Usefulness of R for managing data

Fábio A. Silva

2022-06-15

Contents

1 Introduction **5**

2 Basic R **7**

2.1 Note 7
2.2 What is R and RStudio 7
2.3 Why R? 8
2.4 Where to get it and how to run it 8
2.5 Environment 8
2.6 Scripts, Notebooks and RMarkdown 9
2.7 Working directories 10
2.8 Operators 10
2.9 Classes, Types and Structures 11
2.10 Comments 12
2.11 Objects 13
2.12 Combinations 13
2.13 Subsetting 13
2.14 Ifs and Whiles 14
2.15 Functions 15
2.16 Packages 16
2.17 Shortcuts 17
2.18 Chunk settings 17
2.19 Code completion 18
2.20 Other useful features 18

2.21 Getting help 18
2.22 Further reading 19

3 R - Dealing with data **21**
3.1 Opening files 21
3.2 Opening multiple files 23
3.3 Merging 24
3.4 Exporting 26
3.5 Special cases in R 27
3.6 Manipulating the data in dataframes 27
3.7 End 48

4 Exploratory Data Analysis (EDA) **49**
4.1 Loading data 50
4.2 Checking data 50
4.3 Cleaning 54
4.4 Summarising 60
4.5 Plots 62
4.6 End 80

Chapter 1

Introduction

This is a guide on R and its role in basic data manipulation. It requires no prior experience/knowledge in R.

Currently, it is divided into 3 chapters.

- **Chapter 1** will introduce you to R and provide you the basic programming tools, alongside other useful tips, to help you navigate R.
- **Chapter 2** will introduce you to data management tools.
- **Chapter 3** will introduce you to a more practical example of data exploration, emphasizing plot building.

In the future I plan to further elaborate on this guide, but for now I think its good enough for most people to get a picture on what R is, why you should use it, as well as, of course, how to use it.

I hope you learn something and find this guide useful.

Chapter 2

Basic R

2.1 Note

This portion of the guide is intended to present you the basics of R to get you up and running.
However, this is not meant to give you all the guidance and know-how, as well as answer all your questions. There are plenty of internet tutorials already that answer more specific and basic questions.
I'll provide a few links at the end that complement what I'll show you here.

2.2 What is R and RStudio

From their own website, simply put "R is a language and environment for statistical computing and graphics.".
So R is a programming language, just like Python, Matlab or even the syntax you find in SPSS. This language is particularly driven towards mathematical operations and thus proves quite useful when it comes to data management and analysis. Importantly, its an expansive (things keep getting added all the time) and open-source (the code is freely available).

As for RStudio, it is an interface for this programming language. Simply put, it makes working with R easier and everyone (in general) uses this interface to code in R. When you install R it already installs a very basic editor that lets you do some stuff, but RStudio is way more convening for everything other than simple calculations. You can see the basic R editor as Textpad (or Notepad application) and RStudio as Word.

2.3 Why R?

Here follows a list of things R is can do (and excels, pun intended, at):

- Read (nearly) all types of data.
- Manipulate data in every way possible.
- Create any type of graphic data visualization.
- Perform any type of statistical analysis.
- Perform any type of machine learning applications.
- Web scraping.
- Creating reports.
- Creating web apps.
- Many more…

2.4 Where to get it and how to run it

You can download R from https://cran.r-project.org/ and RStudio from https://www.rstudio.com/.

2.5 Environment

This is your basic environment. You have 4 (actually 5 with the upper bar area) divisions.

The top pane (1), gives you access to plenty of options regarding RStudio, letting you open files, save, search, etc..

The pane number 2 is your main area of work. This is your script. The script is the area where you will write your code. This code (all or whatever lines you select) will then be executed in the console.

The console (4) is where your code is executed. You can write single lines directly into it. This is where your information about your errors will show up.

Your right area of the screen contains 2 distinct areas, that can show different things. Area number 3 usually contains your environment. In here you will see every variable that you have attributed (they will be stored here). You can access them by clicking on them. It also contains other sub menus, but I will not go into detail here.

The bottom area (5) contains the files that are in the directory where you area working, will show your plots, will contain your packages (for easy access), the help menu and the viewer (interactive plots pane).

Note, all of your environment is customizable, changing both what panels appear and where, and the overall appearance of Rstudio (you change to dark mode if you think you're cool enough, for instance).

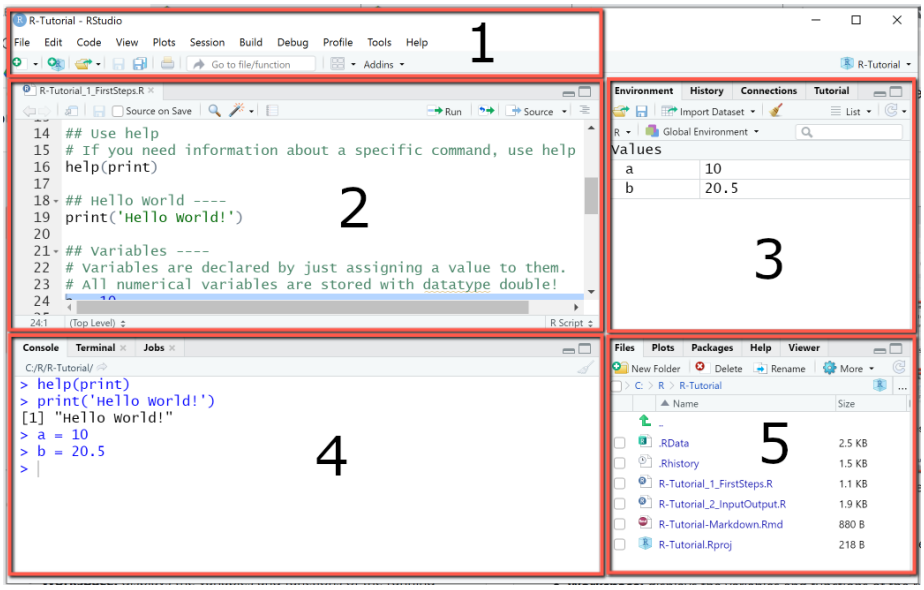


Figure 2.1: R IDE.

2.6 Scripts, Notebooks and RMarkdown

In creating a new R script you can choose one of three different R documents, **R Script**, **R Notebook** and **R Markdown**. Well you can actually choose more, but these are perhaps (particularly regarding the focus of this book) the most important ones.
R script is a simple sheet where you write code and add comments by using #. R scripts are best used when are writing scripts for automacy (automating tasks) and don't necessarily need to view any output along the way, but only at the end (when everything is computed and done). Otherwise, I would almost always recommend either one of the other two formats.

The other two formats are pretty much the same. They only differ in terms of their preview function. These types of special scripts are built with a markdown language, such that the script panel can now be seen as a text document and you can, within that "text document," insert code chunks and run these code

chunk as needed. Importantly, and perhaps one of the best features, is that the output of each chunk is, by default, shown just below that code chunk. This makes these scripts extremely useful when doing data analysis of any kind, given that, provided that they are not too messy, they have the potential to make great reports of your data analysis (showing results and graphs along the way). So if you are new o this, I would recommend using **R Notebook** otherwise just default to **R Markdown**.

2.7 Working directories

Whenever you create a new R-type file (e.g., R-script, Rmarkdown, etc.) and you save it somewhere (e.g., "C:/Users/fabio/OneDrive/My Things/Stats/Teaching/R_Book") it sets your working directory to that folder. You can change verify this by using the command `getwd()`.

You can also alter this working directory by executing `setwd('what working directory I want")`.

Importantly, when saving your working directory it should in a string format and separated by "/" and not "\". For instance `setwd("C:/Users/MyName/Work/Project1")`.

2.8 Operators

R has **arithmetic operators** and **logical operators**.
The first ones, **arithmetic operators** are the following (there are more but, I believe, less important):

`+`: adds
`-`: subtracts
`*`: multiplies
`/`: divides
`^` or `**`: exponentiates

Then there are **logical operators**:

`<`: lesser than
`<=`: lesser or equal than
`>`: bigger than
`>=`: bigger or equal than
`==`: equals
`!=`: not equal to
`!x`: not x
`x | y`: x OR y
`x & y`: x AND y

2.9 Classes, Types and Structures

The data you imported can be in a wide range of classes (types). There are 3 basic types of classes, built-in (different functions can use more), you need to be aware (although there are more). These are:

- *character*: Strings (words), such as `"hello"` or `"hi123"`.
- *numeric*: Any type of number, such as `2`or `30.4`.
- *logical*: The true or false values. `TRUE` or `FALSE`.
- *date*: In a date format. Can be converted from different types (e.g., `2007-11-11`, `2jan1960`)

You can ask R about what type of object it is by using the `class(object)` command or the `typeof(object)`

R also has different data structures. The ones worth talking about here are:

- *atomic vector*: Basically every R data structure. A vector could be a character or an numeric object, for instance.
- *lists*: a list of objects. Can be created by using `list()`. You can retrieve the value by using `[[]]`.

```
my_list <- list('Potatoes', TRUE, 15, c('Strawberry', 1000))
# or
my_list <- list(ingredients = c('Strawberry', 'Milk', 'Coffee'), type = 'Milkshake', rating = 10)
```

- *matrix*: are like tables.

```
my_matrix <- matrix(1:9, nrow = 3, ncol = 3)
my_matrix <- matrix(1:9, nrow = 3, dimnames = list(c("X","Y","Z"), c("A","B","C")))
```

- *data frame*: fancy matrices (more common)

```
d <- iris
```

- *factor*: more of a type of class than anything. This object is a factor with levels.

```
class(d$Species)
```

```
## [1] "factor"
```

```
levels(d$Species)
```

```
## [1] "setosa"     "versicolor" "virginica"
```

- *tibble*: a special data frame from tidyverse

For additional info, visit: https://swcarpentry.github.io/r-novice-inflammation/13-supp-data-structures/

2.10 Comments

Commenting your code is good practice. You can comment anything, but perhaps its most practical use is to leave instructions as to what the code, or section of code, is doing.

Commenting is also great for when you don't want a certain part of your code to run.

Below you see a function, kinda of a confusing one, where comments in each line help me (and other readers) understand what the function is doing.

```
# My list
my_list <- c('Water0','Fire10','Ear899th', '1Wind0')

# Creating a function to remove numbers from my characters
rem_num <- function(list_of_words){
  final_word_list <- c()  # creates empty list
  for (word in list_of_words){  # creating loop
    numberless <- gsub('[[:digit:]]+', '', word)  # removing numbers
    final_word_list <- c(final_word_list, numberless)  # adding to a final list
  }
  print(final_word_list) # show list at the end
}

# Executing function
rem_num(list_of_words = my_list)
```

```
## [1] "Water" "Fire"  "Earth" "Wind"
```

2.11 Objects

In R you can create objects. This can be any type of classes we discussed above. Your object can be a letter, a number, a word, a mix of letters and numbers, a dataframe, a vectors, a list, multiple lists, multiple data frames, you name it. If your object contains multiple things, it becomes a list. Objects, when created, appear in your Environment panel (top right). To create an object you must give it a name and then you use <- followed by whatever you want to make an object.
For instance:

```
a <- 1
b <- 'Hi'
y <- c(1, 'Hi', 3.5)
y <- c(5, 6)
d <- data.frame('Name' = c('Ana','João','Pedro'), 'Age' = c(25, 40, 19))
```

2.12 Combinations

In R, as you've seen above, you can use combinations and attribute these combinations to a variable.

```
# Create a combination with multiple items.
c <- c(1, 'Hi', 3.5)

# Asking the mean of that set of items.
mean(x = c(3, 4))
```

```
## [1] 3.5
```

2.13 Subsetting

When you have complex objects (not just a single entry object) you can use subset to select only a specific part of said object.

```
a <- c('apples','bananas','carrots','oranges','strawberry')

a[1]   # selects first entry
```

```
## [1] "apples"
```

```
a[c(2,3)]  # selects entries 2 and 3
```

```
## [1] "bananas" "carrots"
```

```
a[3:5]  # selects entry 3 to entry 5
```

```
## [1] "carrots"    "oranges"    "strawberry"
```

2.14 Ifs and Whiles

`if()`, as well as `while()`, functions are really basic functions in programming languages, that allow you to iterate certain actions (i.e., perform an action multiple times). They are written, and mean, the following:

If function

if (condition){

perform_this # if true

} else {

perform_that # if false

}

While function

while (condition){

perform_this

}

Here are just a few basic examples for you to get an idea.

```
# First example
if (3 > 2){
  print('True')
} else {
  print('False')
}
```

```
## [1] "True"
```

```
# Second example
my_list = c('orange', 'banana', 'apple')

if (my_list[2] == 'apple'){
  print('The second item in the list is apple!')
} else if (my_list[2] == 'orange'){
  print('The second item in the list is orange!')
} else {
  print('The second item in the list must be banana!')
}
```

```
## [1] "The second item in the list must be banana!"
```

```
# Third example
my_number = 0

while(my_number < 5){
  print('My current number is lower than 5')
  my_number <- my_number + 1
}
```

```
## [1] "My current number is lower than 5"
## [1] "My current number is lower than 5"
## [1] "My current number is lower than 5"
## [1] "My current number is lower than 5"
## [1] "My current number is lower than 5"
```

2.15 Functions

A function is a set of statements (commands) organized in a way to perform a task. R, for instance, already has a large number of in-built functions (for instance, `mean(c(5, 10, 15))` is a function). The cool thing, however, is that, as with other programming languages, we can build our own function. The function takes a set of arguments and performs actions over that argument and output something (if we want).
Its general structure is as follows:

```
function_name <- function(arg_1, arg_2, ...){
   actions and commands
}
```

Here are a few examples. Lets say we want to create a function that tells us the square root of our input.

```
square_function <- function(number_to_be_squared){
  final_number <- (number_to_be_squared)^2
  return(final_number)
}

# Now lets call the function
square_function(number_to_be_squared = 9)
```

```
## [1] 81
```

Functions can also be called without arguments.

```
arg_less <- function(){
  print('Just print this')
}

arg_less()
```

```
## [1] "Just print this"
```

They can also have multiple inputs

```
mult_fun <- function(a, b, c){
  output <- a + b^2 + sqrt(c)
  print(output)
}

mult_fun(a = 1, b = 2, c = 5)
```

```
## [1] 7.236068
```

```
# Or simply
# mult_fun(1, 2, 5)
```

2.16 Packages

There are over 9000 packages. There are packages for nearly everything you want to do. These packages add new functions to R.

You have *base packages* (already installed, but not loaded) and *contributed packages* (need to be installed and loaded).

You can install the latter by using the following command `install.packages('package_name')` and call (load) packages with `library(package_name)`.
Most packages come from CRAN (Comprehensive R Archive Network, which is basically R's house), thus all you need to write is that.

However, if some packages come from github for instance you need to install the packages "devtools" (`install.packages('devtools')`) then load it (`library(devtools)`) and finally use `install_github("profile_name/repository")` to install the package. Usually if you google these packages they will have instructions on how to install them.

You can also load and detach (remove package from your toolset for the time being) packages that you have already installed by going to the packages panel (bottom right) and searching and then selecting the package you want to load.

In specific situations, particularly if you just want to use one function of a package, instead of loading the package you can simply write `package_name::function_name()`. R will just use that function from that package for that line in specific, but not load the package.

2.17 Shortcuts

Here follows a list of useful shortcuts. Note that these are the ones I can't live without. There are plenty more, you just have to google them.

- **Insert the pipe operator (%>%)**: Command + Shift + M on a Mac, or Ctrl + Shift + M on Linux and Windows.
- **Run the current line of code**: Command + Enter on a Mac or Control + Enter on Linux and Windows.
- **Run all lines of code**: Command + A + Enter on a Mac or Control + A + Enter on Linux and Windows.
- **Comment or un-comment lines**: Command + Shift + C on a Mac or Control + Shift + C on Linux and Windows.
- **Insert a new code chunk**: Command + Option + I on a Mac or Control + Alt + I on a Linx and Windows.

