column for every variable, and a row for every observation. Well-structured data will save you lots of time when making figures with ggplot2

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

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

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

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

```
ggplot(data = surveys_complete)
```

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

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

add 'geoms' – graphical representations of the data in the plot (points, lines, bars).

`geom_point()` for scatter plots, dot plots, etc. `geom_boxplot()` for, well, boxplots! `geom_line()` for trend lines, time series, etc.

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

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

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

Use the correct syntax when using +

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

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

Third, modify the plot to extract more information from it. For instance, we can add transparency (alpha) to avoid overplotting

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

Add colors for all the points

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

Color each species in the plot differently, using 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

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

Boxplot

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

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

Add points to the boxplot

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

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

Plotting time series data

Calculate the number of counts per year for each genus. First we need to group the data and count records within each group

```
yearly_counts <- surveys_complete %>%
  count(year, genus)
```

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

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

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

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

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

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

Integrating the pipe operator with ggplot2

We can also use the pipe operator to pass the data argument to the ggplot() function. The hard part is to remember that to build your ggplot, you need to use + and not %>%

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

The pipe operator can also be used to link data manipulation with consequent data visualization.

```
yearly_counts_graph <- surveys_complete %>%
    count(year, genus) %>%
    ggplot(mapping = aes(x = year, y = n, color = genus)) +
```

```
    geom_line()

yearly_counts_graph
```

ggplot has a special technique called faceting that allows the user to split one plot into multiple plots based on a factor included in the dataset. We will use it to make a time series plot for each genus

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

Add a different color scheme using the color package viridis

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

ggplot(data = yearly_counts, aes(x = year, y = n)) +
    geom_line(aes(color=genus)) +
    scale_color_viridis(discrete=TRUE) +
    facet_wrap(facets = vars(genus))
```

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

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

head(yearly_sex_counts)
```

We can now make the faceted plot by splitting further by sex using color (within a single plot)

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

We can also facet both by sex and genus

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

You can also organize the panels only by rows or only by columns

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

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

ggplot2 themes Every single component of a ggplot graph can be customized using the generic theme() function. There are pre-loaded themes available that change the overall appearance of the graph without

much effort. See more HERE.

A simple white background using the theme_bw() function

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

Change the names of axes to something more informative than 'year' and 'n' and add a title to the figure

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

Improve axis readability by increasing the font size. This can be done with the generic theme() function

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

Change the orientation of the labels and adjust them vertically and horizontally so they don't overlap

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

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

```
grey_theme <- theme(axis.text.x = element_text(colour="grey45", size = 12,
                                               angle = 90, hjust = 0.5,
                                               vjust = 0.5),
                    axis.text.y = element_text(colour = "grey45", size = 12),
                    text=element_text(size = 16))

ggplot(surveys_complete, aes(x = species_id, y = hindfoot_length)) +
    geom_boxplot() +
    grey_theme
```

Experimenting with layers

```
library(RColorBrewer)

pal <- colorRampPalette(c("pink","orange","skyblue"))(14)

ggplot(surveys_complete, aes(x = species_id, y = hindfoot_length)) +
    geom_boxplot(aes(color=species_id)) +
  scale_color_manual(values = pal) +
  labs(title = "Observed genera through time",
       x = "Year of observation",
       y = "Number of individuals") +
  theme_bw() +
  theme(
    axis.title.x = element_text(face = "bold"),
    axis.title.y = element_text(face = "bold"),
    legend.position = "none")
```

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 `patchwork` package allows us to combine separate ggplots into a single figure while keeping everything aligned properly.

- `+` to place plots next to each other
- `/` to arrange them vertically
- `plot_layout()` to determine how much space each plot uses
- More information HERE

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

plot_weight <-
  ggplot(data = surveys_complete, aes(x = species_id, y = weight)) +
  geom_boxplot() +
  labs(x = "Species", y = expression(log[10](Weight))) +
  scale_y_log10()

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

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

Exporting plots

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

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

```
    labs(title = "Observed genera through time",
        x = "Year of observation",
        y = "Number of individuals") +
    theme_bw() +
    theme(axis.text.x = element_text(colour = "grey20", size = 12, angle = 90,
                                     hjust = 0.5, vjust = 0.5),
          axis.text.y = element_text(colour = "grey20", size = 12),
          text = element_text(size = 16))

ggsave("path/name_of_file.png", my_plot, width = 15, height = 10)

## This also works for plots combined with patchwork
plot_combined <- plot_weight / plot_count + plot_layout(heights = c(3, 2))
ggsave("path/plot_combined.png", plot_combined, width = 10, dpi = 300)
```