Fundamentals of R for BIologists

Dillon A. Jones

LEARN
ADVENTUROUSLY
LearnAdventurously.com

Fundamentals of R for Biologists

Dillon Jones

2022-07-25

Contents

Preface

This textbook was written to help those in the biological sciences learn the R programming language. This body of work was written out of frustration of my own R coding journey. While I was learning the R language many years ago, I kept running into tutorials that assumed a programming background, used data sets that had no direct relevance to my work, and often included so much material that it was overwhelming to parse through.

This textbook attempts to remedy those issues by assuming no coding background of its reader, using example biological datasets, and by only focusing on the most important functions needed to be successful in R.

While this work is catered for those in the early to intermediate biology careers, the content contained within is applicable for students of any background.

This textbook was developed in-line with an online course of the same name found at LearnAdventurously.com/courses. While this textbook is free to download, the course offers over 70 video lessons, simple quizzes to cement your knowledge, and "Mastery Checks" which are mini-projects found at the end of every major section. This course is entirely online and self-paced with a certificate awarded to those who complete the course.

In the interest of keeping this preface short, I want to leave you with one last note. The world we live in is a vast landscape filled with limitless information. So much information that we humans could never hope to collect, categorize, or comprehend it all. We live in a time with no single solution for our problems. Instead there are dozens, hundreds, or even thousands of equally valid answers for the same question.

As you learn the R language, there will be times where you feel stuck. Where you are unsure of what to do next. Where you may think there is no solution to your unique problem. The beauty of coding is that we can create new solutions. Throughout this text, I am providing you not with old solutions to old problems. Rather, I hope to give you the tools so that you can create your own solutions.

I hope the information contained within this text will help you with whatever questions you may seek to answer in this world.

Happy coding!

-D

Overview of material.

As mentioned in the preface, this text assumes no background knowledge of coding in any language. However, I recognize that many people may have already downloaded R + Rstudio, assigned a few objects, and perhaps even analyzed some data in R.

Section 0 - Introduction to R and Rstudio is meant for those who may have never written a single line of code, are incredibly lost anytime they read documentation for R, or may have never even heard of R and Rstudio before. I would recommend all students to at least glance at Section 0, just in case there is material that is beneficial to you. However, it is also a section that is meant to be skipped for those that may already have some knowledge of R.

Section 1 - Basics of R covers some of the basic elements of R. This section introduces the different classes and types of objects we will use in R. Then, we learn the fundamentals of loading and saving data into and out of R. Finally, we introduce the basics of plotting and visualizing your data.

Section 2 - Manipulating Objects and Logic largely covers subsetting and logic. Here students will learn how to extract specific elements from all types of objects in R. Then they will learn how to combine different datasets using the rbind(), cbind() and merge() functions. This section is incredibly important to grasp as it lays down the foundation for many topics covered later in the text.

Section 3 - Introduction to Tidyverse introduces the tidyverse package. In short, tidyverse is a collection of functions specifically catered to data wrangling, manipulation, and visualization. This section will start by detailing pipes and pipelines. Then we introduce a variety of tidyverse functions that allow us to extract data (select() and pull()), manipulate how our data appears (arrange() and relocate()), and how to filter our data based on some criteria (filter() and distinct()). Finally, we then cover how to do simple analyses using the mutate(), count(), summarize(), and group_by() functions.

Section 4 - Cleaning data and creating pipelines is specifically aimed at cleaning datasets. Here we introduce best practices for data cleaning and uses a variety of functions learned in Section 3 to fix problematic data. I also detail how to use functions such as unique(), is.na(), and glimpse() to uncover potential problems in our data. This section is likely the most important section for those in the biological sciences to learn, as cleaning our data often takes a considerable amount of time.

Section 5 - logic, custom functions, and apply() takes a step back and introduces a variety of operations that are common place in nearly all programming languages. Students will learn how to fork their code with if-else statements, run functions over entire data sets use for-loops, and teaches students how to create their own custom functions. In my opinion, learning and understanding this section is the absolute best thing you can do to increase your coding ability. This section provides you with the tools to solve almost any problem you may run into.

Section 6 - Plotting with ggplot2 covers how to visualize data with the ggplot2 package. ggplot2 is the most widely used data visualization package and is often a source of frustration. This section will break down ggplot into its fundamental components. We discuss how to create basic visualizations using all of the parameters available in ggplot2.

Finally, there are a variety of "bonus" lessons scattered throughout this textbook. As mentioned in the preface, this textbook has an accompanying online course found at LearnAdventurously.com/courses. The bonus lessons are completed lessons that for one reason or another were cut from the final course. This could be due to the complexity of the lesson, how it would fit into the overall course, or any number of reasons. Rather than removing this already completed material, I kept them in the text as "bonus" lessons. These are still in the online course as well, however they have no accompanying video lecture. It is very likely, that these lessons will be used in future courses and texts that cover a particular topic in more detail.

As you go through this text it is highly recommended that you follow along in your own Rstudio environment. I encourage you to actually write out the code and try to understand how it all works. If you would like to download each section as an R Markdown document, which allows you to run the code embedded throughout this text, that is all available in the online course.

I hope this adequately covers all of the material that you are about to learn! When you’re ready, hop into any of the sections and get started on your R journey!

Happy coding!

Section 0 - Introduction to R and Rstudio

0.0 Section 0 Overview

Welcome to the course!

This section is meant for those who have absolutely no prior experience with R, need a refresher on how to install R/Rstudio, or are struggling with some of the core concepts. The topics covered this week include:

- **What is R and Rstudio?**
- **How to install R and Rstudio**
- **The layout of Rstudio**
- **What is a function?**
- **What is a package?**
- **How to add comments to your scripts**

This section is considered optional dependent on your current level of R. If you need to come back to this section, it will always be available!

0.1.1 What is R and Rstudio?

What is R and Rstudio?

Lets start with a very basic overview of R and Rstudio.

What is R? "R is a language and environment for statistical computing and graphics." https://www.r-project.org/about.html

In short, **R is a language that allows you to manipulate, analyze, and visualize data**. A big benefit of R, is that it is free and open source, meaning anyone can develop their own packages that include custom functions. We'll touch more on R functions and packages shortly, but first lets introduce Rstudio.

What is Rstudio? Rstudio is an Integrated Development Environment (IDE) specifically catered for R. In essence, **Rstudio is a software that is tailor-made to run the R language** with several major quality of life features that make coding in R much easier.

Both R and Rstudio are 100% free for you to download and use on your own device.

Installing R To install R you will need to download and install the respective software from https://cran.r-project.org/

At the top of that page, there are download links for Windows, Mac, and Linux. Install the latest R version and you can then begin to run R on your computer!

Installing Rstudio After you have installed R, you can install Rstudio on your device by downloading it from https://www.rstudio.com/products/rstudio/download/#download and following the installation wizard.

An R alternative in your Browser Alternatively, you can use RStudio in your browser without needing to download anything by navigating to rstudio.cloud. While there are minor differences between RStudio and rstudio.cloud, I have made sure the course can operate through either platform.

Take note, that rstudio.cloud does have a limit on how much you can use the free version. If you are just wanting to get started right now, this is your best bet!

Use what you are most comfortable with!

0.1.2 The layout of Rstudio

How do you use RStudio?

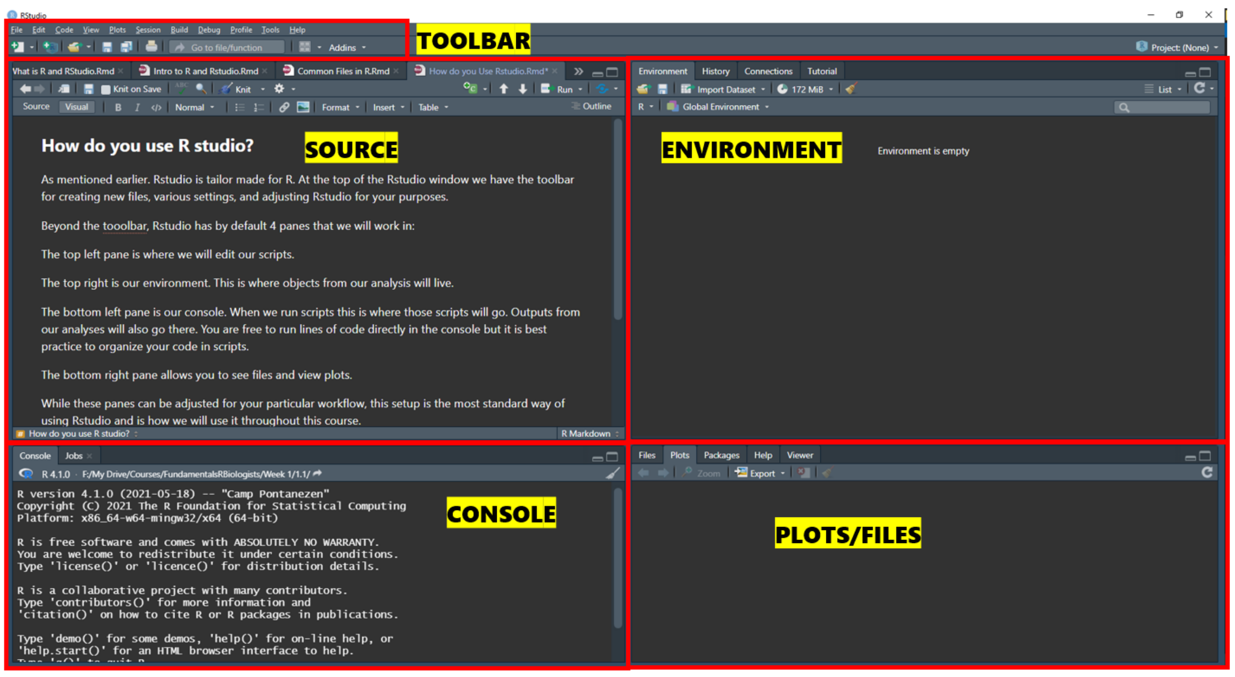
As mentioned earlier, RStudio is tailor made for R. **At the top of the Rstudio window we have the toolbar with various buttons** for creating new files, configuring settings, and adjusting Rstudio for your purposes.

Beyond the toolbar, Rstudio has by default 4 panes that we will work in:

- The **top left pane** is our source. Here where we will edit our scripts and Rmarkdown files. We can also view full data sets here.
- The **top right pane** is our environment. This is where objects from our analysis will live.
- The **bottom left pane** is our console. When we run scripts this is where those scripts will go. Outputs from our analyses will also go there. You are free to run lines of code directly in the console as well.
- The **bottom right pane** allows you to see files and view plots.

While these panes can be adjusted for your particular workflow, this setup is the most standard way of using Rstudio and is how we will use it throughout this course.

You may also notice that my Rstudio has a different color scheme than yours. That can be changed by going to **Tools> Global Options> Appearance** and changing the editor theme. I use Idle fingers as I have a good deal of light sensitivity that makes the default theme difficult to look at!



How do you use R studio?

As mentioned earlier. Rstudio is tailor made for R. At the top of the Rstudio window we have the toolbar for creating new files, various settings, and adjusting Rstudio for your purposes.

Beyond the tooolbar, Rstudio has by default 4 panes that we will work in:

The top left pane is where we will edit our scripts.

The top right is our environment. This is where objects from our analysis will live.

The bottom left pane is our console. When we run scripts this is where those scripts will go. Outputs from our analyses will also go there. You are free to run lines of code directly in the console but it is best practice to organize your code in scripts.

The bottom right pane allows you to see files and view plots.

While these panes can be adjusted for your particular workflow, this setup is the most standard way of using Rstudio and is how we will use it throughout this course.

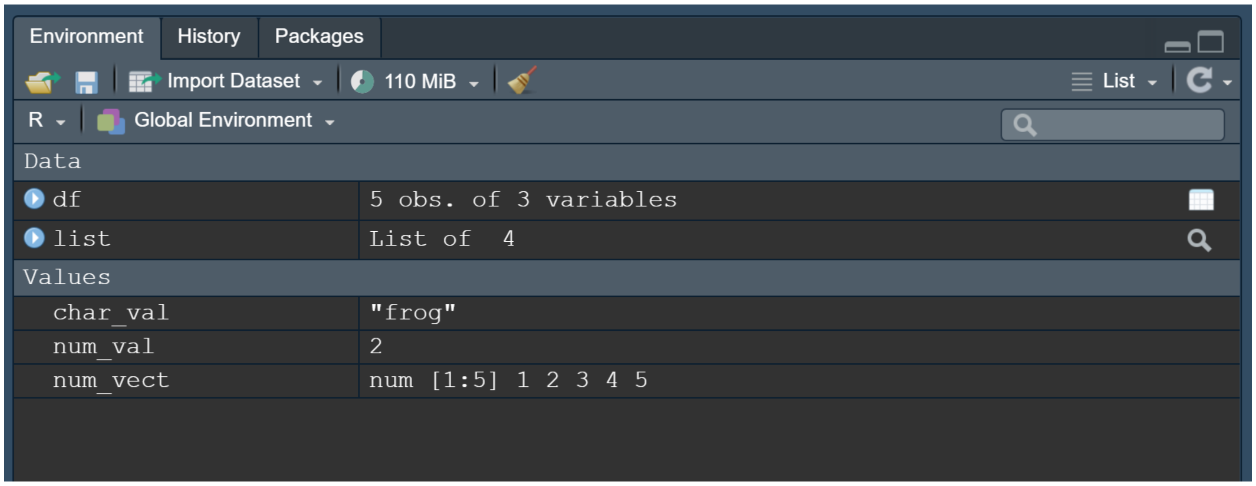
```
R version 4.1.0 (2021-05-18) -- "Camp Pontanezen"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
```

0.1.3 The Environment

The Environment pane

The environment pane lives on the upper right hand side of Rstudio. The environment allows us to see what objects are saved in R as well as quickly preview their contents. Running the chunk below will add a series of objects to the environment that you can then explore.

```r
num_val <- 2 #A single numeric value
char_val <- "frog" # A single character value

num_vect <- c(1,2,3,4,5) # a vector of numbers

df <- data.frame(col_1 = c(1,2,3,4,5), #A dataframe of 3 columns and 5 rows
                 col_2 = c('a','b','c','d','e'),
                 col_3 = c(22.4,23.7,34.2,12.3,45.3))

list <- list(num_val,char_val,num_vect,df) #a list that contains all the previous objects
```



```
Data
df          5 obs. of 3 variables
list        List of  4
Values
char_val    "frog"
num_val     2
num_vect    num [1:5] 1 2 3 4 5
```

For values and vectors, a preview of the data is simply displayed beside their object name.

For more complex objects, such as dataframes and lists, a preview can be displayed by selecting the circular button with a right facing triangle to the left of the object.

A complete view of the data can be seen be selecting either the table (in the case of data frames and matrices) or the magnifying glass icon (in the case of lists) to the right of the object.

0.2.1 Common Files in R

Common files in R

This section will briefly cover some of the most common file types exclusive to R

R scripts (.R files) In most uses of R, we create scripts. **Scripts are a collection of functions written in the R language that can be interpreted by the R software.** We call the process of creating these scripts, coding.

Scripts can achieve any number of goals such as manipulating a data set, conducting a statistical analysis, or visualizing the results of your research. **Scripts are kept in R as a .R file.**

R Markdown (.Rmd files) RMarkdown objects can be thought of as an extension of scripts that allow you to run segments of code (called chunks) alongside text. RMarkdown files can also be exported into PDF, html, or even word documents. Once exported, they retain all of the text and code while also displaying the results from the chunks.

All of these lessons are actually created using RMarkdown! If you are reading this on the website, you are reading the exported html file from an RMarkdown object.

Here is an example chunk that simply adds 2+2. Remember, chunks are snippets of code that we can run and achieve a simple result. If we run this chunk, we will see the output of 2+2 below the chunk.

```
2+2
```

```
## [1] 4
```

Of course, future chunks are going to be more complicated than just 2+2. If opening the RMarkdown file directly, you can run each chunk as you wish. Some people prefer to do all of their R coding solely in RMarkdown objects instead of in scripts as they can easily organize their thoughts. How you decide to use R is up to you!

Data files (.RData) As we will learn soon, R allows the creation of many objects. These objects could be data sets, results from analyses, or visualizations related to your study!

While it is entirely possible to save each of these objects individually, and there are certainly cases where that is preferable, we can also save collections of these objects as a single .RData file. This makes file management much easier especially if you are conducting your analysis in several steps, need to collaborate with someone on the project, or just like to keep your data organized!

0.2.2 The Working Directory

Working directory

Before we load data into R, we need to introduce the working directory.

In short, **the working directory is a folder on your computer that R has access to**. This folder is the spot that you designate for R **to load in data, export results, and in general, work out of.**

This working directory can be almost anywhere on your computer.

Lets say you make a new folder on your desktop named "example". What we need to get now is the path to this working directory.

Path to Working Directory The path tells R how to find this folder. If the folder is on your desktop, it is very likely your path is something similar to "C:\Users\Dillon\Desktop\example" if you're on windows

YOUR PATH WILL BE DIFFERENT THAN MINE

Unless the username on your machine is also Dillon, it is likely your path will be different. If you are not tech savvy, it may be confusing to figure our what your path is the first time.