2.18 Chunk settings

When using R Notebooks or R Markdown, you'll work with code chunks, as explained already. One important thing to know is that these chunks are ad-

justable, and these adjustments are both easy to make and can come in handy when compiling your code. These adjustments are made on the top part of the chunk that is between `{}`. So for instance, in a normal chunk you have the top portion as "{r}". The first parameter tells you which language you are coding in, which in this case is R. If you are wondering, yes you can change the setting to code in other languages depending if you have them installed and configured with RStudio (e.g., "`{python}`"). Besides the language, you can change plenty of parameters. They have to be separated only by commas. Below are some brief examples of the ones I most commonly use (all of which, except "fig.width" and "fig.height" are defaulted to TRUE if you don't set them):

1. `include = FALSE` : Runs the code, but does not include the code and the results in the finished file.
2. `echo = FALSE` : Runs the code, but does not include the code in the finished file (but it includes the results).
3. `message = FALSE` : Prevents messages generated by the code from appearing.
4. `warning = FALSE` : Prevents warnings generated by the code from appearing.
5. `fig.width = "width_in_pix"` and `fig.height = "height_in_pix"` : Set the width and height of the picture/plot.

For more details go to: https://rmarkdown.rstudio.com/lesson-3.html

2.19 Code completion

When you begin to type, R will pop up suggestions as to what you want to write. These can be names of functions, variables or data frames for example. You can press TAB to select and let R auto-fill or just click on it with your mouse.

2.20 Other useful features

Pressing CTRL + F will open up the find/replace menu at the top of your script screen. This can be immensely useful for you to find, and specially, replace lots of things within a portion (the portion you have selected) or the whole code (if you have no code selected).

2.21 Getting help

To get help you have several options, which depend a lot on which type of help you need.

If you want to know what arguments a functions requires, you can write its name and, and having your writing bar inside the "()" press TAB.

If you need generic help about a function - what arguments go where, or, for instance, what does "na.rm" mean in that function - you can use the command `?function_name()` in which "function_name" is the name of your function. This will pop up a help menu regarding that function that will appear on your help panel (bottom right). You can also click "F1" while your cursor is over the function name. Depending on the function and its respective documentation this help can either be good or poor.

If you need further help, or need help in problems that are not just concerned about a single function, you will want an internet connection. Having google searching skills will help lots with your R programming.
A few notes and advices:
- Search your problem using English language.
- Search your problem in a generic way using proper searching language (don't write sentences, write the most relevant words of the problem).
- If its an error just post the error message on google.

Don't be afraid or let down if you need to use google to help you with stuff you learned but just don't remember. Its normal, trust me.

2.22 Further reading

Need a more in-depth fundamentals of R? See the links below:
https://www.evamariakiss.de/tutorial/r/

https://www.youtube.com/watch?v=_V8eKsto3Ug&ab_channel=freeCodeCamp.org

Chapter 3

R - Dealing with data

3.1 Opening files

Before opening the data
First, you need to tell R where the files are located. He is lazy that way, and will not look everywhere on your computer for them. So, tell him by using the command `setwd()`.

Reading the data
Usually, when using R, you want to work with data. This data is usually already there and you want to open it in R. The good thing is that R can read just about anything (just google "read"file type" in R" on google). Here I show you how to read some of the most common formats. Be sure to install the `xlsx` and `haven` packages to open Excell and SPSS files, respectively. Additionally, there are multiple ways to read the same file. Some will be built in R itself, others will require external packages. This is important to know because some functions, although working, may be outdated or just give you some sort of weird error. Maybe you can overcome this by using a different package.

If you want to import/read an Excel file, just use:
`read.xlsx(file = 'example.xlsx', sheetName = 'page_1', header = TRUE)` (xlsx package)
If a text
`read.delim(file = 'example.txt', header = TRUE, sep = ',', dec = '.')`
CSV:
`read.csv(file = 'example.csv', header = TRUE, sep = ',', dec = '.')`
SAV (SPSS):
`read_sav(file = 'example.sav')` (haven package)

Managing your imported data
To have your data in your environment, so that you can mess with it, you should assign your `read` command to a variable (object). Lets say you do the following `mydata <- read.delim(file = 'example.txt', header = TRUE, sep = ',', dec = '.')`. Now, your `mydata` object is the dataframe containing that imported data set.

```
mydata <- read.csv('data/heart/heart_2020_cleaned.csv')
```

Possible problems
You may encounter several problems. Here are a few of the most common error messages you will face when importing data to R.

- "the number of columns is superior to the data" or the data is all jumbled.

Perhaps one of the most common problems. This probably has to due with R separating columns where it shouldn't and making more columns than it should. You can fix this perhaps by making sure the `sep` command is specifying the exact separator of the file. It helps to open the file with excel, for instance, and check the separator and the decimals symbol (you don't want to be separating columns by the decimal symbol). For instance, sometimes R reads the .csv file (which means comma separated file) and you have commas as decimals (instead ";" is the separator). This creates way to many columns that mismatch the number of headers present.

- cannot open file 'name_of_file.csv': No such file or directory.

Make sure you are in the right working directory or are specifying the path correctly.

There will surely be more problems, but you can find a way around them by using google.

Checking the data
After you've opened the data, you should take a peak at it. There's several ways of doing this such as `head(df)` or some others I'm not recalling at the moment. Lets see bellow.

```
head(mydata)

# you can add a "," and say the number of rows you want to preview
head(mydata, 10)

# Or you can just View it all
#View(mydata)
```

3.2 Opening multiple files

Lets say you have a folder, lets assume is named "data", and in there you have all your data files (files from each participant). Ideally, to analyze and view the data as a whole, you would want to import all of the data files, and then merge them into a big data file, containing all the participants (identified accordingly). Here's a snippet of code that would allow you to do that. Beware though, any file that matches the criteria (in this case ".csv" files) will be gathered from the folder (your working directory).

Firstly, lets gather the names of the files in our directory that we want to import.

```
# Setting our working directory
setwd('C:/Users/fabio/OneDrive/My Things/Stats/Teaching/R_Book/data')

# Empty list to list all txt files in folder
list_of_files <- list()

# Searches for ".csv" files in the folder
list_all_data <- dir(pattern='.csv')

# For each ".csv" file, append this file to a list that will contain all ".csv" file names.
for (file in list_all_data){
  list_of_files <- c(list_of_files, file)
  cat('\nFile processed:', file)  # prints the file names
}
```

```
##
## File processed: Participant_1.csv
## File processed: Participant_10.csv
## File processed: Participant_2.csv
## File processed: Participant_3.csv
## File processed: Participant_4.csv
## File processed: Participant_5.csv
## File processed: Participant_6.csv
## File processed: Participant_7.csv
## File processed: Participant_8.csv
## File processed: Participant_9.csv
```

```
# Organizing the files
list_of_files <- as.character(list_of_files)  # To character
#list_of_files <- mixedsort(sort(list_of_files))  # Sorting them by name
list_of_files <- as.list(list_of_files)  # Transforming it to list

# How many files were processed?
cat('\nTotal number of files processed:', length(unlist(list_of_files)))
```

```
##
## Total number of files processed: 10
```

Then we need to create a function that iterates over all of this file names. It will need to import each file (by their name) and append them to a general dataframe.

```
# Setting our working directory
setwd('C:/Users/fabio/OneDrive/My Things/Stats/Teaching/R_Book/data')

# Create an empty dataframe where all data will be stored
df_all <- data.frame()

# Create a simple reading function for every element in a list
import_data <- function(file_name){
  for (name in file_name){  # for every name in that list
    df <- read.csv(name, header=TRUE, encoding = 'utf-8', sep=',')  # read that name a
    df_all <- rbind(df_all, df)  # add "df" to that list
  }
  df_all <<- df_all  # Makes this object available globally
}

# Running function
import_data(file_name = list_of_files)
```

3.3 Merging

You can join two data frames either by rows or by columns. Typically, you use rows when you try too join more data to your current data frame. To do so, you can use `rbind()`.

```
# Splitting the data by rows
d1 <- USArrests[1:20, ]
d2 <- USArrests[21:nrow(USArrests), ]

# Creating a new dataframe with the merged data
merged_d <- rbind(d1, d2)
```

More frequently, perhaps, you want to join complementary information (more variables) to your preexisting data. To do so, you can use `cbind()`.

```
# Splitting the data by columns
d1 <- USArrests[, c(1,2)]
```

```
d2 <- USArrests[, c(3,4)]

# Creating a new dataframe with the merged data
merged_d <- cbind(d1, d2)
```

However, this above code only works correctly (as intended) if your data is perfectly lined up. For instance, `rbind()` will work if you have the same number of variables (columns), with the same names and in the same positions. So you need to be sure this is the case before you merge the data frames.

As for `cbind()` on the other hand, it requires you to have the same number of entire (rows) and for these to be arranged in the same manner (otherwise your info would mismatch). You can try and order things correctly, but you can easily place some information incorrectly. To circumvent this, you can use `merge()`. In this command you only have to specify the IDs (e.g., "sample_ID" or "person_ID") that allow R to connect the information in the right place.

```
# Preparing the data
d <- USArrests
d$State <- rownames(d)
rownames(d) <- NULL
d <- d[, c(5,3,1,2,4)]

# Creating two separate dataframes
d1 <- d[, c(1:2)]
d2 <- d[, c(1, 3:5)]
```

```
# Joining dataframes by the "State" column
d_all <- merge(x = d1, y = d2, by = 'State')
```

Now lets say the data frames weren't perfectly matched. For instance lets say we remove Alabama from `d1`.

```
d1 <- d1[-1, ]  # Removing Alabama

# Merging
d_all <- merge(x = d1, y = d2, by = 'State')  # adds only what matches
d_all <- merge(x = d1, y = d2, by = 'State', all = TRUE)  # adds everything

head(d_all)
```

```
##        State UrbanPop Murder Assault Rape
## 1    Alabama       NA   13.2     236 21.2
## 2     Alaska       48   10.0     263 44.5
```

```
## 3    Arizona       80    8.1     294 31.0
## 4   Arkansas       50    8.8     190 19.5
## 5 California       91    9.0     276 40.6
## 6   Colorado       78    7.9     204 38.7
```

Now you check the `d_all` you will see that there is no Alabama. You can use the parameter `all` or `all.x` or `all.y` to indicate if you want all of the rows in the data frames (either all or just the x or y data frames, respectively) to be added to the final data frame. If so, as you can see, Alabama is also imported, even thought there is `NA` in the in one of the fields (because even though its not in d1, it is in the d2 data frame). There are other parameters that can be tweaked for more specific scenarios, just run `?merge()` to explore the function and its parameters.

3.4 Exporting

Aside from importing, sometimes we also want to export the files we created/modified in R. We can do this with several commands, but perhaps the simpler ones are:

```
write.table(x = df, file = 'namewewant.txt', sep = ',', dec =
'.')
```

This tells R to export the `df` dataframe, to a not existing file with a name "namewewant.txt", that is separated by commas and has "." for decimal points. We can also export to an existing data file, and ask for `append = TRUE`, thus appending our data to the data already existing in that file. Be sure thought, that this data has the same structure (e.g., number of columns, position of the columns).

We can also do the same thing as above, but instead create a ".csv" file.

```
write.csv(x = df, file = 'namewewant.csv')
```

As an example, lets export the dataframe we created in the chunks above. Note that if we don't specify the path along with the name of the file to be created, R will save the file to the current working directory.

```
# Tells the path I want to export to.
path = 'C:/Users/fabio/OneDrive/My Things/Stats/Teaching/R_Book/'

# Merges the path with the file name I want to give it.
filename <- paste(path, 'some_data.csv', sep = '')

# Export it
write.csv(x = df_all, file = filename)
```

3.5 Special cases in R

In R variables, but more specifically on data frames, you can encounter the following default symbols:

- **NA**: Not Available (i.e., missing value)
- **NaN**: Not a Number (e.g., 0/0)
- **Inf** e **-Inf**: Infinity

These are special categories of values and can mess up your transformations and functions. We will talk about them more in the next chapter.

3.6 Manipulating the data in dataframes

Now, in R you can manage your dataframe as you please. You can do anything. And I truly mean anything. Anything you can do in Excel and then some.

3.6.1 Subsetting a dataframe

Subsetting is a very important skill that you should try to master. It allows you to select only the portions of your data frame that you want. This is vital for any type of data manipulation and cleaning you try to accomplish.

The symbols `$` lets you subset (select) columns of a dataframe really easily, if you just want a column.

```
df <- iris
df$Sepal.Length
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
##  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
##  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
##  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
##  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
##  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

If you want more columns, you can use `[]`. By indicating `df[rows,columns]`.

```
df[ , 'Sepal.Length']  # just the "Sepal.Length]"
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
##  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
##  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
##  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
##  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
##  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

```
df[5, ]   # row 5 across all columns
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 5            5         3.6          1.4         0.2  setosa
```

```
df[1, 'Sepal.Length']  # row 1 of the "Sepal.Length" column
```

```
## [1] 5.1
```

```
df[c(1:5), c('Sepal.Length', 'Sepal.Width')]  # row 1 to 5 from the Sepal.Length" and
```

```
##   Sepal.Length Sepal.Width
## 1          5.1         3.5
## 2          4.9         3.0
## 3          4.7         3.2
## 4          4.6         3.1
## 5          5.0         3.6
```

3.6.2 Columns

Lets start by some simply manipulations. Lets say you want to change column names. Ideally, I would avoid spaces in the headers (and overall actually) but you do as you please.

```
df <- iris  # iris (mtcars) is a built-in dataset. Just imagine I'm reading from a fil
# Option 1
colnames(df) <- c('Colname 1', 'Colname 2', 'Colname 3', 'Colname 4', 'Colname 5')

# Option 2
```

```
names(df) <- c('Colname 1', 'Colname 2', 'Colname 3', 'Colname 4', 'Colname 5')

# Or just change a specific column name
colnames(df)[2] <- 'Colname 2 - New'

# Final result
head(df)
```

```
##   Colname 1 Colname 2 - New Colname 3 Colname 4 Colname 5
## 1       5.1             3.5       1.4       0.2    setosa
## 2       4.9             3.0       1.4       0.2    setosa
## 3       4.7             3.2       1.3       0.2    setosa
## 4       4.6             3.1       1.5       0.2    setosa
## 5       5.0             3.6       1.4       0.2    setosa
## 6       5.4             3.9       1.7       0.4    setosa
```

We can also change the order of the columns.

```
df <- iris # Just restoring the dataframe to be less confusing
df <- df[,c(5,1,2,3,4)]  # 5 column shows up first now, followed by the previous first column, et
```

We can sort by a specific (or multiple columns).

```
df <- df[order(df[, 2]), ]  # Orders by second column
df <- df[order(-df[, 2]), ]  # Orders by second column descending

df <- df[order(-df[, 2], df[, 3]), ]  # Orders by second columns descending and then by third col

# Alternatively since this is a bit confusing (does the same as above, respectively)
df <- dplyr::arrange(df, Sepal.Length)
df <- dplyr::arrange(df, desc(Sepal.Length))

df <- dplyr::arrange(df, desc(Sepal.Length), Sepal.Width)
```

We can create new columns.

```
new_data <- rep('New info', nrow(df))  # Creating new irrelevant data

df$NewColumn <- new_data  # Added this data (data must have same length as dataframe!)

head(df)
```

```
##     Species Sepal.Length Sepal.Width Petal.Length Petal.Width NewColumn
```

```
## 1 virginica          7.9         3.8          6.4         2.0  New info
## 2 virginica          7.7         2.6          6.9         2.3  New info
## 3 virginica          7.7         2.8          6.7         2.0  New info
## 4 virginica          7.7         3.0          6.1         2.3  New info
## 5 virginica          7.7         3.8          6.7         2.2  New info
## 6 virginica          7.6         3.0          6.6         2.1  New info
```

We can remove columns.

```
df$Petal.Length <- NULL
# or
df <- within(df, rm(Sepal.Length))
```

And we can create and transform the columns.

```
df <- iris

df$Sepal_Area <- df$Sepal.Length * df$Sepal.Width  # Creating new variable with is the

df$Sepal_Area <- round(df$Sepal_Area, 1)  # Transforming existing variable, making it

head(df)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Sepal_Area
## 1          5.1         3.5          1.4         0.2  setosa       17.8
## 2          4.9         3.0          1.4         0.2  setosa       14.7
## 3          4.7         3.2          1.3         0.2  setosa       15.0
## 4          4.6         3.1          1.5         0.2  setosa       14.3
## 5          5.0         3.6          1.4         0.2  setosa       18.0
## 6          5.4         3.9          1.7         0.4  setosa       21.1
```

3.6.3 Rows

Altering specific rows is a bit trickier. Fortunatelly, this is usually less relevant, since we usually just want to change or apply a condition to an entire column. Having said this, here's some relevant commands.

Say you want to alter rows that meet a condition.

```
df$Sepal.Length[df$Sepal.Length <= 5] <- '<5'  # Any value in in the Sepal.Length colu

df$Sepal.Length[df$Sepal.Length == 7.9] <- 8  # Changing rows with 7.9 to 8.
```

Or want to create a new entry (i.e., row).

```
row <- data.frame(5.6, 3.2, 1.9, 0.1, 'new_species', 10000)  # Create a new row (all columns mus
colnames(row) <- colnames(df)
df <- rbind(df, row)

tail(df)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width     Species Sepal_Area
## 146          6.7         3.0          5.2         2.3   virginica       20.1
## 147          6.3         2.5          5.0         1.9   virginica       15.8
## 148          6.5         3.0          5.2         2.0   virginica       19.5
## 149          6.2         3.4          5.4         2.3   virginica       21.1
## 150          5.9         3.0          5.1         1.8   virginica       17.7
## 151          5.6         3.2          1.9         0.1 new_species    10000.0
```

Or just want to delete a row.

```
df <- df[-c(151, 152),]  # deletes row 152 and 152
```

If you want a package that allows you to do the above changes in rows and columns just like you would in Excel, you can too. Just visit: https://cran.r-project.org/web/packages/DataEditR/vignettes/DataEditR.html

Although I would argue against it, since this doesn't make your R code easy to re-execute.

3.6.4 Tidyverse & Pipes

Before presenting the following commands below, we should talk quickly about tidyverse and pipes. Tidyverse, as the name implies "Tidy" + "[Uni]verse" is a big package that contains more packages. All of these packages are designed for data science. These are:

- **dplyr**: Basic grammar for data manipulation (also responsible for pipes).
- **ggplot2**: Used to create all sorts of graphics.
- **forcats**: Facilitates functional programming for data science (e.g., can replace loops with maps, a simpler command)
- **tibble**: Better dataframe, making it cleaner and more efficient (although they are mostly interchangeable).
- **readr**: Reads data of several types in a smart manner (including csv).
- **stringr**: Makes working with string information easy.

- **tidyr**: Helps to tidy data presentation.
- **purr**: Makes handling factors (categorical variables) easier.

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.6     v purrr   0.3.4
## v tibble  3.1.5     v dplyr   1.0.8
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1

## Warning: package 'ggplot2' was built under R version 4.1.3

## Warning: package 'tidyr' was built under R version 4.1.3

## Warning: package 'readr' was built under R version 4.1.3

## Warning: package 'dplyr' was built under R version 4.1.3

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

As you can see by the output you get when you load it, it basically loads them all making in a single line.

Now onto pipes. Basically this allow you to chain your commands. It comes from the `dplyr` or `magritrr` packages. It can be read as follows:

`WITH THIS %>% EXECUTE THIS %>% THEN EXECUTE THIS %>% THEN THIS`

So instead of this:

```
object <- function_1(object)
object <- function_2(object)
object <- function_3(object)

# or
object <- function_3(function_2(function_1(object)))
```

We can instead have this

```
object %>%
  function_1() %>%
  function_2() %>%
  function_3()
```

Here are two concrete examples:

1. With 4+4, add another 4. `4+4 %>% +4`
2. With my dataframe (df), select its "column1" and then calculate the mean. `df %>% select(column1) %>% mean()`

Remember, you can call this pipe command by pressing "CTRL & SHIFT + M" in Windows and Command + Shift + M on a Mac.

You may find it weird at first, but trust me, it will become intuitive in no time.

If you want a better tutorial on pipes just visit the following link: https://www.datacamp.com/community/tutorials/pipe-r-tutorial

3.6.5 Filtering

Now, lets say we want to **filter** the dataframe. That is, we want to select our data based on some criteria.

```
df <- iris
#We can filter by Species. In this case we are only selecting "setosa".
df %>%
  filter(Species == 'setosa')

# Or we can select "everything but".
df %>%
  filter(Species != 'setosa')

# And we can even select multiple things
df %>%
  filter(Species != 'setosa' & Sepal.Length > 7.5)

# We can also select one "OR" the other
df %>%
  filter(Species != 'setosa' | Sepal.Length > 7.5)

# We can remove NAs
df %>%
  filter(!is.na(Species))
```

3.6.6 Arranging

We can **arrange** the dataframe as we wish. We can sort by just 1 column or more. In the latter case the second, third and so on variables will break the ties. Missing values are sorted to the end.

```
# It defaults as ascending
df %>%
  arrange(Sepal.Length)

# We can make it descending:
df %>%
  arrange(desc(Sepal.Length))
```

3.6.7 Selecting

Another useful trick is to **select** columns. With this command we can select the columns we want, or do not want.

```
# Selecting Sepal.Lenght and Species columns
df %>%
  select(Sepal.Length, Species)

# We can also select multiple columns by saying from x to y:
df %>%
  select(Sepal.Length:Species)

# To select everything but:
df %>%
  select(-c(Sepal.Length, Species))
```

3.6.8 Mutating

To create new columns (or modify existing ones), we can use **mutate**. This a versatile command that allows you to do several things. Here are a bunch of examples:

```
# Create a new column with just the string "word" on it.
df <- df %>%
  mutate(WordColumn = 'word')

# Create a combination of two columns
df %>%
```

```
  mutate(TwoColsTogether = paste(Species, WordColumn))

# Create the sum of two columns
df %>%
  mutate(SumOfCols = Petal.Length + Petal.Width)

# Among others
df %>%
  mutate(Times100 = Petal.Length*100)

df %>%
  mutate(DividedBy2 = Petal.Width/2)
```

3.6.9 Ifelse

`ifelse()` is a base function of R (not from tidyverse, altough you have `if_else()` from tidyverse which works and does exactly the same thing), but it fits quite well with its workflow. Specifically it fits quite well with the `mutate()` command. What it basically says is: `if THIS_CONDITION_IS_MET then DO_CASE_1 otherwise DO_CASE_2`. The function will look like this: `ifelse(THIS_CONDITION_IS_MET, DO_CASE_1, DO_CASE_2)`. Lets look at some examples below.

```
df <- iris

# Replacing for just 1 condition.
df %>%
  mutate(SpeciesAlt = ifelse(Species == 'setosa', 'Specie1', Species)) %>%
  head()  # just to show the first 5 rows for the purpose of demonstration.
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species SpeciesAlt
## 1          5.1         3.5          1.4         0.2  setosa    Specie1
## 2          4.9         3.0          1.4         0.2  setosa    Specie1
## 3          4.7         3.2          1.3         0.2  setosa    Specie1
## 4          4.6         3.1          1.5         0.2  setosa    Specie1
## 5          5.0         3.6          1.4         0.2  setosa    Specie1
## 6          5.4         3.9          1.7         0.4  setosa    Specie1
```

```
# Replacing for 3 conditions (Gets a bit chaotic)
df %>%
  mutate(SpeciesAlt = ifelse(Species == 'setosa', 'Specie1',
                             ifelse(Species == 'versicolor', 'Specie2',
                                    ifelse(Species == 'virginica', 'Specie3', Species)))) %>%
  head()  # just to show the first 5 rows for the purpose of demonstration.
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species SpeciesAlt
## 1          5.1         3.5          1.4         0.2  setosa    Specie1
## 2          4.9         3.0          1.4         0.2  setosa    Specie1
## 3          4.7         3.2          1.3         0.2  setosa    Specie1
## 4          4.6         3.1          1.5         0.2  setosa    Specie1
## 5          5.0         3.6          1.4         0.2  setosa    Specie1
## 6          5.4         3.9          1.7         0.4  setosa    Specie1
```

As you can see, for changing just 1 Species, its quite easy and practical. But for more than say 2 it starts to get very confusing.

As a simpler alternative, when you deal with plenty of cases, you should use `recode()`.

```
# Recoding all 3 cases
df %>%
  mutate(SpeciesAlt = recode(Species, setosa = "Specie1",
                             versicolor = "Specie2",
                             virginica = 'Specie3')) %>%
  head()  # just to show the first 5 rows for the purpose of demonstration.
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species SpeciesAlt
## 1          5.1         3.5          1.4         0.2  setosa    Specie1
## 2          4.9         3.0          1.4         0.2  setosa    Specie1
## 3          4.7         3.2          1.3         0.2  setosa    Specie1
## 4          4.6         3.1          1.5         0.2  setosa    Specie1
## 5          5.0         3.6          1.4         0.2  setosa    Specie1
## 6          5.4         3.9          1.7         0.4  setosa    Specie1
```

```
# Recoding just 2 and giving all the rest the label "others"
df %>%
  mutate(SpeciesAlt = recode(Species, setosa = "Specie1",
                             versicolor = "Specie2", .default = 'others')) %>%
  head()  # just to show the first 5 rows for the purpose of demonstration.
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species SpeciesAlt
## 1          5.1         3.5          1.4         0.2  setosa    Specie1
## 2          4.9         3.0          1.4         0.2  setosa    Specie1
## 3          4.7         3.2          1.3         0.2  setosa    Specie1
## 4          4.6         3.1          1.5         0.2  setosa    Specie1
## 5          5.0         3.6          1.4         0.2  setosa    Specie1
## 6          5.4         3.9          1.7         0.4  setosa    Specie1
```

As an alternative (since it allows you to make more elaborate conditionals), you can use `case_when()`.

```
# Recoding all 3 cases
df %>%
  mutate(SpeciesAlt = case_when(
    Species == 'setosa' ~ 'Specie1',
    Species == 'versicolor' ~ 'Specie2',
    Species == 'virginica' ~ 'Specie3'
  )) %>%
  head()  # just to show the first 5 rows for the purpose of demonstration.
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species SpeciesAlt
## 1          5.1         3.5          1.4         0.2  setosa    Specie1
## 2          4.9         3.0          1.4         0.2  setosa    Specie1
## 3          4.7         3.2          1.3         0.2  setosa    Specie1
## 4          4.6         3.1          1.5         0.2  setosa    Specie1
## 5          5.0         3.6          1.4         0.2  setosa    Specie1
## 6          5.4         3.9          1.7         0.4  setosa    Specie1
```

```
# Recoding just 2 and giving all the rest the label "others"
df %>%
  mutate(SpeciesAlt = case_when(
    Species == 'setosa' ~ 'Specie1',
    Species == 'versicolor' ~ 'Specie2',
    TRUE ~ 'others'
  )) %>%
  head()  # just to show the first 5 rows for the purpose of demonstration.
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species SpeciesAlt
## 1          5.1         3.5          1.4         0.2  setosa    Specie1
## 2          4.9         3.0          1.4         0.2  setosa    Specie1
## 3          4.7         3.2          1.3         0.2  setosa    Specie1
## 4          4.6         3.1          1.5         0.2  setosa    Specie1
## 5          5.0         3.6          1.4         0.2  setosa    Specie1
## 6          5.4         3.9          1.7         0.4  setosa    Specie1
```

3.6.10 Grouping and Summarizing

`group_by()` and `summarise()`, are two very important functions from `dplyr`. The first one, in itself, does not do anything. It is meant to be followed by the latter.

In the `group_by(variables)` command you tell R on which variables you want to group your data by specifying the column that contains this (or these) variable(s). In the example below the only column that makes sense grouping by is `Species`. By telling R to group with species, the next command `summarise()`

give a summary output for each category of the `Species` column. Lets look at the examples that follow.

```
df <- iris
# Summarising mean Sepal.length by species
df %>%
  group_by(Species) %>%  # Grouping by this variable
  summarise(Mean_By_Species = mean(Sepal.Length))
```

```
## # A tibble: 3 x 2
##   Species    Mean_By_Species
##   <fct>                <dbl>
## 1 setosa                5.01
## 2 versicolor            5.94
## 3 virginica             6.59
```

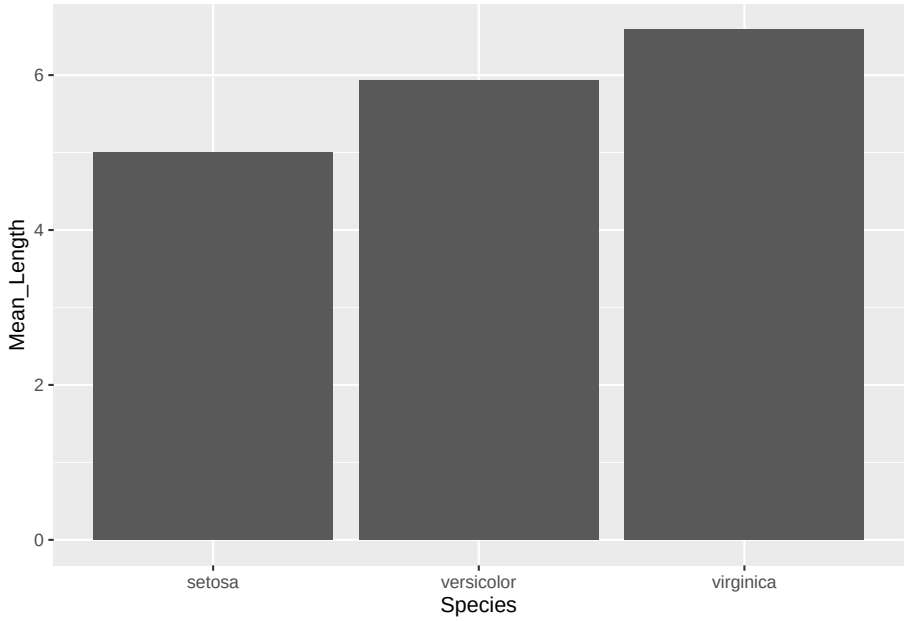
You can group by more than one factor and ask for other summaries, such as median, sd, and other basic operations. For instance:

```
df %>%
  group_by(Species) %>%
  summarise(count = n())  # Gives you the number of entries in each group
```

```
## # A tibble: 3 x 2
##   Species    count
##   <fct>      <int>
## 1 setosa        50
## 2 versicolor    50
## 3 virginica     50
```

You can then build operations on top of your summaries (like mutations or plots)

```
df %>%
  group_by(Species) %>%
  summarise(Mean_Length = mean(Sepal.Length)) %>%
  ggplot(aes(Species, Mean_Length)) +
  geom_col()
```



3.6.11 Changing Format (Wide/Long)

There are two types of ways that the data can be structured in. These ways are important for many reasons, not just for the way they look. Certain analysis, commands or functions used in R prefer (or rather mandate) that the data is in a specific format. This format can be either ***wide*** or ***long***.