I've included a few resources the detail how to find the path for various devices.

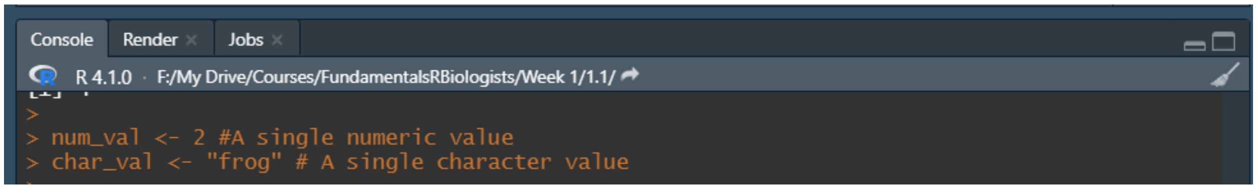
- Windows: https://www.addictivetips.com/windows-tips/get-complete-path-to-a-file-or-folder-on-windows-10/
- Mac: https://www.howtogeek.com/721126/3-ways-to-see-the-current-folder-path-on-mac/
- Linux: https://www.howtouselinux.com/post/linux-command-get-file-path

Set the Working Directory Once you have your path, you will set your working directory via the **setwd()** function

```
#Example for my PC
setwd("C:/users/dillon/desktop/example")
```

Note, that for the **setwd()** function you need to **place the path in quotations** and **ensure you use forward slashes (/)**and not back slashes (\ ).

If the chunk above ran successfully, you should see the working directory at the top of your console in Rstudio.



```
R 4.1.0 · F:/My Drive/Courses/FundamentalsRBiologists/Week 1/1.1/
>
> num_val <- 2 #A single numeric value
> char_val <- "frog" # A single character value
```

Setting a working directory can often be very confusing for first time R users, but it is important to get this concept down. You will use a working directory almost every time you use R.

0.3.1 Intro to Functions

Intro to Functions

Throughout these sections I have provided examples of several functions. Functions are self contained bundles of code that accomplish some task given an input. One example is **sum()**. The **sum()** function takes a vector of numeric values (input) and calculates the sum of all those values (output).

```
sum(c(1,2,34,6,5,4,2,83))
```

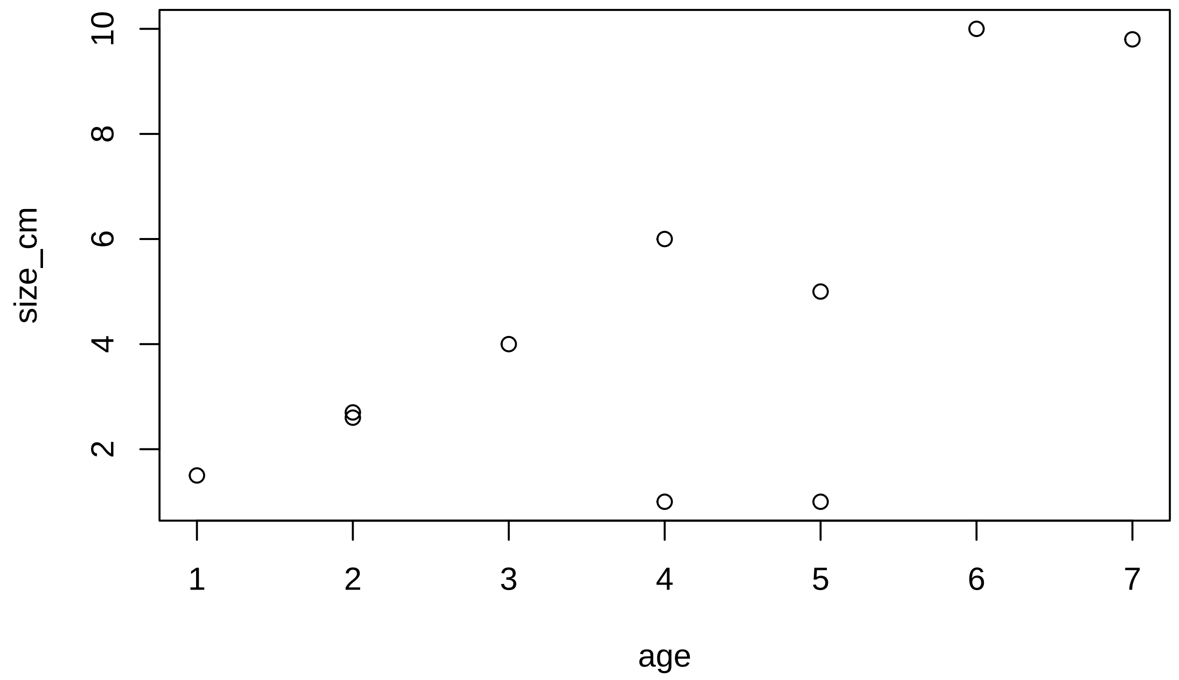
```
## [1] 137
```

There are of course many more functions that accomplish a wide variety of tasks. Any function in R takes in arguments. Arguments are separated by commas and could be an object, a TRUE/FALSE value, or some text that instructs the function in some way.

We can explicitly specify which argument we want to use by using the name of the argument alongside the equal sign.

This is a good time to introduce the plot function!

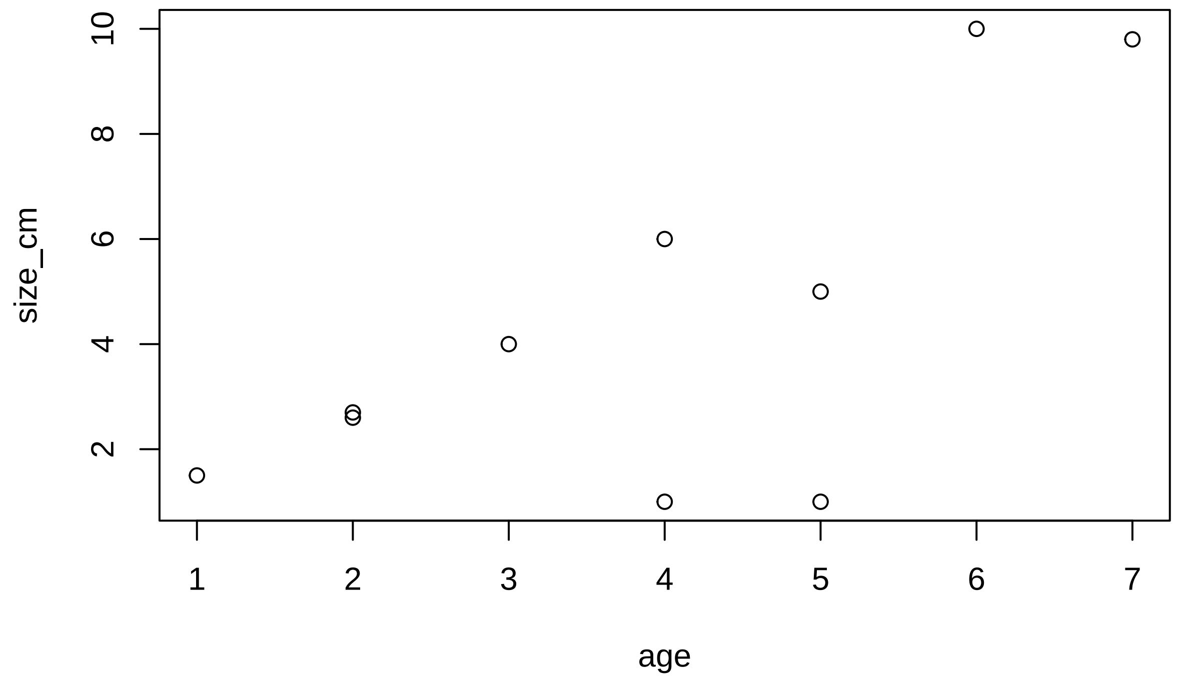
```
#lets first make some equal length vectors of data
age <- c(1,2,5,4,5,4,2,3,7,6)
size_cm <- c(1.5,2.6,1,1,5,6,2.7,4,9.8,10)

#then we can plot that data using the arguments x and y.
# argument x relates to the data you place on the x axis
# argument y relates to the data you place on the y axis

plot(x = age, y = size_cm)
```



Arguments are described by each function in detail and placed in a particular order. For example, with **plot()** x is the first argument, while y is the second.

This order can be shuffled if we explicitly define each argument. For example, this chunk shows how to plot age and size_cm correctly in 3 different ways.

```
plot(age,size_cm)
plot(x = age, y = size_cm)
```

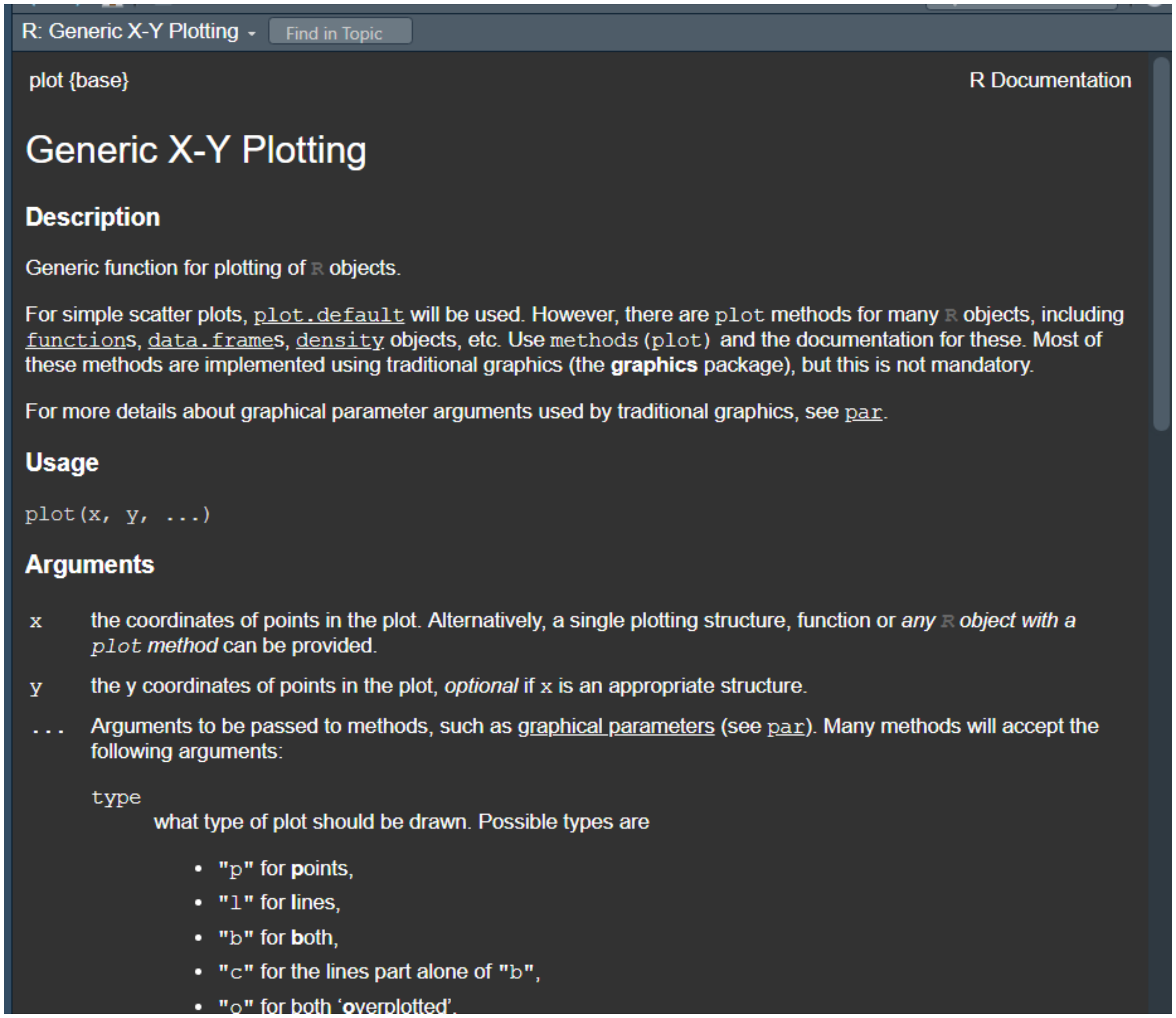


```
plot(y = size_cm,x = age)
```

Naturally, functions can take many different arguments. To understand more about a function and what arguments it takes, we can place a question mark in front of the function to access its help documentation.

An example of the help window is shown below. If you run this on your own device, it will appear in the bottom right pane.

```
?plot
```



plot {base}   R Documentation

Generic X-Y Plotting

Description

Generic function for plotting of R objects.

For simple scatter plots, `plot.default` will be used. However, there are `plot` methods for many R objects, including `functions`, `data.frames`, `density` objects, etc. Use `methods(plot)` and the documentation for these. Most of these methods are implemented using traditional graphics (the **graphics** package), but this is not mandatory.

For more details about graphical parameter arguments used by traditional graphics, see `par`.

Usage

```
plot(x, y, ...)
```

Arguments

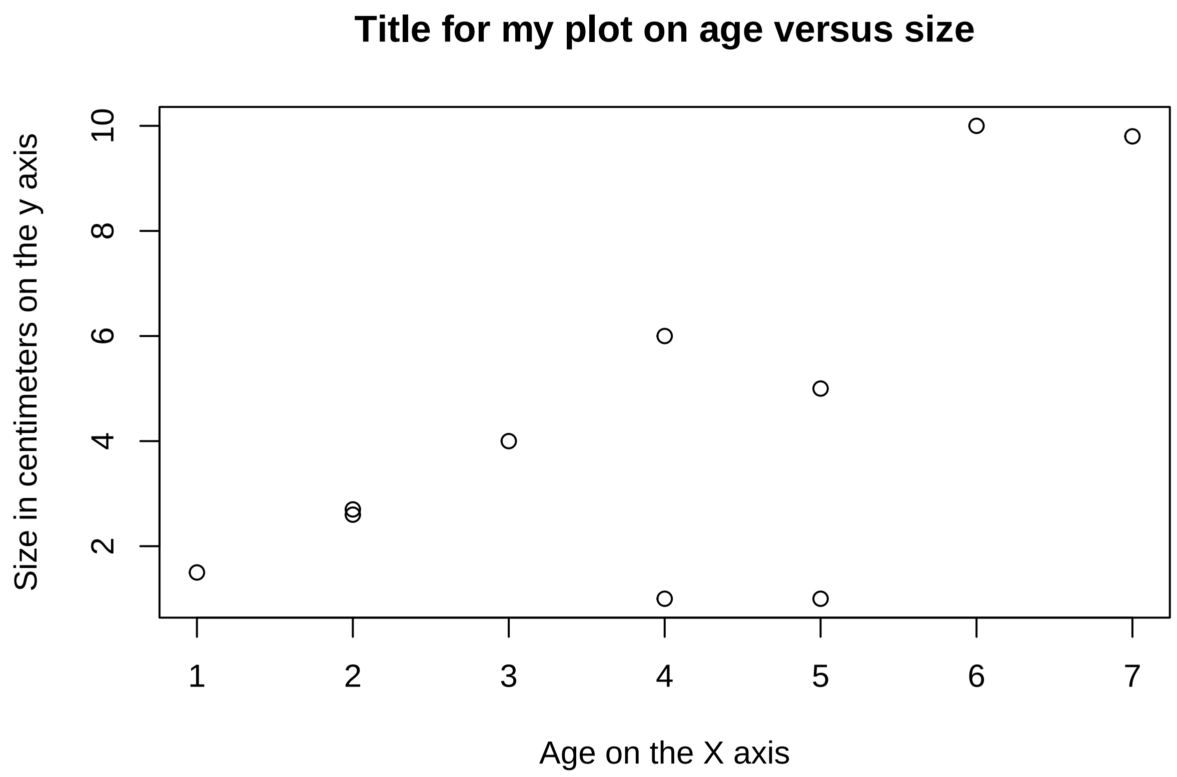
`x`	the coordinates of points in the plot. Alternatively, a single plotting structure, function or *any R object with a `plot` method* can be provided.
`y`	the y coordinates of points in the plot, *optional* if `x` is an appropriate structure.
`...`	Arguments to be passed to methods, such as graphical parameters (see `par`). Many methods will accept the following arguments:

`type`
what type of plot should be drawn. Possible types are

- `"p"` for **p**oints,
- `"l"` for **l**ines,
- `"b"` for **b**oth,
- `"c"` for the lines part alone of `"b"`,
- `"o"` for both 'overplotted',