In the wide format each each variable level has a column. Lets say we are looking at how people rate pictures of happy, angry and neutral people in terms of good looks on a rating of 0-10. If we were to have the data in a wide format, we would have a data frame with (aside from columns related to the ID of the participant and so forth) 3 columns. One, labeled "Ratings_Happy" for instance, that would have all the ratings given by each participant to the happy faces, another with the ratings given to the angry faces and another to the neutral faces. It should look something like this:

However, if we were to have the data in long format, we would instead have just have two columns (aside from the Participant ID and other information columns you might want). One column, labeled "Facial_Expression" for instance, would have either "Happy", "Angry" or "Neutral". The other column, labeled "Rating", would have the rating given to the face. Since all of the participants rated every condition, each participant would have 3 entries in the dataframe (hence making it longer). It should look something like this:

This is quite simple to do actually. The commands we will be using are `pivot_wider` and `pivot_longer` and they are quite intuitive. Lets work

through this example. Lets first say we want to transform data frame from *long* to *wide*.

```
df_long <- data.frame(Participant_ID = rep(1:5,each=3),
                      Facial_Expression = rep(c('Happy','Angry','Neutral'), 5),
                      Ratings = c(6,4,2,6,4,7,6,5,7,5,8,6,5,8,5))

# Transforming
df_wide <- df_long %>%
  pivot_wider(id_cols = Participant_ID,  # Condition that identifies the grouping (ID)
              names_from = Facial_Expression,  # Where to find our future column names
              values_from = Ratings)   # Where are the values that will fill those col


head(df_wide)
```

```
## # A tibble: 5 x 4
##   Participant_ID Happy Angry Neutral
##            <int> <dbl> <dbl>   <dbl>
## 1              1     6     4       2
## 2              2     6     4       7
## 3              3     6     5       7
## 4              4     5     8       6
## 5              5     5     8       5
```

Now doing the reverse, that is, turning the data from the current *wide* format and making it *longer* again.

```
df_long <- df_wide %>%
  pivot_longer(cols = c('Happy','Angry','Neutral'),  # Columns to turn into long
               names_to = 'Facial_Expression',  # What will the column with the labels
               values_to = 'Ratings')  # What will the column with the values be calle

head(df_long)
```

```
## # A tibble: 6 x 3
##   Participant_ID Facial_Expression Ratings
##            <int> <chr>                <dbl>
## 1              1 Happy                    6
## 2              1 Angry                    4
## 3              1 Neutral                  2
## 4              2 Happy                    6
## 5              2 Angry                    4
## 6              2 Neutral                  7
```

3.6.12 Missing Values

We have several ways of dealing with missing values `NA` (which, if you forget already, means "Not Available"). We can remove them, or omit them, depending on the situation. The important thing to note is that you should be aware if your dataframe contains `NA` values, since these might provide misleading results, or simple provide error messages. For instance, if you ask the mean of a column that contains just one `NA`, the result will be `NA`. You can either specify `na.rm = TRUE` on the command (if the specific command allows you to do so), or just remove the `NA` values prior to running the command.

First lets learn how to check for missing values. There are several ways. Here are a few

```
table(is.na(df))  # tells you how many data points are NAs (TRUE) or not (FALSE) in the whole dat
```

```
##
## FALSE
##   750
```

```
colSums(is.na(df))  # tells you more specifically the number of NAs per column in your dataframe
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##            0            0            0            0            0
```

```
which(colSums(is.na(df))>0)  # just tells you exactly the ones that have NAs (and how many)
```

```
## named integer(0)
```

```
df[!complete.cases(df),]  # tells you the whole row that has an NA value in it.
```

```
## [1] Sepal.Length Sepal.Width  Petal.Length Petal.Width  Species
## <0 rows> (or 0-length row.names)
```

```
df$Sepal.Length <- as.numeric(df$Sepal.Length)
# Asking a mean with NA values
df %>%
  summarise(Mean = mean(Sepal.Length))
```

```
##   Mean
## 1   NA
```

```
# Removing NAs when asking the mean
df %>%
  summarise(Mean = mean(Sepal.Length, na.rm=TRUE))
```

```
##       Mean
## 1 5.843333
```

```
# Removing NAs then asking for the mean
df %>%
  filter(!is.na(Sepal.Length)) %>%
  summarise(Mean = mean(Sepal.Length))
```

```
##       Mean
## 1 5.843333
```

We can remove these NA rows or substitute them.

```
# Replacing NA with 0
df <- df %>%
  mutate(Sepal.Length = ifelse(is.na(Sepal.Length), 0, Sepal.Length))

# Removing
df <- df %>%
  filter(!is.na(Sepal.Length))

# or remove all NA rows
df <- na.omit(df)
```

3.6.13 Counts

Already mentioned above. Gives you the number of entries.

```
# Gives you the number per category of Species
df %>%
  group_by(Species) %>%
  summarise(count = n())
```

```
## # A tibble: 3 x 2
##   Species    count
##   <fct>      <int>
## 1 setosa        51
## 2 versicolor    50
## 3 virginica     50
```

```
# Counts the total number of entries
df %>%
  select(Species) %>%
  count()
```

```
##     n
## 1 151
```

3.6.14 Ungrouping

Lastly, you can use `ungroup()` in a pipe to remove the grouping that you've did, if you want to execute commands over the "ungrouped" data. This is very rarely used, at least by me. However, in certain cases it might be useful. Here's an example, where I want to center the variable `Sepal.Length`, but I want to do so considering the species it belongs to.

```
df %>%
  group_by(Species) %>%  # grouping by species
  mutate(Sepal.Width = as.numeric(Sepal.Width),
         Speal.Length = as.numeric(Sepal.Length)) %>%
  mutate(MeanPerSpecie = mean(Sepal.Width),  # creates mean by species
         CenteredWidth = Sepal.Width - mean(Sepal.Width)) %>%   # subtracts the mean (of the corr
  select(Species, Sepal.Width, MeanPerSpecie, CenteredWidth) %>%
  ungroup()  # remove grouping in case i want to do more mutates, but now NOT considering the gro
```

```
## # A tibble: 151 x 4
##    Species Sepal.Width MeanPerSpecie CenteredWidth
##    <fct>         <dbl>         <dbl>         <dbl>
##  1 setosa          3.5          3.40       0.0980
##  2 setosa          3            3.40      -0.402
##  3 setosa          3.2          3.40      -0.202
##  4 setosa          3.1          3.40      -0.302
##  5 setosa          3.6          3.40       0.198
##  6 setosa          3.9          3.40       0.498
##  7 setosa          3.4          3.40      -0.00196
##  8 setosa          3.4          3.40      -0.00196
##  9 setosa          2.9          3.40      -0.502
## 10 setosa          3.1          3.40      -0.302
## # ... with 141 more rows
```

3.6.15 Strings/Characters

Sometimes we want to work on strings/characters. We may want to replace strings, alter them in some way, split them into different columns, etc. So here

I will introduce a few examples of what we can do to strings in R.

For instance lets say we want to find a pattern of a string in a column of a dataframe in R. For that we will use the grep family of functions which is built-in in R.

```
# We can either find the rows on which this pattern appear
grep('set', df$Species)

# We can pull the string in which this pattern appears
grep('set', df$Species, value = TRUE)

# Or return a TRUE or FALSE per row
grepl('set', df$Species)  # the "l" after grep stands for logic (i.e., TRUE/FALSE)

# We can find how many entries with that pattern are present
sum(grepl('set', df$Species))

# We can substitute a pattern directly in the dataframe
sub('set', 'Set', df$Species)
```

There are additional commands within this family of functions that will allow you to extract, find or substitute exactly what you want and obeying each condition you might want. For that just look into: https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/grep

Another relevant package used to deal with strings is `stringr`, which comes with the tidyverse. Here I'll showing some brief examples of what you can do with it, although you can do much more, and you should check its website: https://stringr.tidyverse.org/

```
# Just preparing the df
df2 <- mtcars
df2$CarName <- rownames(mtcars)
rownames(df2) <- NULL

# StringR
df2 %>%
  mutate(CarName = str_replace(CarName, 'Merc', 'Mercedes'))
```

```
##     mpg cyl  disp  hp drat    wt  qsec vs am gear carb             CarName
## 1  21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4           Mazda RX4
## 2  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4       Mazda RX4 Wag
## 3  22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1          Datsun 710
## 4  21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1      Hornet 4 Drive
## 5  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2   Hornet Sportabout
```

```
## 6  18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1            Valiant
## 7  14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4         Duster 360
## 8  24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2      Mercedes 240D
## 9  22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2       Mercedes 230
## 10 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4       Mercedes 280
## 11 17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4      Mercedes 280C
## 12 16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3     Mercedes 450SE
## 13 17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3     Mercedes 450SL
## 14 15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3    Mercedes 450SLC
## 15 10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4 Cadillac Fleetwood
## 16 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4 Lincoln Continental
## 17 14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4  Chrysler Imperial
## 18 32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1           Fiat 128
## 19 30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2        Honda Civic
## 20 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1     Toyota Corolla
## 21 21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1      Toyota Corona
## 22 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2   Dodge Challenger
## 23 15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2        AMC Javelin
## 24 13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4         Camaro Z28
## 25 19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2   Pontiac Firebird
## 26 27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1          Fiat X1-9
## 27 26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2       Porsche 914-2
## 28 30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2        Lotus Europa
## 29 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4      Ford Pantera L
## 30 19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6        Ferrari Dino
## 31 15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8       Maserati Bora
## 32 21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2          Volvo 142E
```

3.6.16 Splits

We can also split a dataframe into multiple ones, by using `group_split()` or `split()`. They work and do the same. The only difference is that the former comes with the tidyverse and also just works a bit better with the pipes. For instance, lets split the dataframe by species.

```
df %>%
  group_split(Species)
```

```
## <list_of<
##   tbl_df<
##     Sepal.Length: double
##     Sepal.Width : character
##     Petal.Length: character
##     Petal.Width : character
```

```
##     Species     : factor<fb977>
##   >
## >[3]>
## [[1]]
## # A tibble: 51 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl> <chr>       <chr>        <chr>       <fct>
##  1          5.1 3.5         1.4          0.2         setosa
##  2          4.9 3           1.4          0.2         setosa
##  3          4.7 3.2         1.3          0.2         setosa
##  4          4.6 3.1         1.5          0.2         setosa
##  5          5   3.6         1.4          0.2         setosa
##  6          5.4 3.9         1.7          0.4         setosa
##  7          4.6 3.4         1.4          0.3         setosa
##  8          5   3.4         1.5          0.2         setosa
##  9          4.4 2.9         1.4          0.2         setosa
## 10          4.9 3.1         1.5          0.1         setosa
## # ... with 41 more rows
##
## [[2]]
## # A tibble: 50 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl> <chr>       <chr>        <chr>       <fct>
##  1          7   3.2         4.7          1.4         versicolor
##  2          6.4 3.2         4.5          1.5         versicolor
##  3          6.9 3.1         4.9          1.5         versicolor
##  4          5.5 2.3         4            1.3         versicolor
##  5          6.5 2.8         4.6          1.5         versicolor
##  6          5.7 2.8         4.5          1.3         versicolor
##  7          6.3 3.3         4.7          1.6         versicolor
##  8          4.9 2.4         3.3          1           versicolor
##  9          6.6 2.9         4.6          1.3         versicolor
## 10          5.2 2.7         3.9          1.4         versicolor
## # ... with 40 more rows
##
## [[3]]
## # A tibble: 50 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl> <chr>       <chr>        <chr>       <fct>
##  1          6.3 3.3         6            2.5         virginica
##  2          5.8 2.7         5.1          1.9         virginica
##  3          7.1 3           5.9          2.1         virginica
##  4          6.3 2.9         5.6          1.8         virginica
##  5          6.5 3           5.8          2.2         virginica
##  6          7.6 3           6.6          2.1         virginica
##  7          4.9 2.5         4.5          1.7         virginica
```

```
##  8          7.3 2.9         6.3          1.8         virginica
##  9          6.7 2.5         5.8          1.8         virginica
## 10          7.2 3.6         6.1          2.5         virginica
## # ... with 40 more rows
```

3.6.17 Mapping

Mapping is quite useful. It allows you to map a function to a certain output. For instance, if you first need to split the dataframe, then perform a correlation test, you can easily do this altogether.

```
df %>%
  mutate(Sepal.Length = as.numeric(Sepal.Length),  # turning these columns to numeric
         Sepal.Width = as.numeric(Sepal.Width)) %>%
  group_split(Species) %>%  # split by pictures
  map(~ cor.test(.$Sepal.Length, .$Sepal.Width))
```

```
## [[1]]
##
##  Pearson's product-moment correlation
##
## data:  .$Sepal.Length and .$Sepal.Width
## t = 6.7473, df = 49, p-value = 1.634e-08
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.5173516 0.8139116
## sample estimates:
##       cor
## 0.6939905
##
##
## [[2]]
##
##  Pearson's product-moment correlation
##
## data:  .$Sepal.Length and .$Sepal.Width
## t = 4.2839, df = 48, p-value = 8.772e-05
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.2900175 0.7015599
## sample estimates:
##       cor
## 0.5259107
##
```

```
##
## [[3]]
##
##  Pearson's product-moment correlation
##
## data:  .$Sepal.Length and .$Sepal.Width
## t = 3.5619, df = 48, p-value = 0.0008435
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.2049657 0.6525292
## sample estimates:
##       cor
## 0.4572278
```

We can see it more clearly in this dataframe.

```
mtcars %>%
  split(.$cyl) %>%
  map(~ lm(mpg ~ wt, data = .x)) %>%
  map_dfr(~ as.data.frame(t(as.matrix(coef(.))))  # returns the result in a dataframe
```

```
##   (Intercept)        wt
## 1    39.57120 -5.647025
## 2    28.40884 -2.780106
## 3    23.86803 -2.192438
```

3.7 End

Chapter 4

Exploratory Data Analysis (EDA)

In Exploratory Data Analysis (EDA for short) we will want to explore our data. We usually start by getting summaries and plots that broadly summarise how our data is structured. We then proceed to more specific exploration of what variables interest us. There is no right way of doing EDA, and everyone does EDA slightly different. What matters in the end is that this part of your data analysis gives you a good glimpse about your data and potential targets to further explore and analyze. And of course, this will be a good time to put your plotting skills to test, and with R, you can do pretty much anything… and, again, I truly mean anything.

What is the data we will be working with? The data exemplified here comes from Dr. Kristen Gorman and the Palmer Station, Antarctica LTER, a member of the Long Term Ecological Research Network. It contains information about 344 penguins, across 3 different species of penguins, collected from 3 islands in the Palmer Archipelago, Antarctica.

Horst AM, Hill AP, Gorman KB (2020). palmerpenguins: Palmer Archipelago (Antarctica) penguin data. R package version 0.1.0. https://allisonhorst.github.io/palmerpenguins/

Now lets look at the data We can either show the first rows (or last rows) to get a general view on what the columns are and what the values associated with each columns are, or we can View the entire dataframe. Additionally we can also use several commands to provide us some more general info about the dataframe itself.

4.1 Loading data

```
penguins_raw <- palmerpenguins::penguins_raw
head(penguins_raw)
```

```
## # A tibble: 6 x 17
##   studyName `Sample Number` Species          Region Island Stage `Individual ID`
##   <chr>               <dbl> <chr>            <chr>  <chr>  <chr> <chr>
## 1 PAL0708                 1 Adelie Penguin ~ Anvers Torge~ Adul~ N1A1
## 2 PAL0708                 2 Adelie Penguin ~ Anvers Torge~ Adul~ N1A2
## 3 PAL0708                 3 Adelie Penguin ~ Anvers Torge~ Adul~ N2A1
## 4 PAL0708                 4 Adelie Penguin ~ Anvers Torge~ Adul~ N2A2
## 5 PAL0708                 5 Adelie Penguin ~ Anvers Torge~ Adul~ N3A1
## 6 PAL0708                 6 Adelie Penguin ~ Anvers Torge~ Adul~ N3A2
## # ... with 10 more variables: `Clutch Completion` <chr>, `Date Egg` <date>,
## #   `Culmen Length (mm)` <dbl>, `Culmen Depth (mm)` <dbl>,
## #   `Flipper Length (mm)` <dbl>, `Body Mass (g)` <dbl>, Sex <chr>,
## #   `Delta 15 N (o/oo)` <dbl>, `Delta 13 C (o/oo)` <dbl>, Comments <chr>
```

```
# Use tails() if you want to see the last rows
```

The `str()` command gives us some detail into the class of each column of the data frame, its length, its values, among others.

P.s., `<dbl>` means a numerical value with decimal points.

4.2 Checking data

```
str(penguins_raw)
```

```
## tibble [344 x 17] (S3: tbl_df/tbl/data.frame)
##  $ studyName          : chr [1:344] "PAL0708" "PAL0708" "PAL0708" "PAL0708" ...
##  $ Sample Number      : num [1:344] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Species            : chr [1:344] "Adelie Penguin (Pygoscelis adeliae)" "Adelie Pe
##  $ Region             : chr [1:344] "Anvers" "Anvers" "Anvers" "Anvers" ...
##  $ Island             : chr [1:344] "Torgersen" "Torgersen" "Torgersen" "Torgersen"
##  $ Stage              : chr [1:344] "Adult, 1 Egg Stage" "Adult, 1 Egg Stage" "Adult
##  $ Individual ID      : chr [1:344] "N1A1" "N1A2" "N2A1" "N2A2" ...
##  $ Clutch Completion  : chr [1:344] "Yes" "Yes" "Yes" "Yes" ...
##  $ Date Egg           : Date[1:344], format: "2007-11-11" "2007-11-11" ...
```

```
##  $ Culmen Length (mm) : num [1:344] 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
##  $ Culmen Depth (mm)  : num [1:344] 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
##  $ Flipper Length (mm): num [1:344] 181 186 195 NA 193 190 181 195 193 190 ...
##  $ Body Mass (g)      : num [1:344] 3750 3800 3250 NA 3450 ...
##  $ Sex                : chr [1:344] "MALE" "FEMALE" "FEMALE" NA ...
##  $ Delta 15 N (o/oo)  : num [1:344] NA 8.95 8.37 NA 8.77 ...
##  $ Delta 13 C (o/oo)  : num [1:344] NA -24.7 -25.3 NA -25.3 ...
##  $ Comments           : chr [1:344] "Not enough blood for isotopes." NA NA "Adult not sampled.
##  - attr(*, "spec")=
##   .. cols(
##   ..   studyName = col_character(),
##   ..   `Sample Number` = col_double(),
##   ..   Species = col_character(),
##   ..   Region = col_character(),
##   ..   Island = col_character(),
##   ..   Stage = col_character(),
##   ..   `Individual ID` = col_character(),
##   ..   `Clutch Completion` = col_character(),
##   ..   `Date Egg` = col_date(format = ""),
##   ..   `Culmen Length (mm)` = col_double(),
##   ..   `Culmen Depth (mm)` = col_double(),
##   ..   `Flipper Length (mm)` = col_double(),
##   ..   `Body Mass (g)` = col_double(),
##   ..   Sex = col_character(),
##   ..   `Delta 15 N (o/oo)` = col_double(),
##   ..   `Delta 13 C (o/oo)` = col_double(),
##   ..   Comments = col_character()
##   .. )
```

```
# You can also use glimpse() from the dplyr package
# Or you can use the skim() function from the skimr package
```

The `summary` command gives us a summary of each column, with, in case of numeric, the distribution parameters of those values, and in the case of categorical columns, the count of each value while also highlighting always the number of `NA` entries.

```
summary(penguins_raw)
```

```
##   studyName         Sample Number      Species             Region
##  Length:344         Min.   :  1.00   Length:344         Length:344
##  Class :character   1st Qu.: 29.00   Class :character   Class :character
##  Mode  :character   Median : 58.00   Mode  :character   Mode  :character
##                     Mean   : 63.15
##                     3rd Qu.: 95.25
```

```
##                    Max.   :152.00
##
##     Island             Stage           Individual ID      Clutch Completion
##  Length:344         Length:344         Length:344         Length:344
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##
##     Date Egg           Culmen Length (mm) Culmen Depth (mm) Flipper Length (mm)
##  Min.   :2007-11-09   Min.   :32.10      Min.   :13.10     Min.   :172.0
##  1st Qu.:2007-11-28   1st Qu.:39.23      1st Qu.:15.60     1st Qu.:190.0
##  Median :2008-11-09   Median :44.45      Median :17.30     Median :197.0
##  Mean   :2008-11-27   Mean   :43.92      Mean   :17.15     Mean   :200.9
##  3rd Qu.:2009-11-16   3rd Qu.:48.50      3rd Qu.:18.70     3rd Qu.:213.0
##  Max.   :2009-12-01   Max.   :59.60      Max.   :21.50     Max.   :231.0
##                       NA's   :2          NA's   :2         NA's   :2
##  Body Mass (g)      Sex            Delta 15 N (o/oo) Delta 13 C (o/oo)
##  Min.   :2700   Length:344         Min.   : 7.632    Min.   :-27.02
##  1st Qu.:3550   Class :character   1st Qu.: 8.300    1st Qu.:-26.32
##  Median :4050   Mode  :character   Median : 8.652    Median :-25.83
##  Mean   :4202                      Mean   : 8.733    Mean   :-25.69
##  3rd Qu.:4750                      3rd Qu.: 9.172    3rd Qu.:-25.06
##  Max.   :6300                      Max.   :10.025    Max.   :-23.79
##  NA's   :2                         NA's   :14        NA's   :13
##    Comments
##  Length:344
##  Class :character
##  Mode  :character
##
##
##
##
```

At this phase you should start assessing your data and will probably need to modify the dataframe a bit. Most of the tools you need were already introduced in Chapter 2, but you will learn some new ones here along the way. First though, lets get familiarized with the data by reading its description.

So a brief intro to this data. This data measures structural size of adult male and female Adélie penguins (Pygoscelis adeliae) nesting along the Palmer Archipelago near Palmer Station. It has 17 columns, namely:

studyName: Sampling expedition from which data were collected, generated, etc.

Sample Number: an integer denoting the continuous numbering sequence for each sample

Species: a character string denoting the penguin species

Region: a character string denoting the region of Palmer LTER sampling grid

Island: a character string denoting the island near Palmer Station where samples were collected

Stage: a character string denoting reproductive stage at sampling

Individual ID: a character string denoting the unique ID for each individual in dataset

Clutch Completion: a character string denoting if the study nest observed with a full clutch, i.e., 2 eggs

Date Egg: a date denoting the date study nest observed with 1 egg (sampled)

Culmen Length: a number denoting the length of the dorsal ridge of a bird's bill (millimeters)

Culmen Depth: a number denoting the depth of the dorsal ridge of a bird's bill (millimeters)

Flipper Length: an integer denoting the length penguin flipper (millimeters)

Body Mass: an integer denoting the penguin body mass (grams)

Sex: a character string denoting the sex of an animal

Delta 15 N: a number denoting the measure of the ratio of stable isotopes 15N:14N

Delta 13 C: a number denoting the measure of the ratio of stable isotopes 13C:12C

Comments: a character string with text providing additional relevant information for data

There is also a subsetted (cleaner) version with just the species, island, size (flipper length, body mass, bill dimensions) and sex variables. We will, however, work with the raw version and transform it ourselves making it cleaner and easier to explore and gather some initial insights about the data.

```
head(penguins_raw)
```

```
## # A tibble: 6 x 17
##   studyName `Sample Number` Species          Region Island Stage `Individual ID`
##   <chr>               <dbl> <chr>            <chr>  <chr>  <chr> <chr>
## 1 PAL0708                 1 Adelie Penguin ~ Anvers Torge~ Adul~ N1A1
## 2 PAL0708                 2 Adelie Penguin ~ Anvers Torge~ Adul~ N1A2
```

```
## 3 PAL0708                 3 Adelie Penguin ~ Anvers Torge~ Adul~ N2A1
## 4 PAL0708                 4 Adelie Penguin ~ Anvers Torge~ Adul~ N2A2
## 5 PAL0708                 5 Adelie Penguin ~ Anvers Torge~ Adul~ N3A1
## 6 PAL0708                 6 Adelie Penguin ~ Anvers Torge~ Adul~ N3A2
## # ... with 10 more variables: `Clutch Completion` <chr>, `Date Egg` <date>,
## #   `Culmen Length (mm)` <dbl>, `Culmen Depth (mm)` <dbl>,
## #   `Flipper Length (mm)` <dbl>, `Body Mass (g)` <dbl>, Sex <chr>,
## #   `Delta 15 N (o/oo)` <dbl>, `Delta 13 C (o/oo)` <dbl>, Comments <chr>
```

4.3 Cleaning

4.3.1 Cleaning col. names

So first off, we might want to clean the data. One thing we often do is rename variables (in this case columns). We can do so simply with the use of the `rename()` function from `dplyr`.
`rename()`: changes the names of individual variables using new_name = old_name syntax.
`rename_with()`: renames columns using a function.

```
# Lets say I want to rename "Individual ID" to "ID".
penguins <- penguins_raw %>%
  rename('ID' = 'Individual ID')
```

Now we might want to make every column, except ID, lower case, substitute spaces with "_" and remove "()".

```
penguins <- penguins %>%
  rename_with(tolower) %>%  # lower-case every column
  rename_with(toupper, starts_with('ID')) %>% # up-case column starting with "ID"
  rename_with(~gsub(" ", "_", .x))  %>% # subs every space (" ") with no space ("_")
  rename_with(~gsub("\\(", "", .x)) %>%  #removes every "("
  rename_with(~gsub("\\)", "", .x))  %>% # removes every ")"
  rename_with(~gsub("/" ,"", .x))  # removes /
```

We could also skip a few things and be less specific, but instead use only the following function from the "janitor" package

```
penguins %>%
  janitor::clean_names()
```

```
## # A tibble: 344 x 17
```

```
##    studyname sample_number species    region island stage id    clutch_completi~
##    <chr>             <dbl> <chr>      <chr>  <chr>  <chr> <chr> <chr>
##  1 PAL0708               1 Adelie Pe~ Anvers Torge~ Adul~ N1A1  Yes
##  2 PAL0708               2 Adelie Pe~ Anvers Torge~ Adul~ N1A2  Yes
##  3 PAL0708               3 Adelie Pe~ Anvers Torge~ Adul~ N2A1  Yes
##  4 PAL0708               4 Adelie Pe~ Anvers Torge~ Adul~ N2A2  Yes
##  5 PAL0708               5 Adelie Pe~ Anvers Torge~ Adul~ N3A1  Yes
##  6 PAL0708               6 Adelie Pe~ Anvers Torge~ Adul~ N3A2  Yes
##  7 PAL0708               7 Adelie Pe~ Anvers Torge~ Adul~ N4A1  No
##  8 PAL0708               8 Adelie Pe~ Anvers Torge~ Adul~ N4A2  No
##  9 PAL0708               9 Adelie Pe~ Anvers Torge~ Adul~ N5A1  Yes
## 10 PAL0708              10 Adelie Pe~ Anvers Torge~ Adul~ N5A2  Yes
## # ... with 334 more rows, and 9 more variables: date_egg <date>,
## #   culmen_length_mm <dbl>, culmen_depth_mm <dbl>, flipper_length_mm <dbl>,
## #   body_mass_g <dbl>, sex <chr>, delta_15_n_ooo <dbl>, delta_13_c_ooo <dbl>,
## #   comments <chr>
```

4.3.2 Date

One interesting and somewhat difficult (sometimes) parameter to adjust is turning a column that is often a character to a `date` format. In this case is simple, but I give you some examples below for you to work your way through other types of transformations.

```
penguins$date_egg <- as.Date(penguins$date_egg)
```

Now here are other examples, aside from this data.

```
characters <- c("1jan1960", "2jan1960", "31mar1960", "30jul1960")
dates <- as.Date(characters, "%d%b%Y")
dates
```

```
## [1] "1960-01-01" "1960-01-02" "1960-03-31" "1960-07-30"
```

```
characters <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
dates <- as.Date(characters, "%m/%d/%y")
dates
```

```
## [1] "1992-02-27" "1992-02-27" "1992-01-14" "1992-02-28" "1992-02-01"
```

Check also these link for better examples. https://www.r-bloggers.com/2013/08/date-formats-in-r/

4.3.3 Cleaning some columns

In particular, I want to remove the species latin name from the "species" column as well as the parenthesis. Now this may seem simple, but messing with characters is one of those things that having an internet connection is key, since base commands, in my opinion, are not at all clear or simple. The command below simple means replace the `species` column with a new one (with the same name), but in this one replace all the text that is between the parenthesis (including the parenthesis themselves) `" *\\(.*?\\) *"` and replace by nothing `""`.

```
penguins <- penguins %>%
  mutate(species = gsub(" *\\(.*?\\) *", "", species))
```

Another thing that jumps to sight is the "stage" column, which appears to be telling the same thing always. Lets check.

```
unique(penguins$stage)
```

```
## [1] "Adult, 1 Egg Stage"
```

Since its all the same, I can choose to remove it.

```
penguins$stage <- NULL
```

4.3.4 Dealing with NA values

Now lets deal with NA values. First lets quickly identify which columns have NA values, and how many of them each one has. We could just look at the `summary(penguins)`, or we could do this with other base R commands, like so:

```
sapply(penguins, function(x) sum(is.na(x)))
```

```
##         studyname     sample_number           species            region
##                 0                 0                 0                 0
##            island                ID clutch_completion         date_egg
##                 0                 0                 0                 0
##  culmen_length_mm   culmen_depth_mm flipper_length_mm       body_mass_g
##                 2                 2                 2                 2
##               sex    delta_15_n_ooo    delta_13_c_ooo          comments
##                11                14                13               290
```

```
apply(is.na(penguins), 2, sum)
```

```
##          studyname      sample_number            species             region
##                  0                  0                  0                  0
##             island                 ID  clutch_completion          date_egg
##                  0                  0                  0                  0
##   culmen_length_mm    culmen_depth_mm  flipper_length_mm        body_mass_g
##                  2                  2                  2                  2
##                sex     delta_15_n_ooo     delta_13_c_ooo           comments
##                 11                 14                 13                290
```

Note: Go to https://www.guru99.com/r-apply-sapply-tapply.html to better understand "apply" and other functions.

Or we can do something more elegant and use some packages that show us, with graphics, where our NAs are. Here are a few examples that I took from a quick google search:

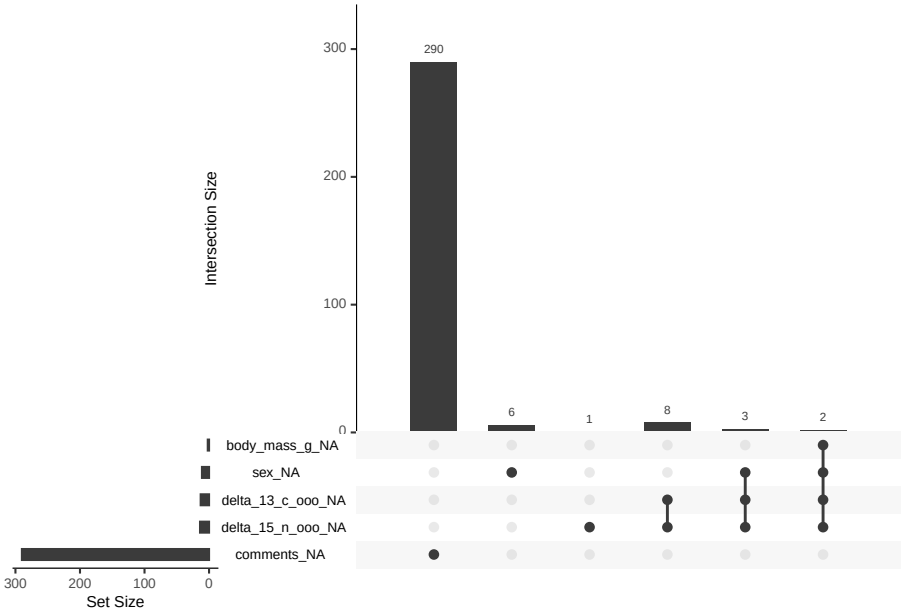
```
# Alternative 1
library(naniar)
```

```
## Warning: package 'naniar' was built under R version 4.1.3
```

```
library(UpSetR)
```

```
## Warning: package 'UpSetR' was built under R version 4.1.3
```

```
penguins %>%
  as_shadow_upset() %>%
  upset()
```