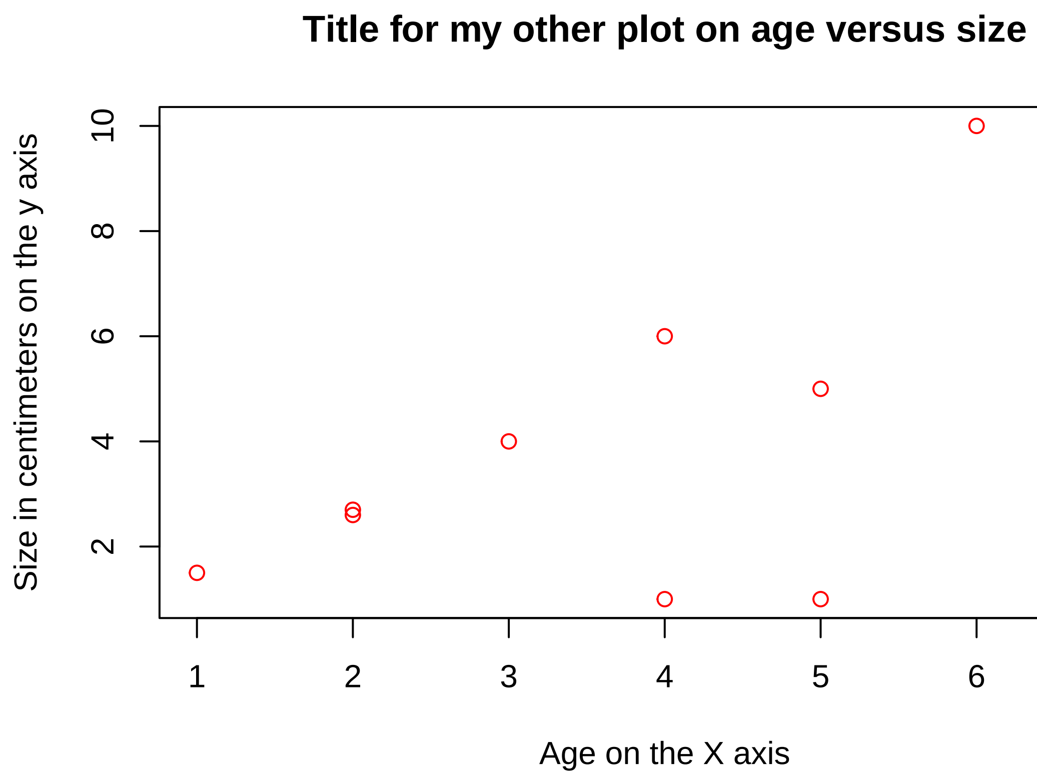
We can also make our functions easier to read by splitting them between multiple rows. To do so, you need to ensure that the function parentheses still contain all the arguments, and that each line is separated by a comma

```
#then we can plot that data using the arguments x and y.
# x relates to the data you place on the x axis
# y relates to the data you place on the y axis
# xlab adds a label on the x axis
# ylab adds a label on the y axis
# main adds a title to the plot

plot(x = age, y = size_cm,
     xlab = "Age on the X axis",
     ylab = "Size in centimeters on the y axis",
     main = "Title for my plot on age versus size")
```



Again, if you are explicitly defining the arguments in your function, the order they appear does not matter.

```
#The following code will produce the same plot, even if the order of these arguments doesnt really make

plot(xlab = "Age on the X axis",
     ylab = "Size in centimeters on the y axis",
     x = age,  main = "Title for my other plot on age versus size",
     y = size_cm, col = "red")
```



Functions are what we will mostly be working with in R. The beauty of R being an open source software, is that anyone can write their own functions and distribute them to other users through packages. Including you!

0.3.2 Intro to Packages

Intro to Packages

Packages are collections of functions written by other users. These packages are often catered to solving a particular issue or are meant for a particular analysis.

For example, there are packages for manipulating spatial data, analyzing phylogenetic history, and interfacing with organizations such as the IUCN to name a few.

For any package, we must first **install it into R** and then we need to **load it with library().**

You can think of installing the package like buying a book and putting it on your book shelf.

Loading the package is similar to pulling the book off the shelf so you can access the information contained within.

To install a package, we just need to use the **install.packages()** function

```
 #This is where we get the "book" and put it on our bookshelf.

install.packages("tidyverse")
```

Then to load the package we just use the **library()** function

```r
#this is where we get the information from the "book" to use in R
library(tidyverse)
```

Once a package is installed on your device, you typically will not need to install it again and can just access it with the **library()** function.

Naturally, it would be impossible to teach a course using all of the different packages available. However, there is one package that in my opinion should be used in nearly every single R project.

That package is tidyverse.

Technically, tidyverse is a package which contains many carefully curated packages. Tidyverse allows all of these packages to communicate with one another seamlessly. You can see all the included packages at tidyverse.org, but here are a few and their uses.

- dplyr - data manipulation
- ggplot2 - visualizations
- stringr - manipulating strings and characters

In Section 3, we introduce tidyverse in detail.

0.3.3 Comments

Commenting

When we code, it is good practice to leave comments for ourselves. Comments are parts of our script that do not run. In R, any text following the # symbol is treated as a comment.

Comments can be used to leave notes for yourself, add organization to your code, and document how your particular analysis is meant to run. Pay attention to comments through this course as they often have instructions or extra information!

Take this chunk for example. Even though there is plenty of text in the comments, only the output 2+2 will run.

```r
#This is a comment and will not affect the code when I run it.
#We need to make sure each line that we want commented is precededed by a # symbol

#3+3

#We can see above that the command 3+3 will not run unless we remove the # symbol

2 + 2
```

```
## [1] 4
```

Using comments often is key to keeping your code neat and organized. Imagine coming back to a particular script 3 months later and having no idea why you did what you did!

In short, use comments and use them often!

Section 1 - Basics of R

1.0 Section 1 Overview

Section 1 Overview

Welcome to Section 1 of Fundamentals of R for biologists!

This section covers introductory material for this course. While not as fundamental as Section 0, the material covered is still considered basics of R and Rstudio.

We will cover:

- Common types and classes of R objects
- How to load, save, and organize data in R
- A brief intro into the plot() command.

1.1.1 Math Rules

Math Rules

One of R's most fundamental uses is as a calculator. R can run any number of mathematical functions. The following chunk shows the basics.

Simple operations Simple mathematical operations can be performed easily

```
2+2 # Addition via the + symbol
4-2 # Subtraction via the - symbol
2*2 # Multiplication via the * symbol
2/2 # Division via the / symbol
2^2 # Exponents via the ^ symbol
```

```
## [1] 4
## [1] 2
## [1] 4
## [1] 1
## [1] 4
```

R follows PEMDAS rules as well.

```
2*3+100
100 + 2*3
(100 +2)*3
```

```
## [1] 106
## [1] 106
## [1] 306
```

Simple functions R also contains a wide variety of functions to perform different mathematical operations.

```
#Heres a few examples

sum(2,2,3,5,6,8) #Sum of all these numbers

mean(2,2,3,5,6,8) #The average of all these numbers

median(2,2,3,5,6,8) #The median of these numbers

sqrt(81) #get the square root of a number

sd(c(2,3,4,5,6,7,8,2,10,10)) #The standard deviation of a a set of numbers
```

```
## [1] 26
## [1] 2
## [1] 2
## [1] 9
## [1] 3.020302
```

We can also round values using **ceiling()** (to round up) and **floor()** (to round down). The **round()** function exists as well and follows standard rounding rules.

```
x <- 3.2

ceiling(x)
floor(x)
round(x)
```

```
## [1] 4
## [1] 3
## [1] 3
```

1.1.2 Intro to Objects

Intro to Objects

Lets introduce a concept that is fundamental to R. Objects and object assignment.

An object is anything in R that holds information for R to use. We assign information to an object using the arrow **<-** or the single equal sign =

```
x <- 2 # a simple object, x, containing the data 2

x #if we type x, then the information within x will be printed
```

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

When we assign data to an object in R, we use the Arrow symbol <-. Think of it as the information you are wanting to retain is flowing into the object via the arrow sign.

Objects can be named any combination of characters and numerics

What makes an object useful, is that we can apply functions to the data contained within.

```r
#Watch that in all of the below examples, x is equal to 10

x<- 10 #x is equal to 10

x+2

x*6

x/2
```

```
## [1] 12
## [1] 60
## [1] 5
```

Objects can contain more than a single number of course. The object below contains many numbers and we can run functions that apply to all of the values contained within the object.

```r
x <- c(2,4,5,6,7,8,2)  #Now x contains all the numbers 2,4,5,6,7,8,2

x+10 #We can add 10 to every number

x*2 #we can multiply all the numbers by 2

sum(x) #we can get the sum of all those values

mean(x) #we can get the mean of all these values
```

```
## [1] 12 14 15 16 17 18 12
## [1]  4  8 10 12 14 16  4
## [1] 34
## [1] 4.857143
```

Objects are a core component of R programming. But objects don't just need to be simple numbers. Lets explore the different classes of objects we can use in R.

1.1.3 Classes of Objects

Intro to Object Classes

All data and objects in R fall into a **class.** Classes describe the object and tell R how to interpret the information.

Most often, you will work with objects from the classes: **character, numeric, and logical**

However, be aware that many custom classes exist for different types of data!

You can find the class of on object by using the function class()

```
class(2) #numeric
class('Urban') #character
class(1==2) #logical
```

```
## [1] "numeric"
## [1] "character"
## [1] "logical"
```

In the below example, we assign numeric data to the variable x. This gives x the class numeric.

```
x <- 2+4

class(x)
```

```
## [1] "numeric"
```

Character Character objects are composed of letters, numbers, and symbols, often referred to as strings. Common strings include dates, names, and locations. You cannot perform mathematical operations over a character object. We can specify an object as a character by using quotation marks (double or single!).

```
x <- "string double quotes"
x

class(x)

x <- 'string single quotes'
x

class(x)
```

```
## [1] "string double quotes"
## [1] "character"
## [1] "string single quotes"
## [1] "character"
```

The next example shows what would happen if you tried to add the number 3 to a character object. Even though the character is the number 3, we cannot apply mathematical operations over it because it is in the character class. To r, this would be like trying to add 2 to the word apple.

```
x <- "3"
x
class(x)

x + 2
```

```
## Error in x + 2: non-numeric argument to binary operator
```

```
## [1] "3"
## [1] "character"
```

Numeric Numeric objects are simply numbers. A numeric refers to any number whether they be whole numbers (integers) or if they contain decimals (doubles). Numeric objects can have mathematical operations performed on them.

```
 x <- 2
 class(x)

 x <- 2.5
 class(x)

 x + 2

 class(x+2)
```

```
## [1] "numeric"
## [1] "numeric"
## [1] 4.5
## [1] "numeric"
```

Logical Logical objects need to be either TRUE or FALSE values. These are often used for presence/absence, TRUE/FALSE, or any other type of binary data. Additionally, logical objects can be used to compare one value to another value. These are really useful while cleaning your data.

```
class(TRUE)
class(FALSE)
```

```
## [1] "logical"
## [1] "logical"
```

We can create logic by comparing 2 values with the double equal sign

```
1==1
1==2
```

```
## [1] TRUE
## [1] FALSE
```

Doing so results in a logical object is we save the output

```
x <- 1==2

class(x)
x
```

```
## [1] "logical"
## [1] FALSE
```

There are many more data classes that you can work with, but this should give us a good start. Lets talk about how we can organize data and objects into different types.

1.1.4 Types of Objects

Types of Objects

Objects can contain data in a variety different of formats. In this course, we will focus on **Values, Vectors, Matrices, Data frames, and Lists.** Each is explained below, alongside what classes of objects they can be composed of.

Values Values, represent single entries of data. They can be of any class.

```
x <- 2
x <- 'froggy'
```

Vectors Vectors, represent 1 dimensional data (a single row) with several values. We often create vectors using the function **c()**.

```
x <- c(2,3,4,5,6)
x

y <- c('red','blue','green','yellow')
y
```

```
## [1] 2 3 4 5 6
## [1] "red"    "blue"   "green"  "yellow"
```

Values can be of any class, however, be aware that mixing classes will result in the character class for all the values.

```
z <- c(2,3,4,'Apple','Orange','Frogpear')
z
class(z)
```

```
## [1] "2"        "3"        "4"        "Apple"    "Orange"   "Frogpear"
## [1] "character"
```

Matrix A Matrix, represents 2 dimensional data (columns and rows) where the classes of data (numeric, logical, character etc.) are all the same

```
mat <- matrix(c(c(1,2,3,4,5),
                c(2,3,4,5,6),
                c(3,4,5,6,7),
                c(1,4,5,6,5)), ncol = 5)
mat
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    5    5    4
## [2,]    2    2    6    6    5
## [3,]    3    3    3    7    6
## [4,]    4    4    4    1    5
```

Similar to a vector, mixing the data classes will result in the data becoming all of the class character.

```
mat <- matrix(c(c(TRUE,2,3,4,5),
                 c(2,3,FALSE,5,6),
                 c("a","frog","fog","frg","frog_2.0"),
                 c(1,4,5,6,5),
                c(4,5,8,2,1)), ncol = 5)
mat
```

```
##      [,1] [,2] [,3]       [,4] [,5]
## [1,] "1"  "2"  "a"        "1"  "4"
## [2,] "2"  "3"  "frog"     "4"  "5"
## [3,] "3"  "0"  "fog"      "5"  "8"
## [4,] "4"  "5"  "frg"      "6"  "2"
## [5,] "5"  "6"  "frog_2.0" "5"  "1"
```

Data frames Data frames, are similar to matrices in that they are 2 dimensional sets of data. However, they contain **columns where each class can be different.**

This is the data type you will most often import your datasets as into R. In the example below, I am creating a data frame by combining equal length vectors via the **data.frame()** function. Notice that each vector is of a different class.

```
age <- c(1,2,5,4,5,4,2,3,7,6)
size_cm <- c(1.5,2.6,1,1,5,6,2.7,4,9.8,10)
habitat <- c("Urban","Urban","Urban","Suburban","Suburban","Suburban","Rural","Rural","Rural","Rural")
true_false <- c(T,T,F,T,F,T,T,T,F,F) #Using shorthand True/False

df <- data.frame(age,size_cm,habitat, true_false)

df
```

```
##    age size_cm  habitat true_false
## 1    1     1.5    Urban       TRUE
## 2    2     2.6    Urban       TRUE
## 3    5     1.0    Urban      FALSE
## 4    4     1.0 Suburban       TRUE
## 5    5     5.0 Suburban      FALSE
## 6    4     6.0 Suburban       TRUE
## 7    2     2.7    Rural       TRUE
## 8    3     4.0    Rural       TRUE
## 9    7     9.8    Rural      FALSE
## 10   6    10.0    Rural      FALSE
```

Lists Lists contain multiple objects within them. These can be any type of object such as single values, data frames, or even other lists! We will not use lists too often in this course, but they are very common outputs from functions in R.

The below example shows a list containing several vectors of equal length.

```
list <- list(x = c(1,2,3,4,5,6),
     y = c(1,45,67,54,23),
     z = c(5,5))
```

```
list
```

```
## $x
## [1] 1 2 3 4 5 6
##
## $y
## [1]  1 45 67 54 23
##
## $z
## [1] 5 5
```

All these different object and data types are critical to understanding R. Throughout the course we will refer to different types of data by their class and structure.

1.2.1 Loading Data

Loading Data

Now lets talk about loading data into R. As a reminder, your working directory needs to be set in order to properly load and save data in R.

Once your working directory is set, lets place the data we want to use inside of that folder.

Here we have a file named example.csv placed inside our working directory.

To load the .csv file into R, we will use the function **read.csv()**

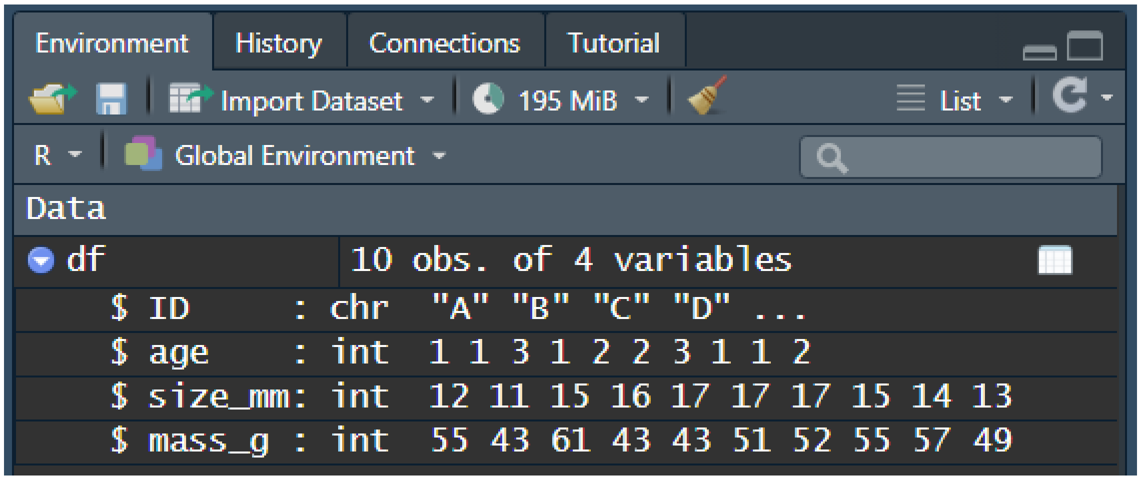
```
read.csv('example.csv')
```

```
##    ID age length_mm mass_g habitat_type
## 1   A   1        12     55        urban
## 2   B   1        11     43        urban
## 3   C   3        15     61        urban
## 4   D   1        16     43        urban
## 5   E   2        17     43     suburban
## 6   F   2        17     51     suburban
## 7   G   3        17     52     suburban
## 8   H   1        15     55        rural
## 9  ID   1        14     57        rural
## 10  J   2        13     49        rural
```

The above example simply reads the data. Reading the data is how we tell R to interpret a data set so that it can be used. If we want to use the data in R, it is best to assign it to an object. Lets do that, naming the object df

```
df <- read.csv('example.csv')
```

If you ran this bit of code yourself, you should see the object df under the environment window in the top right pane of your RStudio.



```
Data
df                10 obs. of 4 variables
    $ ID     : chr  "A" "B" "C" "D" ...
    $ age    : int  1 1 3 1 2 2 3 1 1 2
    $ size_mm: int  12 11 15 16 17 17 17 15 14 13
    $ mass_g : int  55 43 61 43 43 51 52 55 57 49
```

Important considerations

- Your working directory needs to be set correctly and the file you wish to load is placed inside that working directory
- The filename needs to be typed exactly as it appears (capitalizations, spaces, symbols etc.)
- You need to include the file extension. In the case of the example.csv dataset, the file extension is .csv. This file extension stands for Comma Separated Values. Every file has an extension however, they may be hidden on your device.
  - Refer to this documentation for assistance in windows: https://www.howtogeek.com/205086/beginner-how-to-make-windows-show-file-extensions/
  - For Mac: https://support.apple.com/guide/mac-help/show-or-hide-filename-extensions-on-mac-mchlp2304/mac

1.2.2 Saving Data

Saving Data

When we want to export data out of R there are 2 primary ways to save objects: Saving single objects to a file or saving multiple objects to a file that we can open with R.