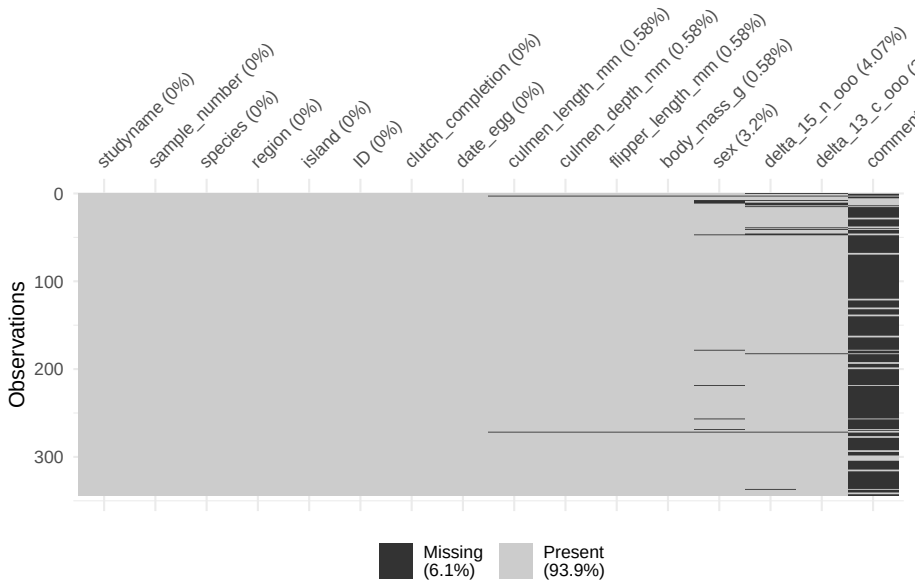



```
# Alternative n2
library(visdat)
```

```
## Warning: package 'visdat' was built under R version 4.1.3
```

```
vis_miss(penguins)
```

```
## Warning: `gather_()` was deprecated in tidyr 1.2.0.
## Please use `gather()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated
```



Now, we need to decide what to do with these NA values. We can see below that two penguins have data missing from their base features (e.g., culmen_lenght and body_mass). Also, a few of them don't have information regarding sex. Since these are critical features, I'm gonna go ahead and eliminate these entries. We can do so by several ways, but below you have two examples on how to eliminate NA values.

```
# Example 1
penguins <- penguins %>%
  filter(!is.na(culmen_length_mm),  # show only rows that are not NA in the culmen_length_mm colu
         !is.na(sex)) # show only rows that are not NA in the sex column

# Example 2
penguins <- penguins %>%
  drop_na(culmen_length_mm, sex)
```

Now for some final touches, I will transform the body mass column's units from grams to kilograms.

```
# Converting body_mass from grams to kg
penguins <- penguins %>%
  mutate(body_mass = body_mass_g/1000) %>%
  mutate(body_mass_g = NULL)  # removing old column
```

To finish this data cleaning, its important to point out that we could have just

done everything in one go, as exemplified below. Although as you can see, when dealing with too many things, its usually best to separate adjustments per chunks.

```
penguins <- penguins_raw %>%
  rename('ID' = 'Individual ID') %>%
  rename_with(tolower) %>%  # lower-case every column
  rename_with(toupper, starts_with('ID')) %>% # up-case column starting with "ID"
  rename_with(~gsub(" ", "_", .x))  %>% # subs every space (" ") with no space ("_")
  rename_with(~gsub("\\(", "", .x)) %>%  #removes every "("
  rename_with(~gsub("\\)", "", .x))  %>% # removes every ")"
  rename_with(~gsub("/" ,"", .x)) %>% # removes /
  mutate(date_egg = as.Date(date_egg)) %>%  # turns the column to data format
  mutate(species = gsub(" *\\(.*?\\) *", "", species)) %>% # removes text between pare
  mutate(stage = NULL) %>%  # removes stage column
  filter(!is.na(culmen_length_mm), !is.na(sex)) %>%   # removes NA rows from these 2 c
  mutate(body_mass = body_mass_g/1000) %>%   # transforms body_mass_g column to kilogr
  mutate(body_mass_g = NULL)
```

4.4 Summarising

Before we start drawing graphs left and right, its important to also explore the data with data summaries (in text format). As mentioned, from my perspective, EDA isn't an exact science, in a sense that there is no appropriate set of measures to explore. Knowing what to do comes from both a) knowing your data, b) what your objectives are and c) experience. Below I'm going to try to cover some aspects of EDA (first in text than with plots) that I would do when analysing this data. Lets start…

So, how many observations per specie are there in the data?

```
penguins %>%
  count(species)
```

```
## # A tibble: 3 x 2
##   species              n
##   <chr>            <int>
## 1 Adelie Penguin     146
## 2 Chinstrap penguin   68
## 3 Gentoo penguin     119
```

Alternatively, we can use the functions `tabyl()` and `adorn_totals()` from the janitor package and create a more elaborate summary with percentages and totals.

```
penguins %>%
  janitor::tabyl(species) %>%
  janitor::adorn_totals()
```

```
##            species   n   percent
##     Adelie Penguin 146 0.4384384
##  Chinstrap penguin  68 0.2042042
##     Gentoo penguin 119 0.3573574
##              Total 333 1.0000000
```

Alright, now lets say we want to know how they are distributed across islands

```
penguins %>%
  group_by(island) %>%
  count(species)
```

```
## # A tibble: 5 x 3
## # Groups:   island [3]
##   island    species              n
##   <chr>     <chr>            <int>
## 1 Biscoe    Adelie Penguin      44
## 2 Biscoe    Gentoo penguin     119
## 3 Dream     Adelie Penguin      55
## 4 Dream     Chinstrap penguin   68
## 5 Torgersen Adelie Penguin      47
```

What seem to be the main problems presented in the observation column?

```
penguins %>%
  filter(!is.na(comments)) %>%
  count(comments)
```

```
## # A tibble: 4 x 2
##   comments                                                          n
##   <chr>                                                         <int>
## 1 Nest never observed with full clutch.                            34
## 2 Nest never observed with full clutch. Not enough blood for isotopes.     1
## 3 No delta15N data received from lab.                              1
## 4 Not enough blood for isotopes.                                   7
```

Since we spotted that some comments only appear very often we could quickly do some data cleaning again. Namely we might wanna combine certain levels of a factor or characters that are less represented (compared to other levels) together,

instead of representing each. In this case we would combine less represented types of comments into a single label (e.g., "OtherProblems"). For that we could use `fct_lump()`. This function "lumps" (joins) together factors or characters with little representation. We just need to say which column we want to use this on, and we can define other parameters of interest. These are:

- **n**: (specifying which of the most common to preserve, default being 1). If I say `n = 2` only the two most common comments will be preserved, with the rest being labeled as "OtherProblems".
- **p**: Alternative to "n" in which we can specify to preserve only the levels/characters which have at least a proportion of x. Again, if we said p = 0.2, only the comments which are represented in 20% of the data would be kept.
- **other_level**: Here you set the name attributed to the lumped factors. In this case we say "OtherProblems".

```
penguins %>%
  mutate(comments2 = fct_lump(f=comments, n = 1, other_level = 'OtherProblems')) %>%
  View()
```

Anyway, back to exploring the data... Now lets ask the mean for each specie `across` all columns in the dataframe, `where` the values are numeric. We can do this, by simply grouping the data by species, that using the command below.

```
penguins %>%
  group_by(species) %>%
  summarise(across(where(is.numeric), mean, na.rm = TRUE)) %>%
  View()
```

4.5 Plots

4.5.1 Formula

https://r-graph-gallery.com/ R has a natural plot language, but its quite "primitive" and complicated. It is adequate if you want really quick plots with minimal customization. Otherwise I would recommend using `ggplot2`.

`ggplot2` uses a distinct grammar for expression plots. It may seem complicated at first, but its actually quite simple. So every plot must start with a simple command:

```
ggplot2("dataframe", aes("data"))
```

In the first parameter you just tell the command which dataframe to use. Assuming your dataframe is labeled as "df" you can just the following:

```
ggplot(df, aes("data"))
```

or

```
df %>% ggplot(aes("data"))
```

The next thing we need to do is specify which variables and measures we want to add to the plot. This is done inside the `aes()` command. You can add the obvious x and y info, and you can also add group related info, which can be coded as "color", "fill" or "shape". Here's a few generic examples:

```
ggplot(df, aes(x = weight, y = height))

ggplot(df, aes(x = education_level, y = average_grade))

ggplot(df, aes(x = country, y = height, color = sex))

ggplot(df, aes(x = country, y = average_reading_time,
               color = sex, shape = genre))
```

After this is specified, you just need to add layers of what you want. By adding, I mean inserting a `+` and specifying the layers you want. Do you want points? Bars? Lines? Circles? Whatever you want, you'll add each as a layer with `geom_something()`, with the something corresponding to what you want. `ggplot2` has plenty of geoms, and I'll be showing you a few below. If you want to see what the rest can do, visit: https://ggplot2.tidyverse.org/reference/

So your general plot code for a simple bar plot should look something like:

```
ggplot(dataframe, aes(x = variable, y = variable2)) +
  geom_bar()
```

Lastly, you might want to specify other little details, such as the labels, the theme, the color scheme, etc. You can specify hundreds of things. I'm just gonna leave you the commands I use the most and I think will be most useful to you.

```
# Specifying the labels
labs(x = 'x-axis label', y = 'y-axis label', title = 'title text', color = 'color label text')

# Specifying the lower and upper bounds of the axis
coord_cartesian(ylim = c('lower_y_limit', 'upper_y_limit'),
                xlim = c('lower_x_limit', 'upper_y_limit'))
```

```
# Splitting the graphs creating several based on a grouping variable
facet_wrap(~'grouping_variable')

# Themes (just some examples)
theme_classic()
theme_light()
theme_gray()
```

If you want to save your plots you can do so either by clicking in the "Export" button on your Plots separator or you can use the function `ggsave()`.

Well this should give you a general idea on how plots work. For more info, visit: http://r-statistics.co/Complete-Ggplot2-Tutorial-Part1-With-R-Code.html

On to some examples with our data.

Note: I purposely leave some additional parameters inside each plot building code, not to confuse you, but to introduce you more ways to tweak your plot. I encourage you to mess with them.

4.5.2 Cols and Bars

The bar plot is one of the most common ones. R has two types of "bar" plots.

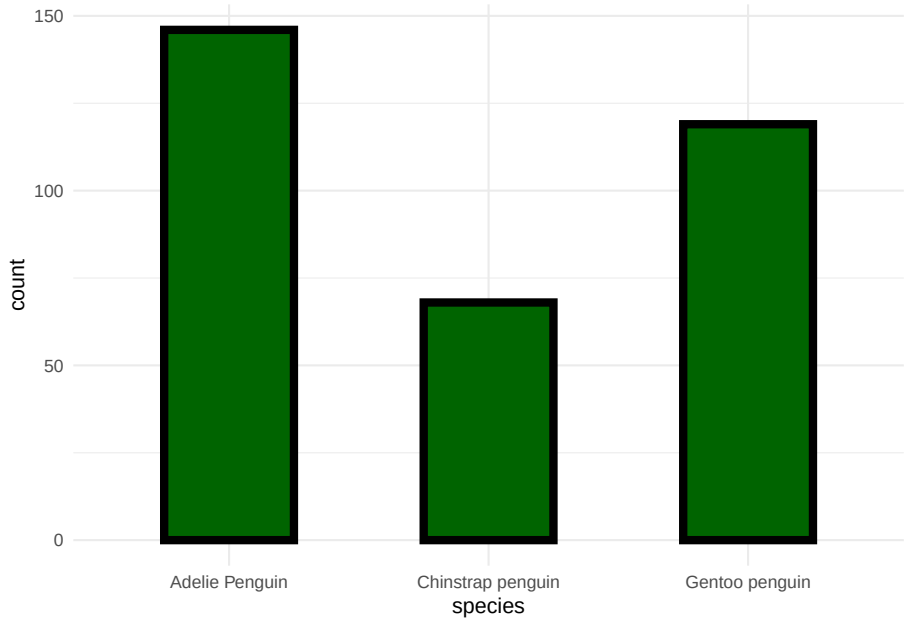
- `geom_bar`: Creates proportional count of entries (same as `count` but in a plot). Only requires one variable.
- `geom_col`: Creates a more typical bar plot, where the height represents values/statistics of the data,

Here's an example usage of `geom_bar`

```
# Count
penguins %>%
  count(species)
```

```
## # A tibble: 3 x 2
##   species               n
##   <chr>             <int>
## 1 Adelie Penguin      146
## 2 Chinstrap penguin    68
## 3 Gentoo penguin      119
```

```
# Geom bar (count as a plot)
penguins %>%
  ggplot(aes(species)) +
  geom_bar(fill = 'darkgreen', color = 'black', size = 2, width = .5) +
  theme_minimal()
```



Now regarding `geom_col` and the more typical use of showing means. You have two ways of doing it, either:

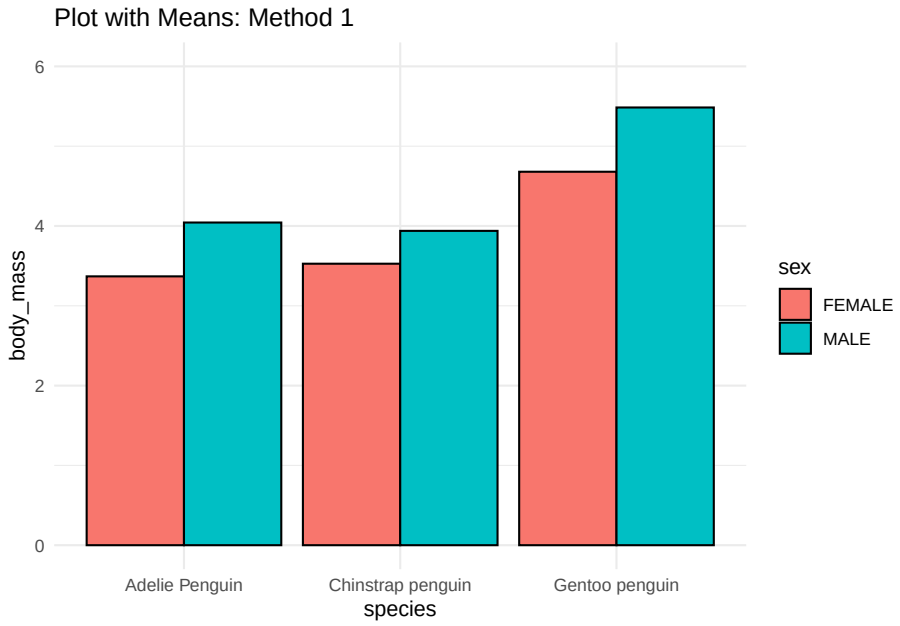
1. you compute the means and build the plot with `geom_col()`
2. you use the `stat_summary()` function to compute the mean inside the ggplot and specify that you want the "col" geom (or "bar" geom).

I usually just use `stat_summary()`, instead of pre-computing the means, because its faster and this allows me to add another function with `stat_summary()` that can also compute things like error bars.

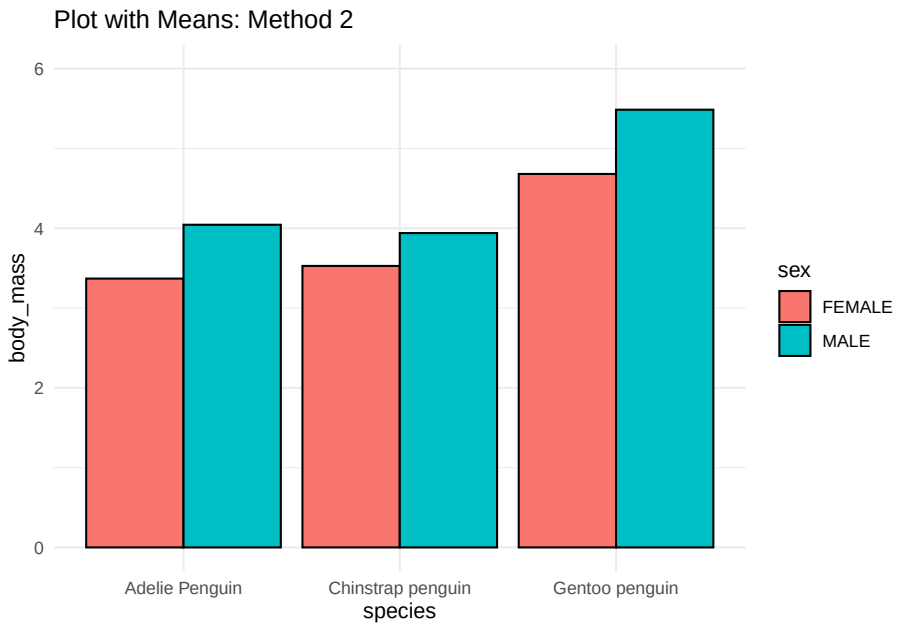
```
# 1.
penguins %>%
  group_by(species, sex) %>%
  summarise(body_mass = mean(body_mass)) %>%
  ggplot(aes(species, body_mass, fill = sex)) +
  geom_col(position = position_dodge(.9), color = 'black') +
```

```
  coord_cartesian(ylim = c(0, 6)) +
  theme_minimal() +
  labs(title = 'Plot with Means: Method 1')
```

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



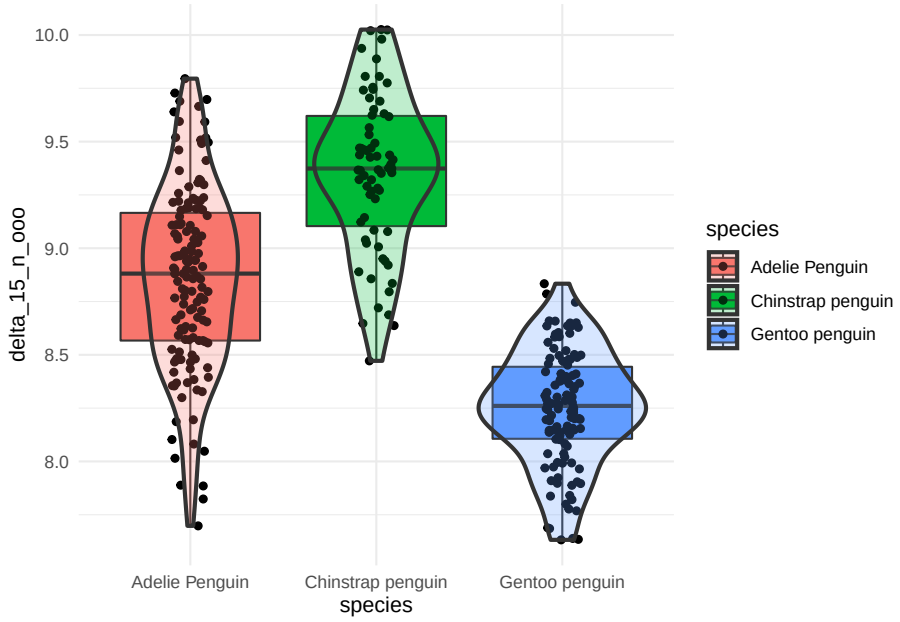

```
# 2.
penguins %>%
  ggplot(aes(species, body_mass, fill = sex)) +
  stat_summary(fun = mean, geom = 'col', position = position_dodge(.9),
               color = 'black') +
  coord_cartesian(ylim = c(0, 6)) +
  theme_minimal() +
  labs(title = 'Plot with Means: Method 2')
```



4.5.3 Boxplots

Boxplots are a really cool way of showing you how your data is distributed across each category. Its particularly important when you have skewed data, or are just simply more interested in the median (as opposed to the mean). Below I introduce you the standard box plot, while also showing you other layers that can both appear on their own or can complement the boxplot geom, namely `geom_jitter()` and `geom_violin()`.

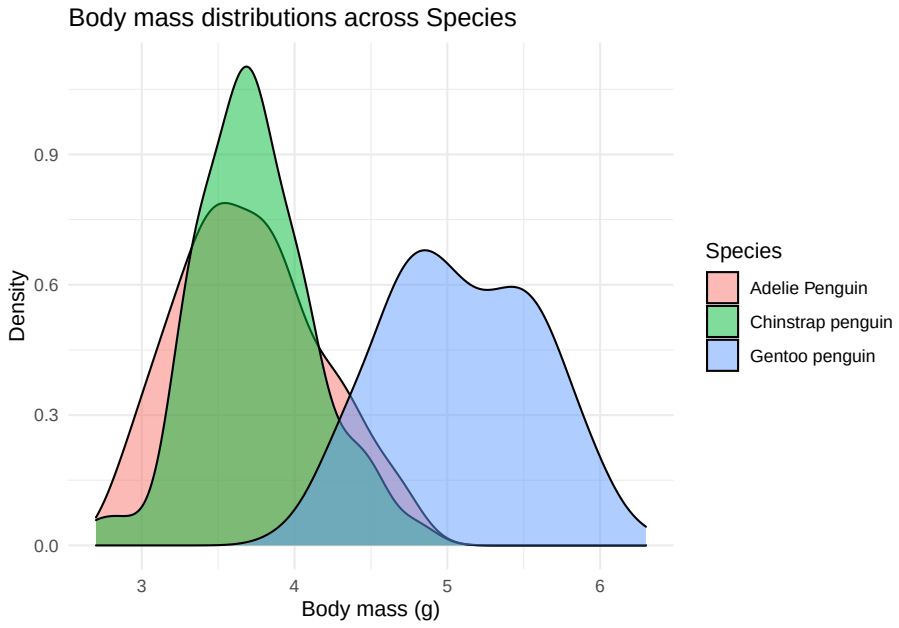
```
penguins %>%
  filter(!is.na(delta_15_n_ooo)) %>%  # remove NAs in the column to prevent warnings
  ggplot(aes(x = species, y = delta_15_n_ooo, fill = species)) +
  geom_boxplot() +
  geom_jitter(width = .1) +
  geom_violin(alpha = .25, size = 1) +  # the alpha parameter models transparency (ranges from 0
  theme_minimal()
```



4.5.4 Density

Density plots are great and provide a simple way for you to measure distribution of a variable's values. You just need to point out the variable name, in this case we will look at "body_mass", and we will additionally want the distributions separated by "species".

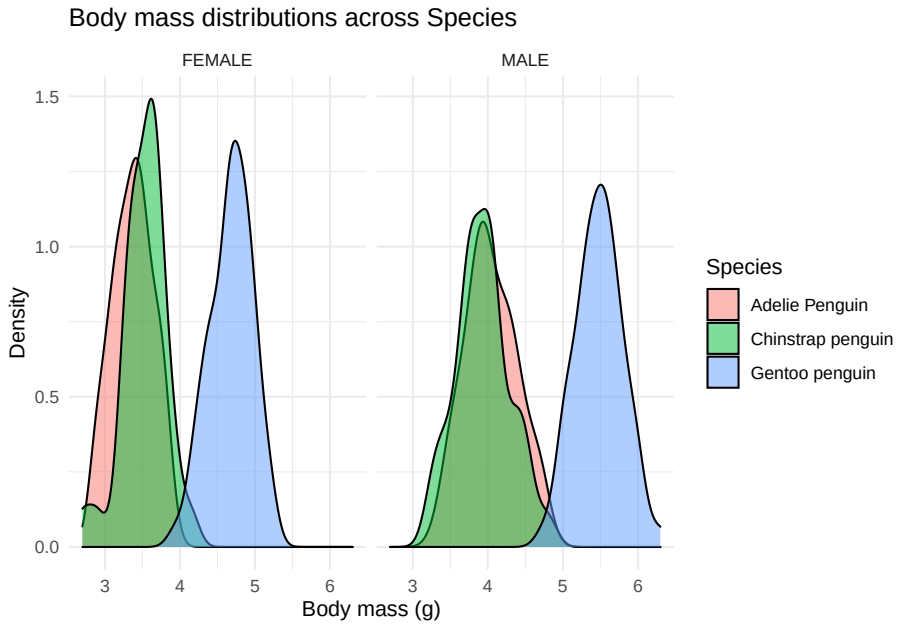
```
penguins %>%
  ggplot(aes(x = body_mass, fill = species)) +
  geom_density(alpha = .5) +
  theme_minimal() +
  labs(x = 'Body mass (g)', y = 'Density',
       fill = 'Species',
       title = 'Body mass distributions across Species') +
  scale_color_brewer(palette = "Dark2")
```



4.5.5 Facets

`facet_wrap` is quite useful if you want to compare two plots across 2 (or more) variables. It divides the data across the variable you name and creates two distinct plots. You just need to specify the variable, which in this case in done so with `~name_of_the_variable`. Importantly, if you want the scales (y-limits and x-limits) to be adjusted for each variable's info, you need to specify this using `scales = 'free'` if every scale (x and y) can vary between facets, or `scales = 'free_x'` and `scales = 'free_y'`, if you just want the x or y scales to vary freely, respetively. In this case since we want to compare them directly, I think its best if they are fixed (the default behavior of the function), so you get a better picture as to how different they really are.

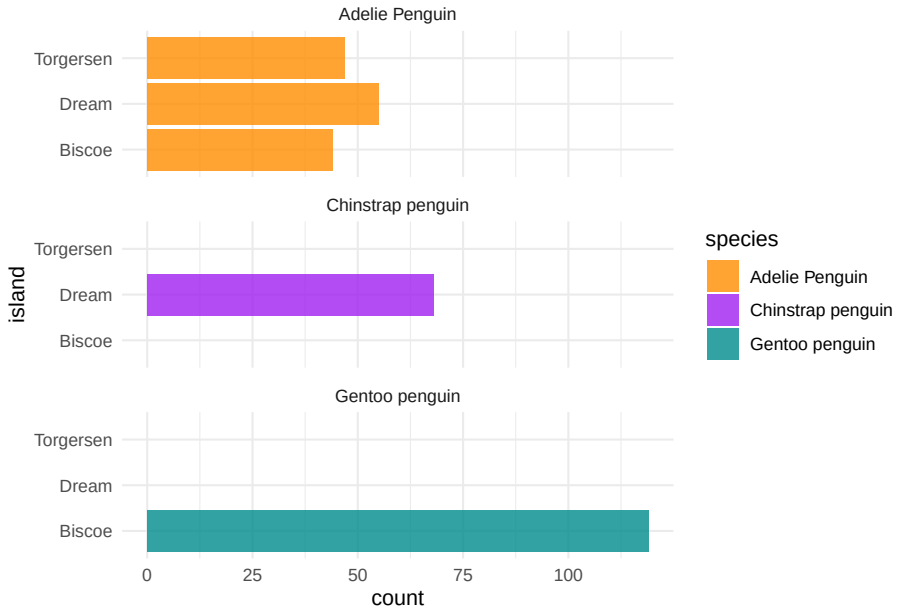
```
penguins %>%
  ggplot(aes(x = body_mass, fill = species)) +
  geom_density(alpha = .5) +
  theme_minimal() +
  labs(x = 'Body mass (g)', y = 'Density',
       fill = 'Species',
       title = 'Body mass distributions across Species') +
  scale_color_brewer(palette = "Dark2") +
  facet_wrap(~sex, scales = 'fixed')
```



4.5.6 Coord flip

For some plots, it might also be interesting (either more aesthetically pleasing or just a necessity given space limitations) to "flip the coordinates". You can do so, by using the command `coord_flip()`. Here's an example.

```
ggplot(penguins, aes(x = island, fill = species)) +
  geom_bar(alpha = 0.8) +
  scale_fill_manual(values = c("darkorange","purple","cyan4")) +
  theme_minimal() +
  facet_wrap(~species, ncol = 1) +
  coord_flip()
```



4.5.7 Correlation plots

Another particularly useful feature that is usually (initially) explored with plots is correlations. So lets say we want to get a general idea about correlations between a set of variables in the dataframe. Here's a way to do it:

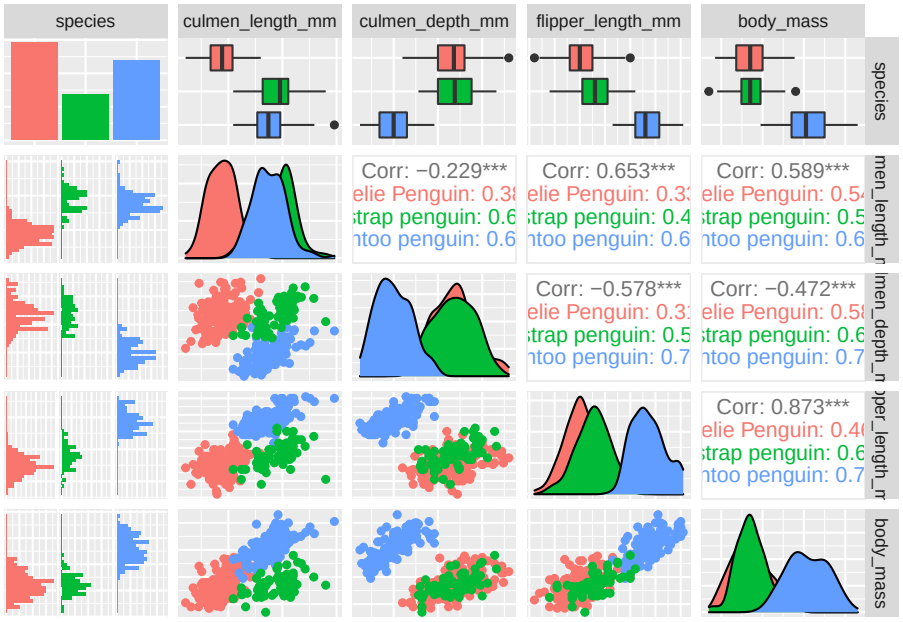
```
correlations <- penguins %>%
  select('culmen_length_mm', 'culmen_depth_mm', 'flipper_length_mm', 'body_mass') %>%
  cor()

corrplot::corrplot(correlations, method = 'number')
```

We can additionally separate this correlations by specie. For that we can use the function `ggpairs()` from the `GGally` package.

```
penguins %>%
  select('species','culmen_length_mm', 'culmen_depth_mm', 'flipper_length_mm', 'body_ma
  GGally::ggpairs(aes(color = species), axisLabels = 'none')
```

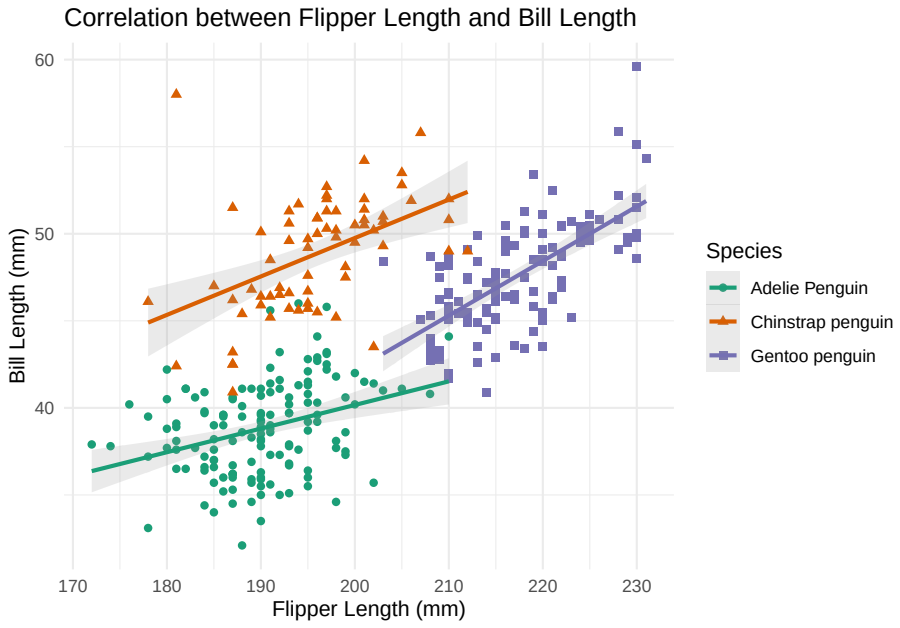
```
## Registered S3 method overwritten by 'GGally':
##   method from
##   +.gg   ggplot2
```