Saving single objects Often we need to work with data in multiple different programs. Say you import a spreadsheet into R, do some type of manipulation, and then want to view it in excel. The same goes for wanting to export an object to view in GIS software, a visualization to put in a publication, etc etc

To save objects, we use the write family of functions.

In the same fashion that we **read data in to R**, we will **write data out of R.** There are a wide variety of write functions that we use for different types of data. Here I will show you how to use the **write.csv()** function.

write.csv() does exactly what you think. It writes an object out of R into a .csv file that you can very easily open up in your favorite spreadsheet viewing software.

Lets take that same data frame we read into R, and now lets write it into our working directory and name it exported_df.csv

```
write.csv(df,file = 'exported_df.csv')
```

The function above simply takes the dataframe **df**, and writes it out as the file **exported_df.csv.** Note that you need to add the file extension, in this case .csv, in order for the file to export correctly.

There are a wide variety of write functions suited for different types of data. I have included a few examples below. Note that many of these will require packages that are noted in the comment.

```
write.tree() #Writes phylogenetics tree from APE package

write.FASTA() #Writes a FASTA file from the APE package

st_write() #Writes a variety of spatial objects from the SF package

write_delim() #Writes out a table not using a comma delimiter from the reader package
```

Saving multiple objects What if you want to save multiple objects at once?

Lets say after mastering the material in this course, you create an advanced series of r scripts for your research. You might have started with a single data set, but now you have multiple data objects, results from analyses, and various visualizations that you don't want to rerun the next time you open up R.

This is where we will save and load .Rdata files

.Rdata files can contain many different objects. The chunk below creates many different objects.

```
num_val <- 2 #A single numeric value
char_val <- "frog" # A single character value

num_vect <- c(1,2,3,4,5) # a vector of numbers

df <- data.frame(col_1 = c(1,2,3,4,5), #A dataframe of 3 columns and 5 rows
                 col_2 = c('a','b','c','d','e'),
                 col_3 = c(22.4,23.7,34.2,12.3,45.3))

list <- list(num_val,char_val,num_vect,df) #a list that contains all the previous objects
```

What we can do, is place all of these different objects into a single .Rdata file which we can load later. We do this by using the **save()** function. Simply list out every object we want to save and then include a file name with the file argument.

```
save(df,list,char_val,num_val,num_vect, file = "ExampleRfile.Rdata")
```

Now if we want to load that same data in the future, we simply use the load() function. Lets clear the data in the environment using the next chunk to demonstrate this.

```
rm(list = ls()) #this nifty command removes everything in the environment
```

Now lets load the data back in using the **load()** function

```r
load('ExampleRFile.Rdata')
```

simple right? It is good practice to get in the habit of saving and loading collections of R objects after major steps in your analysis. This will help keep your code more organized and give you the benefit of not needing to store dozens of different files for a single analysis.

1.3.1 Intro to Plotting

Intro to Plotting

We can create a variety of visualizations using **plot().** **plot()** is a base R function for creating plots. Later in the course we will focus heavily on visualization with **ggplot2,** however, using base R plots are still useful in making simple graphs!

plot() has a variety of arguments we can use:

- x - the x-axis variable (independent variable)
- y - the y axis variable (optional; dependent variable)
- xlab - the label on the x axis
- ylab - the label on the y axis
- main - the title of the plot

There are quite a few more that can be found in the help text for plot! (?plot)

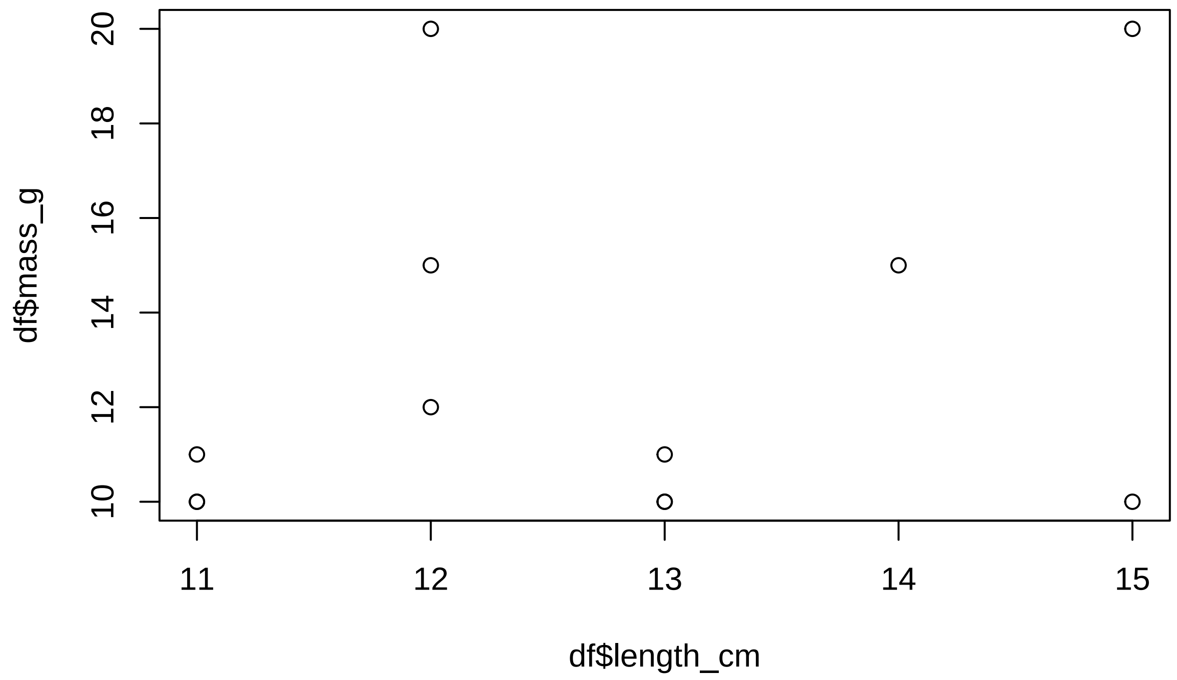
Lets load in a simple data frame we want to make a plot of. It contains 2 numeric data (mass_g and length_cm) and a character data (species)

```r
df <- data.frame(mass_g = c(10,20,10,15,10,10,15,20,11,10,12,11),
                 length_cm = c(11,12,15,12,11,13,14,15,11,13,12,13),
                 species = c("Species_A", "Species_A","Species_A","Species_A","Species_B","Species_B","
                 "Species_C","Species_C","Species_C", "Species_C"))
```

Scatter plot Lets make a simple scatter plot of length and mass. Scatter plots assume that both x and y are numeric data. We assume mass is dependent on length, thus length goes on the x axis.

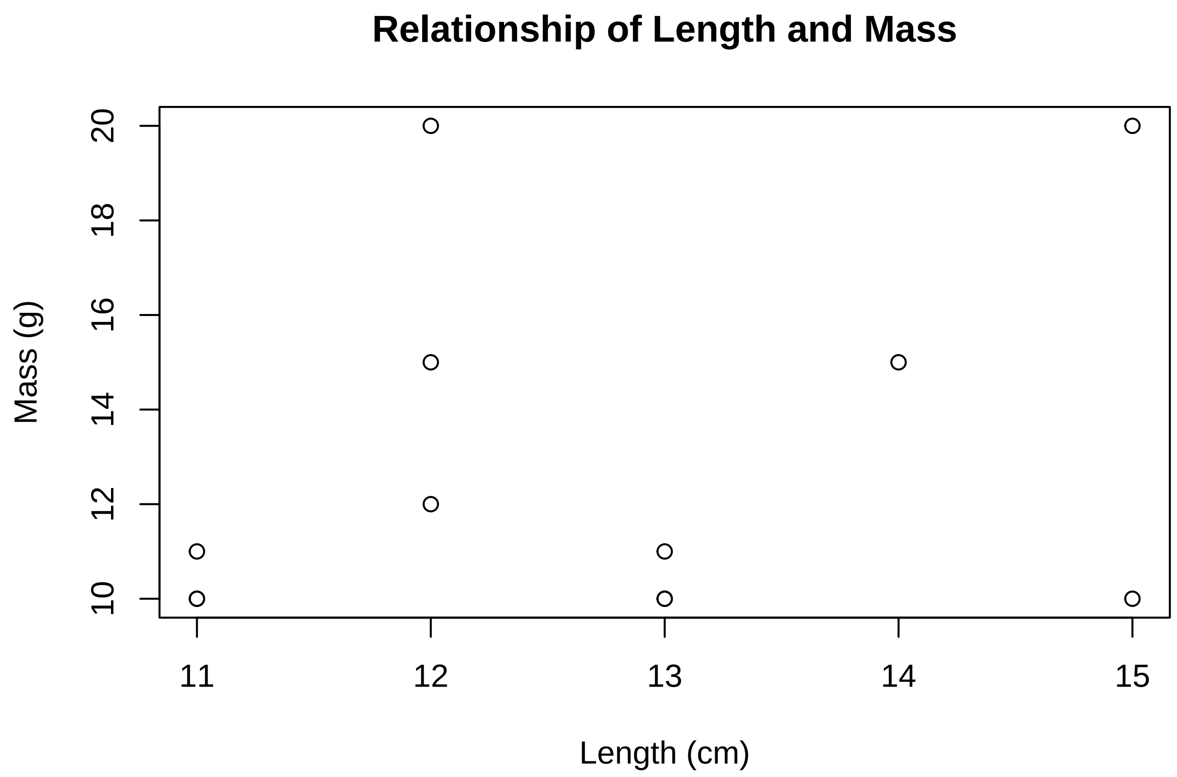
Note: To access data within a data frame, we use the $ next to the name of the object. This is covered in more detail in Section 2.

```r
plot(x = df$length_cm, y = df$mass_g)
```



Simple! Lets now add in the title, x axis label, and y axis label, using the arguments we mentioned.

```
plot(x = df$length_cm, y = df$mass_g,
     xlab = "Length (cm)", ylab = "Mass (g)",
     main = "Relationship of Length and Mass")
```

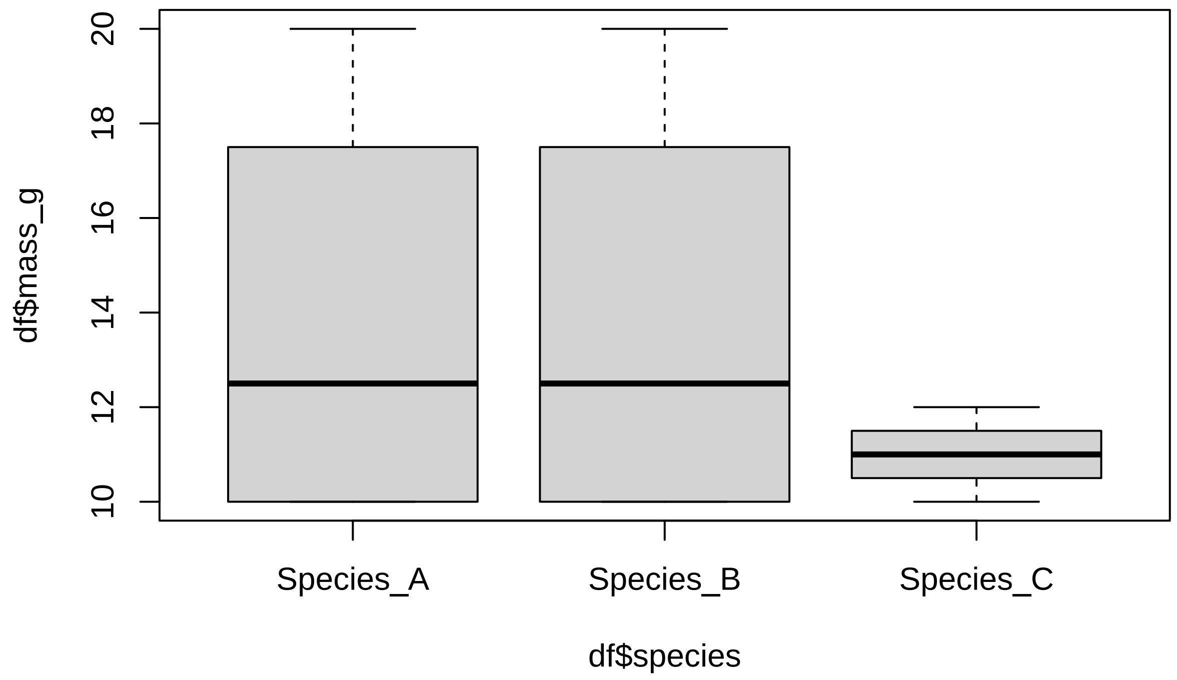


As a reminder, each argument needs to be separated by a comma and all arguments need to be within the parentheses.

Box plot We create box plots when we have character and numeric data. Lets create one for the variables mass_g and species!

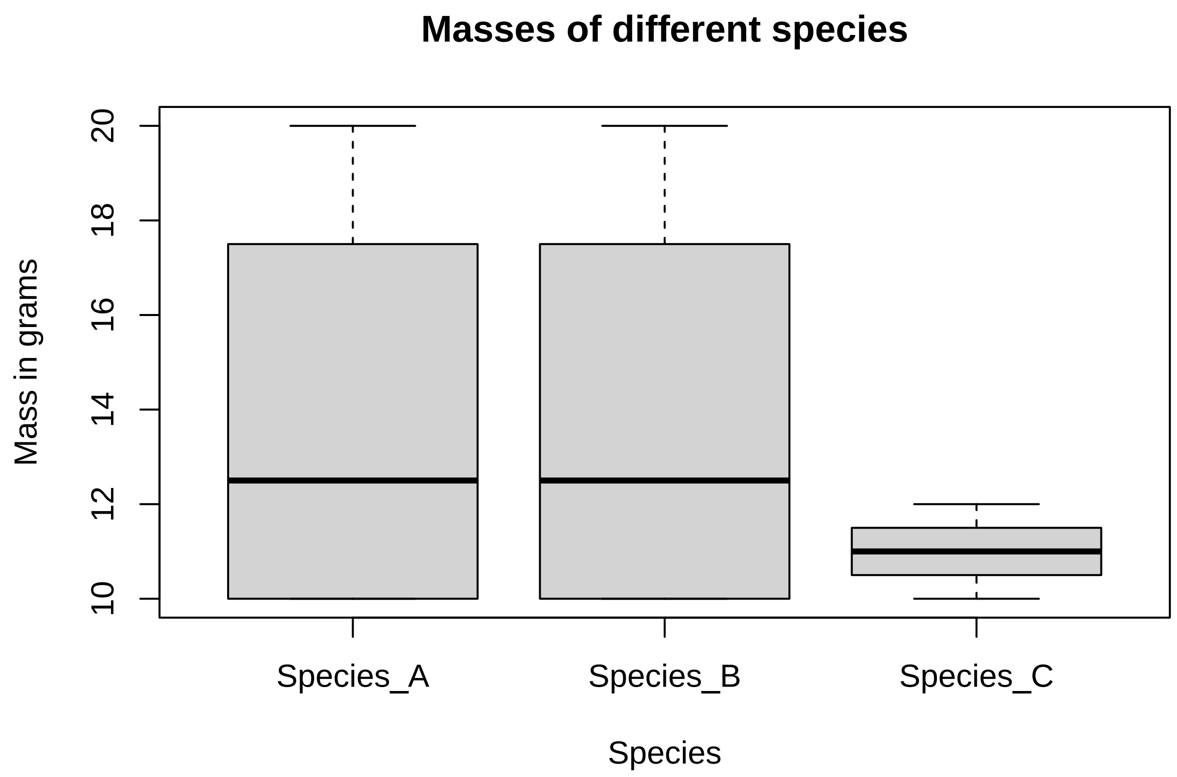
To do so, we use the **boxplot()** function. **boxplot()** uses similar arguments as **plot()**, the only difference is that the data input uses a format of y~x. y is still our dependent variable, while x is our independent variable.

```
boxplot(df$mass_g ~ df$species)
```



As mentioned earlier, we can specify our labels and title using the same arguments as **plot()**. Lets add those now.

```
boxplot(df$mass_g ~ df$species, main = "Masses of different species",
        xlab = "Species", ylab= "Mass in grams")
```



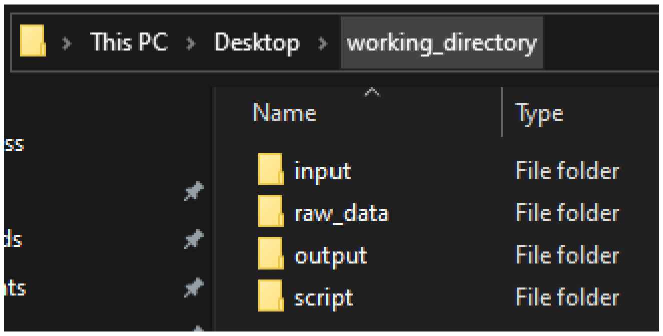
This has been just a brief introduction into plots! Once you want to start making more complex plots, I highly recommend learning the ggplot package we introduce in week 6!