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



When you want to assess something more specific you can use a point plot. Point plots allow us to see how two numeric variables correlate with each other. In this example we explore how bill length and flipper length might be correlated, within each species.

```
penguins %>%
  ggplot(aes(x = flipper_length_mm, y = culmen_length_mm, color = species, shape = species)) +
  geom_point(aes(group = species), size = 1.7) +
  geom_smooth(aes(group = species), method = 'lm', alpha = .2, formula = 'y ~ x') + # adding a li
  theme_minimal() +
  labs(x = 'Flipper Length (mm)', y = 'Bill Length (mm)',
       color = 'Species', shape = 'Species',
       title = 'Correlation between Flipper Length and Bill Length') +
  scale_color_brewer(palette = "Dark2")
```



4.5.8 Multiplots

Another cool thing we might want to do is add multiple plots to a final image. We can do this using the function `multiplot` from the packages `Rmisc`. Lets say we want to measure the mean of the deltas ("both delta_15_n_ooo" and "delta_13_c_ooo"), by species and sex. And we want to see them, side by side. To do so, we must create each plot separately, and assign it to a object. Afterwards, we just need to call these objects inside the `multiplot` function. Here's how to do it:

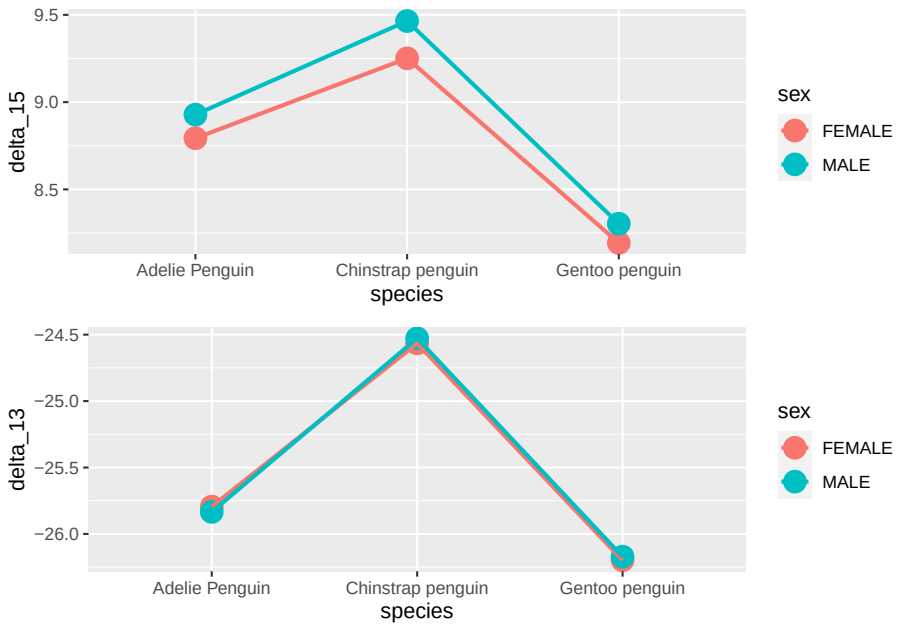
```
d15_plot <- penguins %>%
  filter(!is.na(delta_15_n_ooo)) %>%
  group_by(species, sex) %>%
  summarise(delta_15 = mean(delta_15_n_ooo)) %>%
  ggplot(aes(species, delta_15, color = sex)) +
  geom_point(size = 5) +
  geom_line(aes(group = sex), size = 1)
```

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

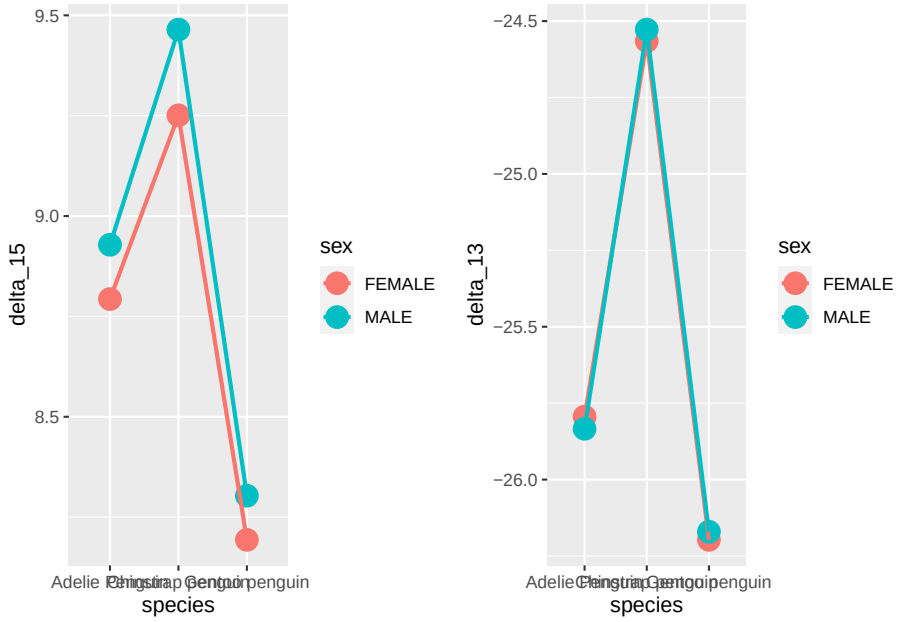
```
d13_plot <- penguins %>%
  filter(!is.na(delta_13_c_ooo)) %>%
  group_by(species, sex) %>%
  summarise(delta_13 = mean(delta_13_c_ooo)) %>%
  ggplot(aes(species, delta_13, color = sex)) +
  geom_point(size = 5) +
  geom_line(aes(group = sex), size = 1)
```

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

```
Rmisc::multiplot(d15_plot, d13_plot)
```



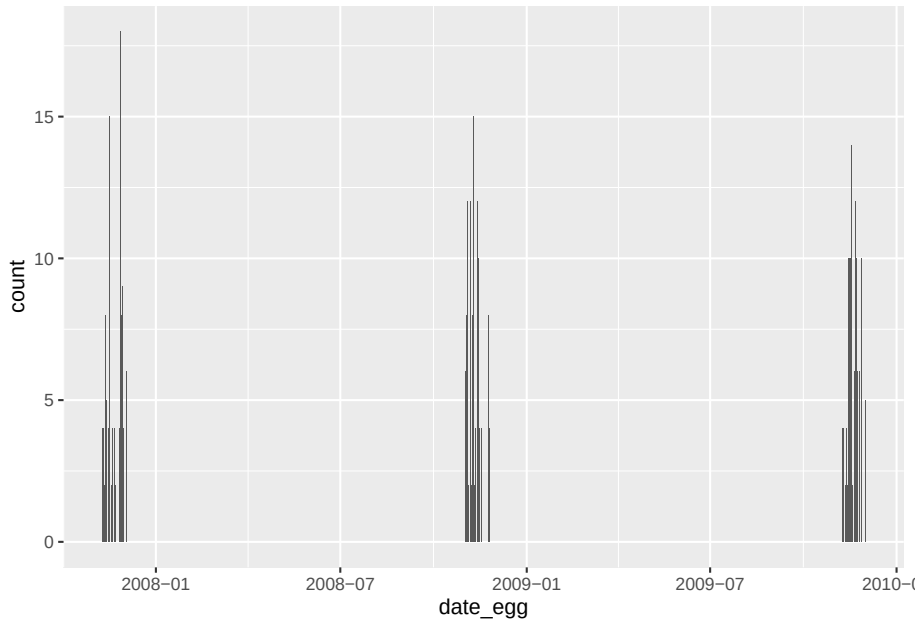

```
# or
Rmisc::multiplot(d15_plot, d13_plot, cols = 2)  # the cols parameter establishes the number of co
```



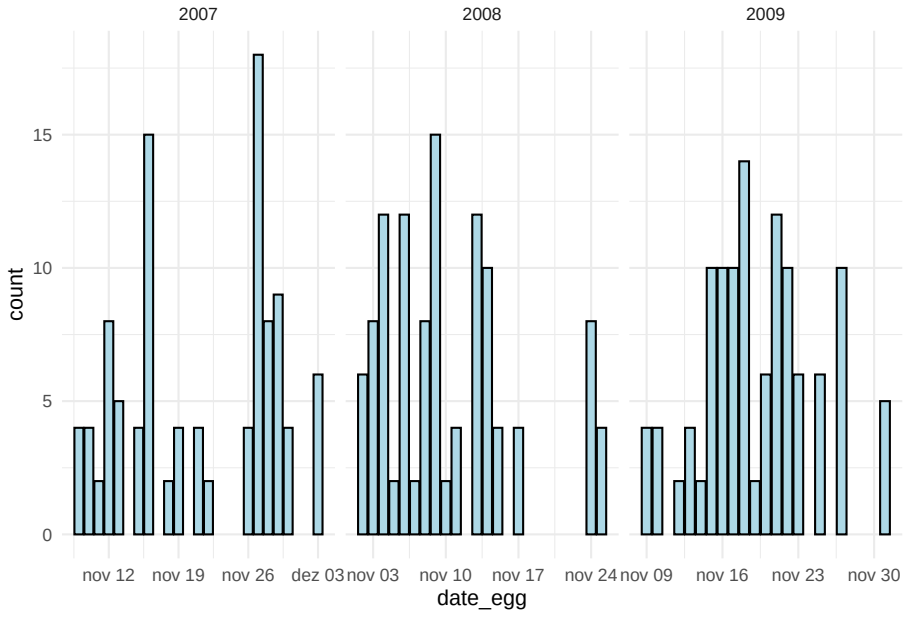
4.5.9 Date

Plotting date is quite easy, and its quite useful when dealing with time-series analysis. In this data, as you will see, data is not that relevant. Below, I first build a general plot showing how many collections there were per date. Since I found a pattern, showing that these collection took place between November and December in three separate years, I then build three blocks of data for each year, and split the plot to better show the number of data entries per date.

```
# Original plot
penguins %>%
  ggplot(aes(date_egg)) +
  geom_bar()
```



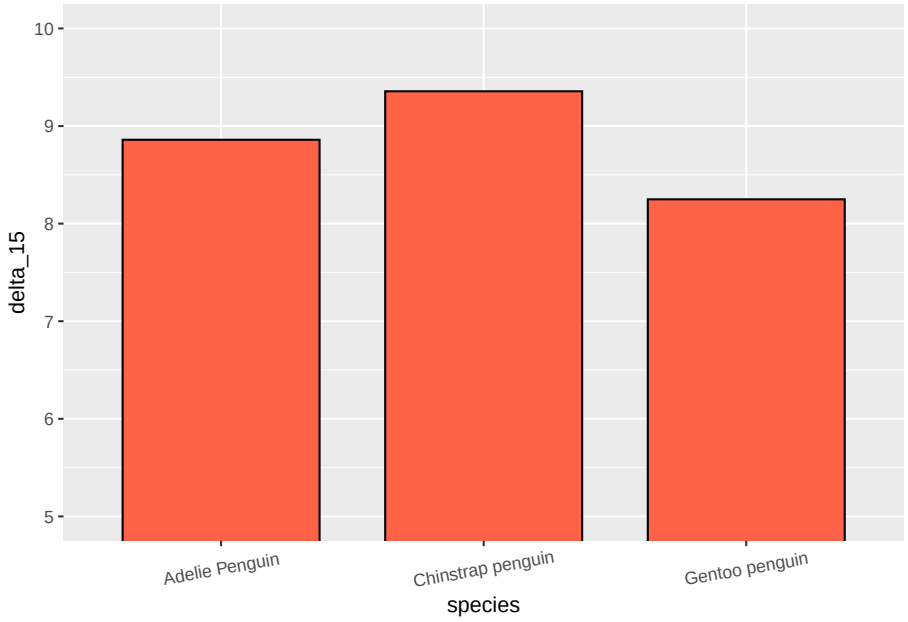

```
# Transformed plot
penguins %>%
  mutate(DateBlock = case_when(  # creating a new column that identifies which year that data bel
    date_egg < '2008-01-01' ~ '2007',
    date_egg > '2008-01-01' & date_egg < '2009-01-01' ~ '2008',
    date_egg > '2009-01-01'  ~ '2009'
  )) %>%
  ggplot(aes(date_egg)) +
  geom_bar(fill = 'lightblue', color = 'black') +
  facet_wrap(~DateBlock, scales = 'free_x') +
  theme_minimal()
```



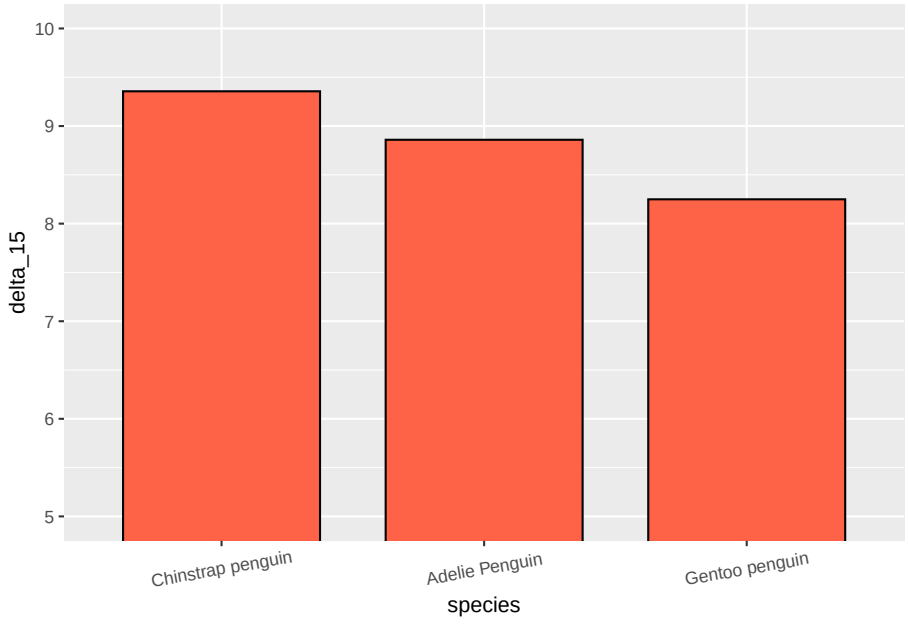
4.5.10 Ordering factors

Another useful function, is the function `fct_reorder()` which comes from the package `forcats`. This function is particularly (although not exclusively) useful for plots. As default, R organizes the levels in a factor by alphabetical order. This function allows us to alter the order by which levels in a factor are presented, according to certain conditions we can define ourselves. For instance if we plot the mean "delta_15_ooo" per species of penguin, we get the x-axis with (alphabettically defined) Adelie, Chinstrap and Gentoo penguins. But lets say instead, we want to reorder the x-axis in a descending order according to mean delta_15 of each species.

```
# Original plot
penguins %>%
  mutate(species = as.factor(species)) %>%  # Turning species to factor
  group_by(species) %>%
  summarise(delta_15 = mean(delta_15_n_ooo, na.rm = TRUE)) %>%
  ggplot(aes(species, delta_15)) +
  stat_summary(fun = mean, geom = 'bar', width=.75, fill = 'tomato1', color = 'black')
  theme(axis.text.x = element_text(angle = 10, vjust = 0.6)) +# just to show the x-axi
  coord_cartesian(ylim = c(5, 10))
```




```
# Reordered plot
penguins %>%
  group_by(species) %>%
  summarise(delta_15 = mean(delta_15_n_ooo, na.rm = TRUE)) %>%
  mutate(species = fct_reorder(species, delta_15, .desc = TRUE)) %>%  # organized the levels of s
  ggplot(aes(species, delta_15)) +
  stat_summary(fun = mean, geom = 'bar', width=.75, fill = 'tomato1', color = 'black') +  # plott
  theme(axis.text.x = element_text(angle = 10, vjust = 0.6)) + # just to show the x-axis text bet
  coord_cartesian(ylim = c(5, 10))
```


10
9
8
7
6
5
delta_15
Chinstrap penguin
Adelie Penguin
Gentoo penguin
species

4.6 End