Bonus: Tips for Data Organization

A quick note on data organization

It is very likely that your working directory is subject to get extremely messy, extremely quickly. Between all the different scripts, datasets, and documentation its easy to get overwhelmed. Here are some suggestions to keep this in order.

Folder Organization In general, its a good idea to have a folder for each scripts, inputs, outputs, and raw_data in your working directory.



This is easy to work with in R. If your working directory is set properly, you can access these subfolders in your functions by placing the name of the subfolder in front of the file name.

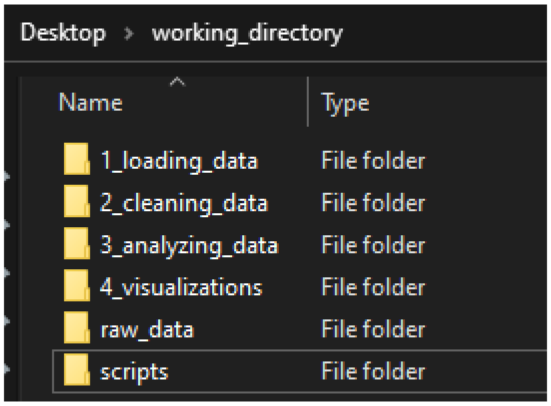
Take the example below which assumes you have your .csv file in a subfolder in your working directory named inputs. It will first read that csv and then write a variety of objects into the outputs folder, which is later loaded by the load() function.

```r
#read the data contained in the inputs subfolder
read.csv('inputs/example.csv')

#save the data into an outputs folder
save(df,list,char_valu,file = "outputs/output_file.rdata")

#load the data found in the outputs folder
load(file = 'outputs/output_file.rdata')
```

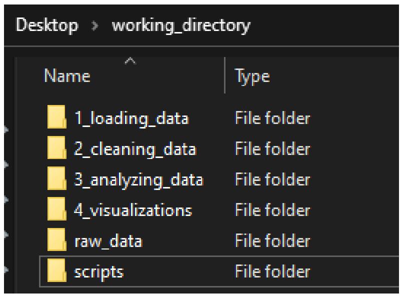
Organization of your project before you even begin to write a line of code can make a huge difference for keeping everything organized, making sure you understand your analysis, and can save you hours of lost effort just trying to find where a particular file is.

Other considerations

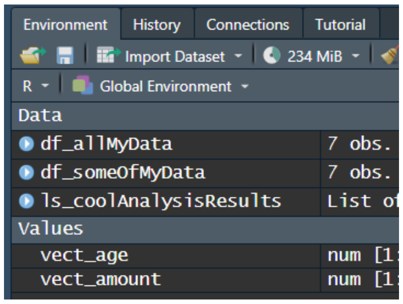
- Segmenting you analysis into ordered steps.



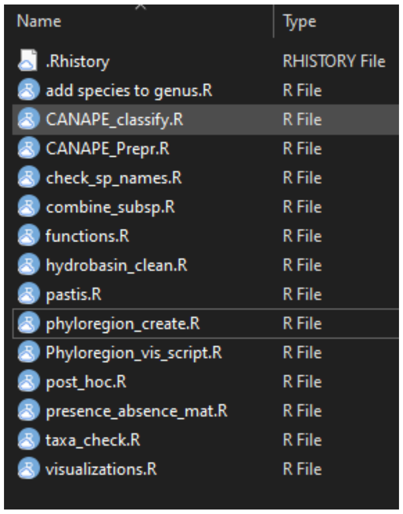
- Trying to keep your labeling consistent for objects, files, and folders. Use short, yet descriptive names.



Name	Type
1_loading_data	File folder
2_cleaning_data	File folder
3_analyzing_data	File folder
4_visualizations	File folder
raw_data	File folder
scripts	File folder

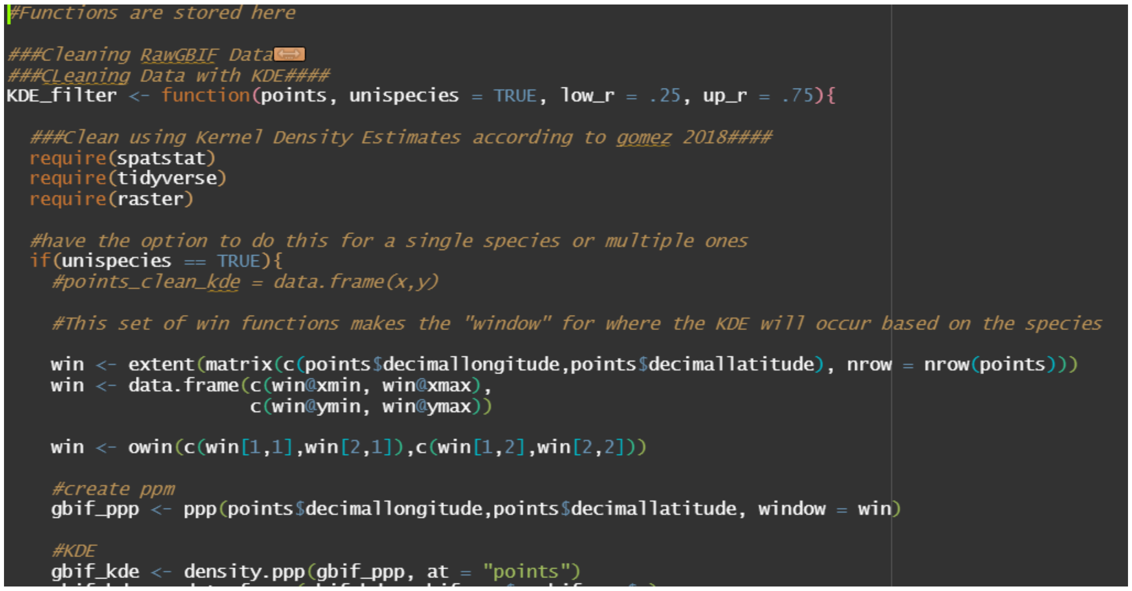


```
Data
df_allMyData             7 obs.
df_someOfMyData          7 obs.
ls_coolAnalysisResults   List of
Values
vect_age                 num [1:
vect_amount              num [1:
```



Name	Type
.Rhistory	RHISTORY File
add species to genus.R	R File
CANAPE_classify.R	R File
CANAPE_Prepr.R	R File
check_sp_names.R	R File
combine_subsp.R	R File
functions.R	R File
hydrobasin_clean.R	R File
pastis.R	R File
phyloregion_create.R	R File
Phyloregion_vis_script.R	R File
post_hoc.R	R File
presence_absence_mat.R	R File
taxa_check.R	R File
visualizations.R	R File

- Keeping good documentation of how your code works and what it interacts with.
  - Heavy use of comments and/or using RMarkdown documents are great solutions!



```r
#Functions are stored here

###Cleaning RawGBIF Data
###CLeaning Data with KDE####
KDE_filter <- function(points, unispecies = TRUE, low_r = .25, up_r = .75){

  ###Clean using Kernel Density Estimates according to gomez 2018####
  require(spatstat)
  require(tidyverse)
  require(raster)

  #have the option to do this for a single species or multiple ones
  if(unispecies == TRUE){
    #points_clean_kde = data.frame(x,y)

    #This set of win functions makes the "window" for where the KDE will occur based on the species

    win <- extent(matrix(c(points$decimallongitude,points$decimallatitude), nrow = nrow(points)))
    win <- data.frame(c(win@xmin, win@xmax),
                      c(win@ymin, win@ymax))

    win <- owin(c(win[1,1],win[2,1]),c(win[1,2],win[2,2]))

    #create ppm
    gbif_ppp <- ppp(points$decimallongitude,points$decimallatitude, window = win)

    #KDE
    gbif_kde <- density.ppp(gbif_ppp, at = "points")
```

Again, there is no perfect method for managing data. However, even a tiny bit of imperfect data organization can make your life all the more easy. I implore you to think about data organization, commenting your code, and keeping track of everything whenever you start working on a project.

I promise it will go a long way!

Section 2 - Manipulating Objects and Logic

2.0 Section 2 Overview

Week 2 Overview

At this stage of the course, we know the basic object and data types, we know the different classes of data , and we know some of the basic R principles.

This week we will cover data manipulation via subsetting and combining datasets. In short, subsetting allows us to extract data via index numbers, filters, or some other criteria. Combining involves adding new information to a data set from another piece of data.

We will cover:

- Subsetting via index number for vectors, data frames, and lists
- Subsetting using logic
- Adding rows via **rbind()**
- Adding columns via **cbind()**
- Merging data frames using **merge()**

The principles in this section will be used throughout your R career and will allow us to introduce more advanced concepts in later weeks!

2.1.1 Subsetting Vectors

Subsetting vectors with index values

This section will cover how to subset vector type data using base R. Base R means that we require no outside packages to perform the tasks listed below.

As review, vectors are 1-dimensional data where all values are of the same type. Lets create a vector of numbers that range from 1 to 10. We will use the c function here. c() combines comma-separated values into a single object.

```
vect <- c(2.4,1.1,3,4.5,7.8,2.3,3.3,1.1,5.5,7)
```

We can subset and extract values using their index.

An index signifies where in an object a particular value lives. In the vector we created, the first value has an index of 1. The second value, an index of 2. So on and so forth.

We can subset a vector by specifying the index number we wish to extract within square brackets [ ]. The example below will print all the values in vect, and then only the 4th value (4.5).

```
vect

vect[4]
```

```
##  [1] 2.4 1.1 3.0 4.5 7.8 2.3 3.3 1.1 5.5 7.0
## [1] 4.5
```

We can extract multiple values at once by listing out all the indices inside the c() function.

```
#Pull the 1st, 5th, and 7th values
vect[c(1,5,7)]

#pull the 2nd, 4th, and 8th, values

vect[c(2,4,8)]
```

```
## [1] 2.4 7.8 3.3
## [1] 1.1 4.5 1.1
```

We can also specify a range of values to extract. To do so, place the bounds of the positions, separated with a colon (:).

```
#pull the first 5 values
vect[1:5]

#Pull the values 3 through 6

vect[3:6]
```

```
## [1] 2.4 1.1 3.0 4.5 7.8
## [1] 3.0 4.5 7.8 2.3
```

The colon denotation can be combined with c for greater flexibility.

```
#Pull the first 5 values as well as the 7th, and the 10th values

vect[c(1:5,7,10)]
```

```
## [1] 2.4 1.1 3.0 4.5 7.8 3.3 7.0
```

This is a very basic overview of how to subset vectors with index values. Subsetting with indices is the most basic, and thus most limited, way to subset. In future sections, we will learn how to subset any type of data using logic.

2.1.2 Subsetting Dataframes

Subsetting Dataframes

In R, you will be working most often with dataframes. As a reminder, these are 2-dimensional datasets organized in columns and rows.

Lets make a simple dataframe named df that includes ID, Habitat type, Length in Millimeters, and age

```
df <- data.frame(ID = c(1,2,3,4,5,6,7,8,9,10),
                 Habitat_type = c('Urban','Urban','Urban',
                                  'Suburban','Suburban','Suburban',
                                  'Rural','Rural','Rural','Rural'),
                 length_mm = c(2.4,2.5,3.6,3.6,2.1,2.2,5,1.9,2.3,2.2),
                 age = c(1,2,2,2,1,1,3,3,4,1))
df
```

```
##    ID Habitat_type length_mm age
## 1   1        Urban       2.4   1
## 2   2        Urban       2.5   2
## 3   3        Urban       3.6   2
## 4   4     Suburban       3.6   2
## 5   5     Suburban       2.1   1
## 6   6     Suburban       2.2   1
## 7   7        Rural       5.0   3
## 8   8        Rural       1.9   3
## 9   9        Rural       2.3   4
## 10 10        Rural       2.2   1
```

To extract a single column of data by name, we use the name of the dataframe, in our case df, followed by the $ symbol.

```
#Extract just the length_mm column
df$length_mm

#Extract just the ID columns
df$ID
```

```
##  [1] 2.4 2.5 3.6 3.6 2.1 2.2 5.0 1.9 2.3 2.2
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Naturally, we can extract this data and pipe it directly into a function. For example, finding the average length of our length_mm column.

```
#Get the mean length
mean(df$length_mm)
```

```
## [1] 2.78
```

This method works great if we want to extract a single column of data, but what if we want to extract multiple columns, or a column and a few rows?

That's where the square brackets and index numbers come into play again

Subsetting with index numbers Using the same format as the vectors, where its the name of the object followed by the square brackets. Inside the square brackets we will input 2 index values into the brackets separated by a comma. This is because the data is 2-dimensional and we can subset by rows or columns.

The value to the left of the comma will correspond to rows. In this below example, we are going to pull the first 5 rows. Since we do not specify anything on the column side, all of the columns are pulled.

```
df[1:5,]
```

```
##   ID Habitat_type length_mm age
## 1  1        Urban       2.4   1
## 2  2        Urban       2.5   2
## 3  3        Urban       3.6   2
## 4  4     Suburban       3.6   2
## 5  5     Suburban       2.1   1
```

The right hand value corresponds to the columns. For this value, we can use the position of the columns or their names placed in quotation marks. The next two chunks will pull the same columns using both of these formats.

```
df[,c(3,4)]
```

```
##    length_mm age
## 1        2.4   1
## 2        2.5   2
## 3        3.6   2
## 4        3.6   2
## 5        2.1   1
## 6        2.2   1
## 7        5.0   3
## 8        1.9   3
## 9        2.3   4
## 10       2.2   1
```

```
df[,c('length_mm','age')]
```

```
##    length_mm age
## 1        2.4   1
## 2        2.5   2
## 3        3.6   2
## 4        3.6   2
## 5        2.1   1
## 6        2.2   1
## 7        5.0   3
## 8        1.9   3
## 9        2.3   4
## 10       2.2   1
```

Naturally, we can combine subsets for rows and columns. Like this example that pulls the first three rows of the age column.

```
df[1:3,'age']
```

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

You may have noticed that leaving either side blank will pull all of the rows or columns respectively. If we leave both sides blank, all of the information contained within the object will be pulled.

```
df[,]
```

```
##    ID Habitat_type length_mm age
## 1   1        Urban       2.4   1
## 2   2        Urban       2.5   2
## 3   3        Urban       3.6   2
## 4   4     Suburban       3.6   2
## 5   5     Suburban       2.1   1
## 6   6     Suburban       2.2   1
## 7   7        Rural       5.0   3
## 8   8        Rural       1.9   3
## 9   9        Rural       2.3   4
## 10 10        Rural       2.2   1
```

2.1.3 Subsetting Lists

Subsetting Lists

Lists are collections of objects in R. Model outputs, collections of data, and some custom classes all use the list format.

Lets create a simple list that contains a single vector and a single dataframe.

```
#A simple vector of the numbers 1 through 10
vect <- c(2.4,1.1,3,4.5,7.8,2.3,3.3,1.1,5.5,7)

#a dataframe that includes ID, Habitat Type, Length in millimetes, and age
df <- data.frame(ID = c(1,2,3,4,5,6,7,8,9,10),
                 Habitat_type = c('Urban','Urban','Urban',
                                  'Suburban','Suburban','Suburban',
                                  'Rural','Rural','Rural','Rural'),
                 length_mm = c(2.4,2.5,3.6,3.6,2.1,2.2,5,1.9,2.3,2.2),
                 age = c(1,2,2,2,1,1,3,3,4,1))

#A ist that contains just the vector and the dataframe
list <- list(vect,df)

list
```

```
## [[1]]
##  [1] 2.4 1.1 3.0 4.5 7.8 2.3 3.3 1.1 5.5 7.0
##
## [[2]]
##    ID Habitat_type length_mm age
## 1   1        Urban       2.4   1
## 2   2        Urban       2.5   2
## 3   3        Urban       3.6   2
## 4   4     Suburban       3.6   2
## 5   5     Suburban       2.1   1
## 6   6     Suburban       2.2   1
## 7   7        Rural       5.0   3
## 8   8        Rural       1.9   3
## 9   9         Rural       2.3   4
## 10  10        Rural       2.2   1
```

Much like vectors and data frames, you can use square brackets to extract an element from the list using its index value. **However, you will use 2 square brackets to extract an object from the list.**

```
#extract the first element in a list. In this case its our vector
list[[1]]

#extract the second element in a list. In this case its our data frame
list[[2]]
```

```
##  [1] 2.4 1.1 3.0 4.5 7.8 2.3 3.3 1.1 5.5 7.0
##    ID Habitat_type length_mm age
## 1   1        Urban       2.4   1
## 2   2        Urban       2.5   2
## 3   3        Urban       3.6   2
## 4   4     Suburban       3.6   2
## 5   5     Suburban       2.1   1
## 6   6     Suburban       2.2   1
## 7   7        Rural       5.0   3
## 8   8        Rural       1.9   3
## 9   9        Rural       2.3   4
## 10 10        Rural       2.2   1
```

If a list has named elements, you can extract each element using the $ operator just like a dataframe. Lets recreate our list, naming the vector, vect_list, and our dataframe df_list

```
named_list <- list(vect_list = vect,
     df_list = df)

named_list$df_list
```

```
##    ID Habitat_type length_mm age
## 1   1        Urban       2.4   1
## 2   2        Urban       2.5   2
## 3   3        Urban       3.6   2
## 4   4     Suburban       3.6   2
## 5   5     Suburban       2.1   1
## 6   6     Suburban       2.2   1
## 7   7        Rural       5.0   3
## 8   8        Rural       1.9   3
## 9   9        Rural       2.3   4
## 10 10        Rural       2.2   1
```

Or you use the name of the object in the double brackets

```
named_list[['vect_list']]
```

```
##  [1] 2.4 1.1 3.0 4.5 7.8 2.3 3.3 1.1 5.5 7.0
```

Extracting information from objects in the list If we want to extract further information from an object in the list, we can use the appropriate operator (single brackets, index numbers, $ etc.) for the object we are extracting.

For example, here we extract the vector from the list and then pull the 5th element. The vector is the first object in the list.

```
list[[1]][5]
```

```
## [1] 7.8
```

The same logic applies to dataframes. In the following code we first pull the second object in the list which is the dataframe. Then we pull the first five rows from the length_mm column.

```
#Pull the second object in our list (the data frame) and then pull the length column from there
list[[2]][1:5,'length_mm']
```

```
## [1] 2.4 2.5 3.6 3.6 2.1
```

As a reminder, there are often multiple ways of doing each of these operations and its totally up to you how you decide to do it. In the next example, we use the $ instead of the square brackets on the named_list and then extract the 5th value from the vector using the index number.

```
named_list$vect_list[5]
```

```
## [1] 7.8
```

What is important to understand about coding is that you can accomplish the same task in multiple ways. By understanding the basics of subsetting, you can chain together more complex subsets to achieve whatever your project requires of you!

2.2.1 Intro to Logic

Intro to Logic

While we could subset every single dataset using just the index numbers, this may prove to be cumbersome or impossible with large datasets.

For example, lets make a vector with 100 numbers. We will also use the colon to specify that we are creating a vector with the numbers 1 through 100.

```
vect <- c(1:100)
vect
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

We can make subsetting easy by using logic. Logic allows us to evaluate all of the values within an object according to some criteria. Each value in the object, will return a TRUE or FALSE value based on the logical statement that you provide. Those TRUE/FALSE values are then used to subset the object.

Common logical operators includes:

- > Greater than
- < Less than
- >= Greater than or equal to
- <= Less than or equal to
- == Equal to
- != Not Equal to

When we create logical statements, we need to first say what object we are checking, include the appropriate logical operator, and then the value(s) we are checking.

For example, lets use all of this logic on the vector we created around the number 30! Note that since we are subsetting the object vect, we will still use the square brackets to indicate that it is a subset.

Pull values that are greater than 30.

```
vect[vect > 30]
```

```
##  [1]  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49
## [20]  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
## [39]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87
## [58]  88  89  90  91  92  93  94  95  96  97  98  99 100
```

Pull values that are less than 30.

```
vect[vect < 30]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29
```

Pull values that are less than or equal to 30.

```
vect[vect <= 30]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30
```

Pull values that are greater than or equal to 30.

```
vect[vect >= 30]
```

```
##  [1]  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48
## [20]  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67
## [39]  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86
## [58]  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

Pull values that are equal to 30. Note that you need to use two equal signs.

```
vect[vect == 30]
```

```
## [1] 30
```

The ! means not. Pull the values that are NOT equal to 30.

```
vect[vect != 30]
```

```
##  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
## [20]  20  21  22  23  24  25  26  27  28  29  31  32  33  34  35  36  37  38  39
## [39]  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58
## [58]  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77
## [77]  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96
## [96]  97  98  99 100
```

Logical statements can be chained together to create more complex filters. The next lesson will cover how to use logic on dataframes and how to create AND as well as OR statements in your logic!

2.2.2 Combining logical statements

Logic with 'and' and 'or' What if we want to create more complex logical statements? For example, all values in the dataset that are greater than 30 AND are from a particular location.

We can combine logical terms using **&** (for AND statements) as well as | (for OR statements).

The operator for OR statements, this symbol |, is called the pipe symbol and is NOT the uppercase i or the lowercase L. You can typically find it on the right side of your keyboard above the enter key.

Lets make a simple dataframe with 3 different sites and the number of species found during 4 different surveys

```
df <- data.frame(SITE_ID = c("A100", 'A100','A100','A100',
                             'B100','B100','B100','B100',
                             'C100','C100','C100','C100'),
                 survey = c(1,2,3,4,1,2,3,4,1,2,3,4),
                 num_species = c(10,7,17,18,10,12,11,13,18,19,20,17))
```

We can use logic on this dataframe just like we did with the vector. Remember that we need to specify the column we are applying the logic unto using the $.

This example says we want to subset the rows where SITE_ID is equal to A100. Remember, we need 2 equal signs for logical statements.

```
df[df$SITE_ID == 'A100',]
```

```
##   SITE_ID survey num_species
## 1    A100      1          10
## 2    A100      2           7
## 3    A100      3          17
## 4    A100      4          18
```

If we want to combine logical terms, we just separate each piece of logic by either & or |

Lets find values where the SITE_ID is equal to A100 AND the number of species found is less than 15

```
df[df$SITE_ID == 'A100' & df$num_species < 15,]
```

```
##   SITE_ID survey num_species
## 1    A100      1          10
## 2    A100      2           7
```

Great! We found 2 entries in our dataset!

Lets run that again, but changing it from AND (&) to OR (|)

```
df[df$SITE_ID == 'A100' | df$num_species < 15,]
```

```
##   SITE_ID survey num_species
## 1    A100      1          10
## 2    A100      2           7
## 3    A100      3          17
## 4    A100      4          18
## 5    B100      1          10
## 6    B100      2          12
## 7    B100      3          11
## 8    B100      4          13
```

We can see there are many more values. Often newcomers to logic have trouble with using AND logic or OR logic effectively. In general, AND logic will be more restrictive, while OR logic will be more inclusive.

You can use any combination of logical operators alongside AND/OR to filter nearly any dataset you are working with!

2.3.1 Adding new columns

Adding new columns With dataframes we often need to add new columns. These columns could be an ID, a new vector of data, or are calculated based on some other column. Luckily, this is very straightforward!

Lets create the dataframe we will use.

```
df <- data.frame(Habitat_type = c('Urban','Urban','Urban',
                                  'Suburban','Suburban','Suburban',
                                  'Rural','Rural','Rural','Rural'),
                 length_mm = c(2.4,2.5,3.6,3.6,2.1,2.2,5,1.9,2.3,2.2),
                 age = c(1,2,2,2,1,1,3,3,4,1))
df
```

```
##    Habitat_type length_mm age
## 1         Urban       2.4   1
## 2         Urban       2.5   2
## 3         Urban       3.6   2
## 4      Suburban       3.6   2
## 5      Suburban       2.1   1
## 6      Suburban       2.2   1
## 7          Rural       5.0   3
## 8          Rural       1.9   3
## 9          Rural       2.3   4
## 10         Rural       2.2   1
```

We can add a column by using the $ operator we learned in the subsetting section. Lets add an ID column that just adds a number for each of the 10 rows.

```
df$ID <- 1:10

df
```

```
##    Habitat_type length_mm age  ID
## 1         Urban       2.4   1   1
## 2         Urban       2.5   2   2
## 3         Urban       3.6   2   3
## 4      Suburban       3.6   2   4
## 5      Suburban       2.1   1   5
## 6      Suburban       2.2   1   6
## 7         Rural       5.0   3   7
## 8         Rural       1.9   3   8
## 9         Rural       2.3   4   9
## 10        Rural       2.2   1  10
```

We can of course create a vector to add to the dataframe.

```
df$ID <- c('a','b','c','d','e','f','g','h','i','j')

df
```

```
##    Habitat_type length_mm age ID
## 1         Urban       2.4   1  a
## 2         Urban       2.5   2  b
## 3         Urban       3.6   2  c
## 4      Suburban       3.6   2  d
## 5      Suburban       2.1   1  e
## 6      Suburban       2.2   1  f
## 7         Rural       5.0   3  g
## 8         Rural       1.9   3  h
## 9         Rural       2.3   4  i
## 10        Rural       2.2   1  j
```

In the above chunk we used the same name as the ID column as the prior chunk. This overwrote the data contained within that column. Be careful with accidentally overwriting data!

We can also perform simple calculations while adding data. Lets say we want to convert all the length values from mm to cm. We simply divide the mm column by 10.

```
df$length_cm <- df$length_mm/10

df
```

```
##    Habitat_type length_mm age ID length_cm
## 1         Urban       2.4   1  a      0.24
## 2         Urban       2.5   2  b      0.25
## 3         Urban       3.6   2  c      0.36
## 4      Suburban       3.6   2  d      0.36
## 5      Suburban       2.1   1  e      0.21
## 6      Suburban       2.2   1  f      0.22
## 7         Rural       5.0   3  g      0.50
## 8         Rural       1.9   3  h      0.19
## 9         Rural       2.3   4  i      0.23
## 10        Rural       2.2   1  j      0.22
```

When adding new columns, **we need to make sure the data being added is equal to the number of rows, or the number of rows is divisible by the data being added.** An example is in order.

In the above examples, we added 10 values to the dataframe because it already had 10 rows. What happens if we try to add 12 values?

```
df$ID <- c(1,2,3,4,5,6,7,8,9,10,11,12)
```

```
## Error in `$<-.data.frame`(`*tmp*`, ID, value = c(1, 2, 3, 4, 5, 6, 7, : replacement has 12 rows, dat
```

We get an error! That is because there are too many values for that dataframe.

Now what happens if we try to add 5 numbers instead?

```
df$ID <- c(1,2,3,4,5)

df
```

```
##    Habitat_type length_mm age ID length_cm
## 1         Urban       2.4   1  1      0.24
## 2         Urban       2.5   2  2      0.25
## 3         Urban       3.6   2  3      0.36
## 4      Suburban       3.6   2  4      0.36
## 5      Suburban       2.1   1  5      0.21
## 6      Suburban       2.2   1  1      0.22
## 7         Rural       5.0   3  2      0.50
## 8         Rural       1.9   3  3      0.19
## 9         Rural       2.3   4  4      0.23
## 10        Rural       2.2   1  5      0.22
```

The data actually gets added twice! That is because 10 is divisible by 5. If we tried to add 4 numbers instead, we get a different result.

```
df$ID <- c(1,2,3,4)
```

```
## Error in `$<-.data.frame`(`*tmp*`, ID, value = c(1, 2, 3, 4)): replacement has 4 rows, data has 10
```

We get another error. That is because 10 is not divisible by 4 into whole numbers.

When adding or overwriting the data in a column, ensure you are adding the same amount of data as rows, or an amount that the number of rows is divisible by.

2.3.2 Combining data with rbind() and cbind()

Combining data with rbind() and cbind()

Combining datasets will happen often through your R career. For biology, we often combine datasets from multiple survey events or to pair information found in different datasets.

Lets load in our first dataset.

```
df <- data.frame(Habitat_type = c('Urban','Urban','Urban',
                                  'Suburban','Suburban','Suburban',
                                  'Rural','Rural','Rural','Rural'),
                 length_mm = c(2.4,2.5,3.6,3.6,2.1,2.2,5,1.9,2.3,2.2),
                 age = c(1,2,2,2,1,1,3,3,4,1))
```

This first dataset is for survey *a*. We will create a new ID column, where all values are *a*.

```
df$ID <- "a"
```

A few weeks later we conduct another survey and want to combine the data with survey a. We'll create new survey data and give it an id of *b*.

```
df_2 <-  data.frame( ID = "b",
  Habitat_type = c('Rural','Urban','Rural',
                                  'Suburban','Urban','Suburban',
                                  'Rural','Urban','Rural','Rural'),
                 length_mm = c(1.4,1.9,7.8,3.1,2.2,2.7,4,3.4,2.3,4.5),
                 age = c(1,1,2,1,1,2,3,1,4,4))
```

rbind() We can combine these 2 datasets using the **rbind()** function. The r in **rbind()** stands for row and it simply adds rows to a dataset from another dataset.

```
rbind(df,df_2)
```

```
##    Habitat_type length_mm age ID
## 1         Urban       2.4   1  a
## 2         Urban       2.5   2  a
## 3         Urban       3.6   2  a
## 4      Suburban       3.6   2  a
## 5      Suburban       2.1   1  a
## 6      Suburban       2.2   1  a
## 7         Rural       5.0   3  a
## 8         Rural       1.9   3  a
## 9         Rural       2.3   4  a
## 10        Rural       2.2   1  a
## 11        Rural       1.4   1  b
## 12        Urban       1.9   1  b
## 13        Rural       7.8   2  b
## 14     Suburban       3.1   1  b
## 15        Urban       2.2   1  b
## 16     Suburban       2.7   2  b
## 17        Rural       4.0   3  b
## 18        Urban       3.4   1  b
## 19        Rural       2.3   4  b
## 20        Rural       4.5   4  b
```

For rbind to work properly, each dataset must have the same number of columns and the columns must be named identically.

cbind() Lets say we want to add columns of data instead. In this case I created a dataframe that has Site ID and the day each survey took place.

```
df_col <- data.frame(Site_ID = c('a','b','c','b','b','b','c','d','c','c'),
           Day = c(1,2,3,4,5,5,5,6,6,7))
```

Instead of binding rows, we are binding columns. So we use the **cbind()** command

```
cbind(df,df_col)
```

```
##    Habitat_type length_mm age  ID Site_ID Day
## 1         Urban       2.4   1  a       a   1
## 2         Urban       2.5   2  a       b   2
## 3         Urban       3.6   2  a       c   3
## 4      Suburban       3.6   2  a       b   4
## 5      Suburban       2.1   1  a       b   5
## 6      Suburban       2.2   1  a       b   5
## 7         Rural       5.0   3  a       c   5
## 8         Rural       1.9   3  a       d   6
## 9         Rural       2.3   4  a       c   6
## 10        Rural       2.2   1  a       c   7
```

Similarly to **rbind(), cbind()** needs the same number of rows in each dataset to combine properly. Also, the order will matter. When using **cbind()** you must be very careful to ensure each set of data has the data in the correct order.

The next lesson will introduce the **merge()** function, which is a more powerful and flexible way of combining datasets together. However, **rbind()** and **cbind()** are still powerful tools to have in your R-senal

2.3.3 Combining Data with merge()

Combining data with merge() While **cbind()** and **rbind()** are certainly useful, they are limited in what they can accomplish. The **merge()** function is a more flexible way of combining datasets and simultaneously handles combining by rows and by columns.

Lets make 2 new datasets. The first (turt_morpho) has morphological information for turtles caught during a tracking study. The second (turt_habitat) will tell us the location the turtle was first found and the distance to water.

Notice that in each dataset we identify unique turtles according to the column *Turtle_ID* and the same turtles show up in each dataset.

```r
turt_morpho <- data.frame(Turtle_ID = c('a','b','c','d','e'),
                          Carapace_length_cm = c(130,140,75,49,120),
                          Mass_g = c(500,550,300,200,435))

turt_habitat <- data.frame(Turtle_ID = c('a','b','c','d','e'),
                           Location = c('Orchard','Orchard','Wetland','Wetland','Wetland'),
                           Dist_m = c(10,50,200,3,0))
```

merge() needs a column shared between the two datasets, typically an ID column. We will merge these 2 datasets based on the *Turtle_ID* column shared between them.

```r
merge(turt_morpho,turt_habitat, by = "Turtle_ID")
```

```
##   Turtle_ID Carapace_length_cm Mass_g Location Dist_m
## 1         a                130    500  Orchard     10
## 2         b                140    550  Orchard     50
## 3         c                 75    300  Wetland    200
## 4         d                 49    200  Wetland      3
## 5         e                120    435  Wetland      0
```

Easy peasy!

We can combine datasets together with different column names by using the *by.x* and the *by.y* arguments in **merge()**. Lets make a new dataset indicating the weather when each turtle was found. In this dataset, the column for the turtle id is simply *ID* instead of *Turtle_ID*

We will also include 2 new turtles with the ID F and G.

```r
turt_weather <- data.frame(ID = c('a','b', 'c', 'd', 'e','f','g'),
                           Weather = c('sunny','sunny','cloudy','cloudy','cloudy','cloud','sunny'))
```

Lets merge them. *by.x* relates to the first dataset and *by.y* related to the second dataset.

```r
merge(turt_habitat,turt_weather, by.x = "Turtle_ID",by.y = "ID")
```

```
##   Turtle_ID Location Dist_m Weather
## 1         a  Orchard     10   sunny
## 2         b  Orchard     50   sunny
## 3         c  Wetland    200  cloudy
## 4         d  Wetland      3  cloudy
## 5         e  Wetland      0  cloudy
```

Considerations with merge With merge, we do need to be aware of a few things. If one dataset has rows not found in the other, the default behavior is to exclude them during the merge. That is why in the above merged dataset, we did not see turtles F and G.

We can change that behavior by setting the arguments *all.x* and *all.y* to TRUE or FALSE. The default is FALSE for each. Setting them to TRUE will keep all the data. Where data is missing, the function will introduce NA values.

```
merge(x = turt_habitat,y = turt_weather,
      by.x = "Turtle_ID",by.y = "ID",
      all.x  = FALSE, all.y = TRUE)
```

```
##   Turtle_ID Location Dist_m Weather
## 1         a  Orchard     10   sunny
## 2         b  Orchard     50   sunny
## 3         c  Wetland    200  cloudy
## 4         d  Wetland      3  cloudy
## 5         e  Wetland      0  cloudy
## 6         f     <NA>     NA   cloud
## 7         g     <NA>     NA   sunny
```

Merging to include NA values or not is neither right nor wrong, it all depends on your data, your analyses, and exactly what you hope to accomplish.

Sometimes you want to exclude the data that doesn't have an overlap and sometimes you want to include it!

Section 3 - Introduction to Tidyverse

3.0 Section 3 Overview

Section 3 Overview

Tidyverse is an extremely useful package that will serve you well throughout your R career.

This week will introduce concepts such as:

- What is tidyverse?
- What is a pipeline?
- How can we use tidyverse to clean and manipulate our data?

The data we will use for this week, is based on a field survey of 3 cacti species cacti found in Baja California. The data is a simple data frame named cactus_df that has 3 variables:

- species - character data that lists the scientific name of each species ("*F. gracilis*", "*F. gatesii*", "*F. cylindraceus*")
- spine_length_cm - numeric data of how long the average spine length of each specimen in centimeters.
- lifestage - character data that specifies if each specimen was "Immature" or "Mature".

```r
set.seed(123)

cactus_df <- data.frame(

  species = rep(c("F. gracilis","F. gatesii","F. cylindraceus"),each = 10),

  spine_length_cm = rnorm(30,8,2),

  lifestage = rep(c("Mature", "Immature"),15))

cactus_df
```

```
##            species spine_length_cm lifestage
## 1      F. gracilis        6.879049    Mature
## 2      F. gracilis        7.539645  Immature
## 3      F. gracilis       11.117417    Mature
## 4      F. gracilis        8.141017  Immature
## 5      F. gracilis        8.258575    Mature
## 6      F. gracilis       11.430130  Immature
## 7      F. gracilis        8.921832    Mature
## 8      F. gracilis        5.469878  Immature
## 9      F. gracilis        6.626294    Mature
## 10     F. gracilis        7.108676  Immature
## 11      F. gatesii       10.448164    Mature
## 12      F. gatesii        8.719628  Immature
## 13      F. gatesii        8.801543    Mature
## 14      F. gatesii        8.221365  Immature
## 15      F. gatesii        6.888318    Mature
## 16      F. gatesii       11.573826  Immature
## 17      F. gatesii        8.995701    Mature
## 18      F. gatesii        4.066766  Immature
## 19      F. gatesii        9.402712    Mature
## 20      F. gatesii        7.054417  Immature
## 21 F. cylindraceus        5.864353    Mature
## 22 F. cylindraceus        7.564050  Immature
## 23 F. cylindraceus        5.947991    Mature
## 24 F. cylindraceus        6.542218  Immature
## 25 F. cylindraceus        6.749921    Mature
## 26 F. cylindraceus        4.626613  Immature
## 27 F. cylindraceus        9.675574    Mature
## 28 F. cylindraceus        8.306746  Immature
## 29 F. cylindraceus        5.723726    Mature
## 30 F. cylindraceus       10.507630  Immature
```

There are a few extra functions this week to create this dataset. **rep()** repeats values and **rnorm()** creates a normal distribution of values given some mean and standard deviation

This is the first week where we will use the same dataset throughout the whole week. The code to create the dataset will always be in the overview section of that week. If you are following along in your own R environment, make sure the data is loaded for you!

3.1.1 Intro to tidyverse

Introduction

So far in this course we have largely been working in base R. This means we have only been using the set of functions that come preinstalled with R. These functions are absolutely needed and will serve you well throughout your R career.

From here on in the course, we will heavily use the suite of functions from the **tidyverse** package. **tidyverse** is technically multiple packages that have been tweaked to communicate with one another. **tidyverse** includes tools primarily for data manipulation/cleaning and data visualization.

Here are some of the packages used:

- ggplot2 - data visualization
- dplyr - data manipulation
- tidyr - tidy data for consistency
- readr - loading in data
- stringr - manipulating string objects

More info can be found on the tidyverse website: https://www.tidyverse.org/packages/

To install tidyverse we need to use the **install.packages()** function

```r
install.packages('tidyverse')
```

Once a package is installed on your machine, you then need to load it with **library()** to actually use the functions contained within. Make sure the package is installed *before* you try to load it!

```r
library(tidyverse)
```

If all is loaded properly, we can now use all of the functions within tidyverse! From this point on in the course, assume that tidyverse is always loaded. The overview section of each week will reload tidyverse for anyone following along in their own R environment!

3.1.2 Intro to Pipes

Intro to Pipes

A core component of tidyverse are pipes. Pipes allow us to take the output of one function, and input it into a following function. In this way, we can chain together functions to create a pipeline!

Pipes make it much easier to string together multiple functions. Lets take a very simple example.

First lets take the mean of spine length the way we've already learned

```r
mean(cactus_df$spine_length_cm)
```

```
## [1] 8.356677
```

Now lets do the same operation, but this time using a pipe.

```r
cactus_df$spine_length_cm %>%
  mean()
```

```
## [1] 8.356677
```

A pipe is denoted by %>% and is placed after a function. The input for the pipe is the output from the prior function. By default, the pipe places that output automatically into the first argument of the next function. We can explicitly specify where we want to place the output using a period (.) if we so desire. This is useful if we want to pipe the output into a different argument.

For example, this chunk will run the same as the prior chunk:

```r
cactus_df$spine_length_cm%>%
  mean(.)
```

```
## [1] 8.356677
```

Think of a pipe like, well, a pipe.

Some object “flows” into the the start of the pipe and is then passed through each function before ultimately exiting the pipe. The mean example above is rather simple. Lets show a more complex pipe.

In this example, we are filtering the cactus dataset to only include *mature, F. Gracilis* records and then taking the mean of the spine length. We’ll learn about the **filter()** function soon, but, in short, it filters an object based on some criteria.

```
mean(filter(filter(cactus_df, species == "F. gracilis"), lifestage == "Mature")$spine_length_cm)
```

```
## [1] 8.95645
```

While this works, it is rather difficult to read and can be confusing with the different commas, parentheses and quotations. If you make an error here it can be difficult to troubleshoot!

Lets now do the same operation, but instead using pipes! The **pull()** function here pulls the column of data as a vector so that we can then calculate the mean.

```
cactus_df%>%
  filter(species == "F. gracilis")%>%
  filter(lifestage == "Mature")%>%
  pull(spine_length_cm)%>%
  mean()
```

```
## [1] 8.95645
```

This pipeline is much easier to understand. We know the object is first filtered by species, then by lifestage. Then it pulls out the *spine_length_cm* column and calculates the mean.

Notice that we also do not need to call the column name using the $ symbol in our functions! Within a pipeline, we can just use the column names from the input object.

If we want to save the output from the pipeline, we just need to assign it like we would any other object. While we can do it at the start of the pipe, I find it is more intuitive to do it at the end using a -> arrow.

```
#Either of these will save the output of the pipeline
#to an mean_mature_F_gracilis_spine_cm object

cactus_df%>%
  filter(species == "F. gracilis")%>%
  filter(lifestage == "Mature")%>%
  pull(spine_length_cm)%>%
  sum() -> mean_mature_F_gracilis_spine_cm

mean_mature_F_gracilis_spine_cm <- cactus_df%>%
  filter(species == "F. gracilis")%>%
  filter(lifestage == "Mature")%>%
  pull(spine_length_cm)%>%
  sum()
```

The pipe is a core part of the tidyverse package. From this point forward, this course will almost always use pipes except in the case of very simple functions!

3.2.1 select and pull

---

select() and pull()

Tidyverse makes it easy to pull data from objects. Here we will learn of the 2 most common functions for this task, **select()** and **pull()**

select() **select()** allows you to grab specific columns from an input dataset.

```
cactus_df%>%
  select(species)
```

```
##            species
## 1      F. gracilis
## 2      F. gracilis
## 3      F. gracilis
## 4      F. gracilis
## 5      F. gracilis
## 6      F. gracilis
## 7      F. gracilis
## 8      F. gracilis
## 9      F. gracilis
## 10     F. gracilis
## 11      F. gatesii
## 12      F. gatesii
## 13      F. gatesii
## 14      F. gatesii
## 15      F. gatesii
## 16      F. gatesii
## 17      F. gatesii
## 18      F. gatesii
## 19      F. gatesii
## 20      F. gatesii
## 21 F. cylindraceus
## 22 F. cylindraceus
## 23 F. cylindraceus
## 24 F. cylindraceus
## 25 F. cylindraceus
## 26 F. cylindraceus
## 27 F. cylindraceus
## 28 F. cylindraceus
## 29 F. cylindraceus
## 30 F. cylindraceus
```

You can select multiple columns by separating them with a comma. As you can see, **select()** returns a data object containing all of the columns.

```
cactus_df%>%
  select(spine_length_cm,species)
```

```
##    spine_length_cm         species
## 1         8.759279     F. gracilis
## 2         6.995353     F. gracilis
## 3         7.333585     F. gracilis
## 4         5.962849     F. gracilis
## 5         5.856418     F. gracilis
## 6         8.607057     F. gracilis
## 7         8.896420     F. gracilis
## 8         8.106008     F. gracilis
## 9         9.844535     F. gracilis
## 10       12.100169     F. gracilis
## 11        7.017938      F. gatesii
## 12        3.381662      F. gatesii
## 13       10.011477      F. gatesii
## 14        6.581598      F. gatesii
## 15        6.623983      F. gatesii
## 16       10.051143      F. gatesii
## 17        7.430454      F. gatesii
## 18        5.558565      F. gatesii
## 19        8.362607      F. gatesii
## 20        7.722217      F. gatesii
## 21        8.011528 F. cylindraceus
## 22        8.770561 F. cylindraceus
## 23        7.258680 F. cylindraceus
## 24        9.288753 F. cylindraceus
## 25        7.559027 F. cylindraceus
## 26        8.663564 F. cylindraceus
## 27       10.193678 F. cylindraceus
## 28        8.870363 F. cylindraceus
## 29        7.348137 F. cylindraceus
## 30       10.297615 F. cylindraceus
```

pull() **pull()** is very similar to **select()** in that it pulls data from an object. The main difference is that **pull()** returns a vector instead of a dataframe.

```
cactus_df%>%
  pull(spine_length_cm)
```

```
##  [1]  8.759279  6.995353  7.333585  5.962849  5.856418  8.607057  8.896420
##  [8]  8.106008  9.844535 12.100169  7.017938  3.381662 10.011477  6.581598
## [15]  6.623983 10.051143  7.430454  5.558565  8.362607  7.722217  8.011528
## [22]  8.770561  7.258680  9.288753  7.559027  8.663564 10.193678  8.870363
## [29]  7.348137 10.297615
```

This makes it possible for us to pipe our outputs directly into functions that require a vector input, such as **mean()**.

```
cactus_df%>%
  pull(spine_length_cm)%>%
  mean()
```

```
## [1] 8.048841
```

3.2.2 arrange and relocate

---

arrange() and relocate()

A common data cleaning step is sorting by values and rearranging the order of columns. The **arrange()** and **relocate()** functions will accomplish each of those tasks!

arrange() **arrange()** allows us to sort our data according to one of our columns. The default is by ascending order.

```
cactus_df%>%
  arrange(spine_length_cm)
```

```
##             species spine_length_cm lifestage
## 1        F. gatesii        4.664116  Immature
## 2  F. cylindraceus        4.764235    Mature
## 3       F. gracilis        5.947158  Immature
## 4  F. cylindraceus        5.951742  Immature
## 5        F. gatesii        6.096763    Mature
## 6  F. cylindraceus        6.300591    Mature
## 7        F. gatesii        6.430191    Mature
## 8        F. gatesii        6.579187    Mature
## 9  F. cylindraceus        6.718588  Immature
## 10      F. gracilis        6.744188  Immature
## 11      F. gracilis        6.799481  Immature
## 12 F. cylindraceus        6.849306    Mature
## 13       F. gatesii        7.239547    Mature
## 14       F. gatesii        7.304915  Immature
## 15       F. gatesii        7.506616    Mature
## 16      F. gracilis        7.528599    Mature
## 17 F. cylindraceus        7.888876  Immature
## 18       F. gatesii        7.909945  Immature
## 19 F. cylindraceus        8.211352    Mature
## 20      F. gracilis        8.477463    Mature
## 21       F. gatesii        8.513767  Immature
## 22 F. cylindraceus        8.602307  Immature
## 23 F. cylindraceus        9.038814    Mature
## 24      F. gracilis        9.096794  Immature
## 25 F. cylindraceus        9.215929  Immature
## 26       F. gatesii        9.837993  Immature
## 27      F. gracilis        9.987008    Mature
## 28      F. gracilis       10.721305    Mature
## 29      F. gracilis       11.065221  Immature
## 30      F. gracilis       12.374666    Mature
```

If we want to sort in descending order, we need to use the **desc()** function within **arrange()**

```
cactus_df%>%
  arrange(desc(spine_length_cm))
```

```
##            species spine_length_cm lifestage
## 1      F. gracilis       12.374666    Mature
## 2      F. gracilis       11.065221  Immature
## 3      F. gracilis       10.721305    Mature
## 4      F. gracilis        9.987008    Mature
## 5       F. gatesii        9.837993  Immature
## 6  F. cylindraceus        9.215929  Immature
## 7      F. gracilis        9.096794  Immature
## 8  F. cylindraceus        9.038814    Mature
## 9  F. cylindraceus        8.602307  Immature
## 10      F. gatesii        8.513767  Immature
## 11     F. gracilis        8.477463    Mature
## 12 F. cylindraceus        8.211352    Mature
## 13      F. gatesii        7.909945  Immature
## 14 F. cylindraceus        7.888876  Immature
## 15     F. gracilis        7.528599    Mature
## 16      F. gatesii        7.506616    Mature
## 17      F. gatesii        7.304915  Immature
## 18      F. gatesii        7.239547    Mature
## 19 F. cylindraceus        6.849306    Mature
## 20     F. gracilis        6.799481  Immature
## 21     F. gracilis        6.744188  Immature
## 22 F. cylindraceus        6.718588  Immature
## 23      F. gatesii        6.579187    Mature
## 24      F. gatesii        6.430191    Mature
## 25 F. cylindraceus        6.300591    Mature
## 26      F. gatesii        6.096763    Mature
## 27 F. cylindraceus        5.951742  Immature
## 28     F. gracilis        5.947158  Immature
## 29 F. cylindraceus        4.764235    Mature
## 30      F. gatesii        4.664116  Immature
```

relocate() The **relocate()** function uses the same format as **select()** but reorders columns without removing any columns. Whatever is inputted into the function appears as the first set of columns. You do not need to input all columns.

```
cactus_df %>%
  relocate(spine_length_cm)
```

```
##    spine_length_cm         species lifestage
## 1         9.987008     F. gracilis    Mature
## 2         9.096794     F. gracilis  Immature
## 3         8.477463     F. gracilis    Mature
## 4         6.744188     F. gracilis  Immature
## 5        10.721305     F. gracilis    Mature
## 6         6.799481     F. gracilis  Immature
## 7        12.374666     F. gracilis    Mature
## 8        11.065221     F. gracilis  Immature
## 9         7.528599     F. gracilis    Mature
## 10        5.947158     F. gracilis  Immature
## 11        6.579187      F. gatesii    Mature
## 12        8.513767      F. gatesii  Immature
## 13        7.506616      F. gatesii    Mature
## 14        7.304915      F. gatesii  Immature
## 15        6.096763      F. gatesii    Mature
## 16        7.909945      F. gatesii  Immature
## 17        6.430191      F. gatesii    Mature
## 18        4.664116      F. gatesii  Immature
## 19        7.239547      F. gatesii    Mature
## 20        9.837993      F. gatesii  Immature
## 21        6.849306 F. cylindraceus    Mature
## 22        9.215929 F. cylindraceus  Immature
## 23        4.764235 F. cylindraceus    Mature
## 24        7.888876 F. cylindraceus  Immature
## 25        9.038814 F. cylindraceus    Mature
## 26        8.602307 F. cylindraceus  Immature
## 27        8.211352 F. cylindraceus    Mature
## 28        6.718588 F. cylindraceus  Immature
## 29        6.300591 F. cylindraceus    Mature
## 30        5.951742 F. cylindraceus  Immature
```

3.2.3 filter and distinct

Introduction to filter() and distinct()

Removing problematic data is a must for data analysis. This section will cover how to filter records based on some criteria using **filter()** and how to identify unique records using **distinct()**

filter() **filter()** allows us to filter columns based on some criteria. The **filter()** function utilizes the same logical statements that we learned about in Section 2 .

```
cactus_df%>%
  filter(species == "F. gracilis")
```

```
##        species spine_length_cm lifestage
## 1  F. gracilis        8.235293    Mature
## 2  F. gracilis        6.105051  Immature
## 3  F. gracilis        7.018885    Mature
## 4  F. gracilis        7.487816  Immature
## 5  F. gracilis       11.687724    Mature
## 6  F. gracilis        6.696100  Immature
## 7  F. gracilis        8.470773    Mature
## 8  F. gracilis        8.155922  Immature
## 9  F. gracilis        6.076287    Mature
## 10 F. gracilis        7.857384  Immature
```

We can chain multiple filters together by separating them with a comma.

```
cactus_df %>%
  filter(species == "F. gracilis" , lifestage == "Mature")
```

```
##       species spine_length_cm lifestage
## 1 F. gracilis        8.235293    Mature
## 2 F. gracilis        7.018885    Mature
## 3 F. gracilis       11.687724    Mature
## 4 F. gracilis        8.470773    Mature
## 5 F. gracilis        6.076287    Mature
```

Using a comma is the equivalent of saying "I only want rows that equal my first criteria AND my second criteria". If we want to find records using OR, we would use the | symbol.

```
cactus_df %>%
  filter(species == "F. gracilis" | lifestage == "Mature")
```

```
##            species spine_length_cm lifestage
## 1      F. gracilis        8.235293    Mature
## 2      F. gracilis        6.105051  Immature
## 3      F. gracilis        7.018885    Mature
## 4      F. gracilis        7.487816  Immature
## 5      F. gracilis       11.687724    Mature
## 6      F. gracilis        6.696100  Immature
## 7      F. gracilis        8.470773    Mature
## 8      F. gracilis        8.155922  Immature
## 9      F. gracilis        6.076287    Mature
## 10     F. gracilis        7.857384  Immature
## 11      F. gatesii       10.889102    Mature
## 12      F. gatesii        8.082466    Mature
## 13      F. gatesii        3.893506    Mature
## 14      F. gatesii        5.078720    Mature
## 15      F. gatesii       11.818207    Mature
## 16 F. cylindraceus        9.403569    Mature
## 17 F. cylindraceus        4.855712    Mature
## 18 F. cylindraceus        4.796928    Mature
## 19 F. cylindraceus        5.076489    Mature
## 20 F. cylindraceus       12.200218    Mature
```

distinct() **distinct()** finds unique rows of data. Lets select just the species column and see what distinct species we have.

```
cactus_df%>%
  select(species)%>%
  distinct()
```

```
##           species
## 1     F. gracilis
## 2      F. gatesii
## 3 F. cylindraceus
```

Cool! 3 species! If we run distinct over multiple columns, we can find all unique arrangements of those data! For example, what distinct combinations of species and lifestage do we have?

```
cactus_df%>%
  select(species,lifestage)%>%
  distinct()
```

```
##           species lifestage
## 1     F. gracilis    Mature
## 2     F. gracilis  Immature
## 3      F. gatesii    Mature
## 4      F. gatesii  Immature
## 5 F. cylindraceus    Mature
## 6 F. cylindraceus  Immature
```

3.3.1 mutate and count

Introduction to mutate() and count()

mutate() **mutate()** allows us to add a new column to the end of the dataset or to alter data already present. This could be as simple as adding an ID column using **row_number().** or converting centimeters to millimeters for example. We specify the name of the new column first, and then what information that column contains.

If we use a column name already in the data, **mutate()** will overwrite the data!

```
cactus_df%>%
  mutate(ID = row_number())
```

```
##            species spine_length_cm lifestage ID
## 1      F. gracilis        9.575478    Mature  1
## 2      F. gracilis        9.538084  Immature  2
## 3      F. gracilis        8.664405    Mature  3
## 4      F. gracilis        5.983247  Immature  4
## 5      F. gracilis        7.761095    Mature  5
## 6      F. gracilis        7.439209  Immature  6
## 7      F. gracilis        9.125979    Mature  7
## 8      F. gracilis        7.255122  Immature  8
## 9      F. gracilis        9.953947    Mature  9
## 10     F. gracilis        7.250838  Immature 10
## 11      F. gatesii       10.105423    Mature 11
## 12      F. gatesii        5.901646  Immature 12
## 13      F. gatesii        5.479690    Mature 13
## 14      F. gatesii       14.482080  Immature 14
## 15      F. gatesii        7.166285    Mature 15
## 16      F. gatesii        8.596455  Immature 16
## 17      F. gatesii        9.273139    Mature 17
## 18      F. gatesii        7.032439  Immature 18
## 19      F. gatesii        9.033724    Mature 19
## 20      F. gatesii        8.737929  Immature 20
## 21 F. cylindraceus        7.569239    Mature 21
## 22 F. cylindraceus        8.130586  Immature 22
## 23 F. cylindraceus        7.931865    Mature 23
## 24 F. cylindraceus       12.256904  Immature 24
## 25 F. cylindraceus        6.517328    Mature 25
## 26 F. cylindraceus        5.808007  Immature 26
## 27 F. cylindraceus        8.075577    Mature 27
## 28 F. cylindraceus        8.620961  Immature 28
## 29 F. cylindraceus        8.873047    Mature 29
## 30 F. cylindraceus        7.083269  Immature 30
```

mutate() can also be used with simple calculations. For example, converting cm to mm.

```
cactus_df%>%
  mutate(spine_length_mm = spine_length_cm*10)
```

```
##            species spine_length_cm lifestage spine_length_mm
## 1      F. gracilis        9.575478    Mature        95.75478
## 2      F. gracilis        9.538084  Immature        95.38084
## 3      F. gracilis        8.664405    Mature        86.64405
## 4      F. gracilis        5.983247  Immature        59.83247
## 5      F. gracilis        7.761095    Mature        77.61095
## 6      F. gracilis        7.439209  Immature        74.39209
## 7      F. gracilis        9.125979    Mature        91.25979
## 8      F. gracilis        7.255122  Immature        72.55122
## 9      F. gracilis        9.953947    Mature        99.53947
## 10     F. gracilis        7.250838  Immature        72.50838
## 11      F. gatesii       10.105423    Mature       101.05423
## 12      F. gatesii        5.901646  Immature        59.01646
## 13      F. gatesii        5.479690    Mature        54.79690
## 14      F. gatesii       14.482080  Immature       144.82080
## 15      F. gatesii        7.166285    Mature        71.66285
## 16      F. gatesii        8.596455  Immature        85.96455
## 17      F. gatesii        9.273139    Mature        92.73139
## 18      F. gatesii        7.032439  Immature        70.32439
## 19      F. gatesii        9.033724    Mature        90.33724
## 20      F. gatesii        8.737929  Immature        87.37929
## 21 F. cylindraceus        7.569239    Mature        75.69239
## 22 F. cylindraceus        8.130586  Immature        81.30586
## 23 F. cylindraceus        7.931865    Mature        79.31865
## 24 F. cylindraceus       12.256904  Immature       122.56904
## 25 F. cylindraceus        6.517328    Mature        65.17328
## 26 F. cylindraceus        5.808007  Immature        58.08007
## 27 F. cylindraceus        8.075577    Mature        80.75577
## 28 F. cylindraceus        8.620961  Immature        86.20961
## 29 F. cylindraceus        8.873047    Mature        88.73047
## 30 F. cylindraceus        7.083269  Immature        70.83269
```

mutate() is one of the most often used functions in tidyverse despite its simple appearance!

count() A very common data analysis practice is to count the number of times a particular value occurs in a dataset. **count()** will accomplish this task.

For example, counting the number of records for each species and each lifestage. The sort argument will sort them from largest to smallest

```
cactus_df%>%
  count(species, lifestage, sort = TRUE)
```

```
##           species lifestage n
## 1 F. cylindraceus  Immature 5
## 2 F. cylindraceus    Mature 5
## 3      F. gatesii  Immature 5
## 4      F. gatesii    Mature 5
## 5     F. gracilis  Immature 5
## 6     F. gracilis    Mature 5
```

We can also count logical statements! For example, how many spine lengths are longer than 10 cm?

```
cactus_df%>%
  count(spine_length_cm > 10)
```

```
##   spine_length_cm > 10  n
## 1                FALSE 27
## 2                 TRUE  3
```

3.3.2 summarize and group_by

Intro to summarize() and group_by()

summarize() and group_by() **summarize()** is very similar to mutate in that it summarizes the data in some way. In fact, if we run the same code from the mutate section using **summarize()** we will get the same output.

```
cactus_df%>%
  summarize(spine_length_mm = spine_length_cm*10)
```

```
##    spine_length_mm
## 1         58.73348
## 2        105.26370
## 3         73.00699
## 4         62.68974
## 5         75.27441
## 6         76.05648
## 7        102.19841
## 8         81.69475
## 9         95.08108
## 10        70.01416
## 11        84.28891
## 12        73.50628
## 13        81.89167
## 14        62.09273
## 15        53.78397
## 16       119.94427
## 17        92.01418
## 18        54.97457
## 19        67.77668
## 20        56.29040
## 21       123.97621
## 22       106.24826
## 23        74.69710
## 24        90.86388
## 25        71.71320
## 26        70.47506
## 27        64.22794
## 28        68.10765
## 29       113.01815
## 30        78.91944
```

The difference is that **summarize()** does not add the output to the input dataset. This may seem less useful as first, but in reality, we use **summarize()** often with the next function **group_by()**

group_by() **group_by()** will split your data into groups based on unique values. This example will group our data based on each of our 3 species.

```
cactus_df%>%
  group_by(species)
```

```
## # A tibble: 30 x 3
## # Groups:   species [3]
##    species     spine_length_cm lifestage
##    <chr>                 <dbl> <chr>
##  1 F. gracilis            5.87 Mature
##  2 F. gracilis           10.5  Immature
##  3 F. gracilis            7.30 Mature
##  4 F. gracilis            6.27 Immature
##  5 F. gracilis            7.53 Mature
##  6 F. gracilis            7.61 Immature
##  7 F. gracilis           10.2  Mature
##  8 F. gracilis            8.17 Immature
##  9 F. gracilis            9.51 Mature
## 10 F. gracilis            7.00 Immature
## # ... with 20 more rows
```

It may seem like not much has happened, but now lets reintroduce **summarize()** after the **group_by()** function to get the mean spine length.

```
cactus_df%>%
  group_by(species)%>%
  summarize(mean_spine_length_cm = mean(spine_length_cm))
```

```
## # A tibble: 3 x 2
##   species         mean_spine_length_cm
##   <chr>                          <dbl>
## 1 F. cylindraceus                 8.62
## 2 F. gatesii                      7.47
## 3 F. gracilis                     8.00
```

We now have 3 outputs, one for each group passed from the **group_by()** function!

We can group by multiple criteria as well! Simply separate each grouping variable by a comma.

```
cactus_df%>%
  group_by(species, lifestage)%>%
  summarize(mean_spine_length_cm = mean(spine_length_cm))
```

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

```
## # A tibble: 6 x 3
## # Groups:   species [3]
##   species         lifestage mean_spine_length_cm
##   <chr>           <chr>                    <dbl>
## 1 F. cylindraceus Immature                  8.29
## 2 F. cylindraceus Mature                    8.95
## 3 F. gatesii      Immature                  7.34
## 4 F. gatesii      Mature                    7.60
## 5 F. gracilis     Immature                  7.91
## 6 F. gracilis     Mature                    8.09
```

group_by() with **summarize()** is perhaps the most useful pairing of functions in tidyverse!

Section 4 - Cleaning data and creating pipelines

4.0 Section 4 Overview

Intro to data cleaning

This week we are going to start actually applying the skills we have learned throughout this course.

When working with any data set, it needs to be cleaned. Cleaning involves standardizing data, removing problematic records, and ensuring that your data is as accurate as possible. We will cover topics such as:

- Identifying potential problems in datasets
- Fixing a variety of common issues
- Creating a pipeline to clean data

Creating our data and loading tidyverse Our data set for this week involves collecting genetic samples from endangered guppies.

- survey_date - Dates the sample was taken
- gene_id - a unique identifier for each gene sample taken
- location - An informal location where the data was taken
- lat - the latitude in decimal degrees
- lon - the longitude in decimal degrees
- length_mm - the length of the fish in millimeters

This data set has very intentional errors that we will identify throughout this week!

```r
library(tidyverse)

guppy_genes_df <- data.frame(
  'survey_date' = c('7/27/2020','7/27/2020','7/27/2020','7/28/2020','7/28/2020',
                    '7/30/2020','7/30/2020','7/30/2020','7/31/2020','8/1/2020',
                    '8/1/2020','8/5/2020','8/50/2020','8/5/2020','8/7/2020'),

  "gene_id" = c('FMMJ_GUP_1924','FMMJ_GUP_1924','AHTD_GUP_5866','POIU_GUP_5241',
                'TT_G_4','GRRA_GUP_5693','AGTH_GUP_1540',NA,'ASDD_GUP_3596',
                'SDAD_GUP_3114','THTE_GUP_3944','SSAS_GUP_1456','AASD_GUP_1217',
                'JNKT_GUP_4566',NA),

  'location' = c("UPPER_STREAM","UPPER_STREAM","UPER_STREAM","UPPER_STREAM",
                 "UPPER_STREAM","UPER_STREAM","UPPER_STREAM","UPPER_STREAM",
                 "UPPER_STREAM","LOWER_STREAM","LOWER_STREAM","LOWER_STREAM",
                 "LOWER_STREAM","LOWER_STREAM","LOWER_STREAM"),

  'lat' = c(17.124975,17.124975,17.124634,1.7124844,17.124049,17.124028,
            17.124837,17.124910,17.124526,17.125321,17.125925,17.124701,
            17.125541,17.125032,17.124425),
  'lon' = c('-88.124909','-88.124909','-88.125183','-88.125748','-88.125883',
            '-88.124755','-88.124802','-88.124369','-88.125922','-88.124812',
            '-88.125675','-88.124641','-88.124052','-88.124394','-88.125226'),

  'length_mm' = c(25,25,19,23,20,22,22,24,230,37,35,230,32,32,40))
```

4.1.1 Best Practices for Data Cleaning

Best practices for cleaning data

When it comes to cleaning data, what are best practices?

Really, cleaning data is more of an art than a science. There are many philosophies, methods, and ways in which we can clean a data set. There is no one-size-fits all approach.

However, below are a few tips that should be taken into account when cleaning ANY dataset.

Backup your data You should always create a backup BEFORE you start cleaning it. A backup will ensure that you will always have an original copy to work off of in case of errors. I would recommend making a backup for each stage of your data cleaning process.

Keep notes When you clean your data, keep detailed notes of what you changed and why you changed it. RMarkdown documents, a word document, or even a notes column on your spreadsheet are all valid options based on your specific goals. Keep these short and accurate. Remember, these notes are for **you** in the future!

Fix at the source In this course, I show you how to fix your data in R. However, it is often best to fix your data at the source **BEFORE** you input it into R. This typically means updating the spreadsheet in excel or some other spreadsheet software manually.

That being said, using R to **identify** potential issues while then fixing them in your spreadsheet software is a highly effective strategy!

Make your data consistent Consistent data is incredibly important. Checking for consistency ensures your data is accurate, the data class of each column is correct, and that errors are fixed before you analyze.

A few examples for consistent biological data include:

- Taxonomy naming conventions (Dryophytes cinerea vs D. cinerea vs D_cinerea)
- Missing data format (Null vs NA vs N/A)
- Dates (July 27, 1996 vs 27/7/1996 vs 7/27/1996)
- Coordinates (S17.1117 vs -17.1117 vs 17 6' 42.1194")
- Units and conversions (100 F vs 37.778 C)

Trust YOUR knowledge of YOUR data When cleaning data you will often have to make judgement calls. A few examples of judgement calls include determining which taxonomy is "correct", which format you keep your data in, or determining the true meaning of a typo. These judgement calls can be difficult to make, but you understand your study better than anyone else.

However, lean towards excluding problematic data rather than including it. Just like people! If you aren't sure which value is correct, you can always convert it to an NA value and include a note for yourself. This is also where backups come in handy to preserve the original entries!

4.1.2 glimpse and simple plots

Identifying issues with glimpse() and simple plots

Before we clean and manipulate the data, we should explore the data. Exploring the data at this stage allows us to get a "feel" for what potential issues there may be.

glimpse() Lets understand what our data looks like. You could go through it manually, however, tidyverse offers a really nice function through dplyr called **glimpse().**

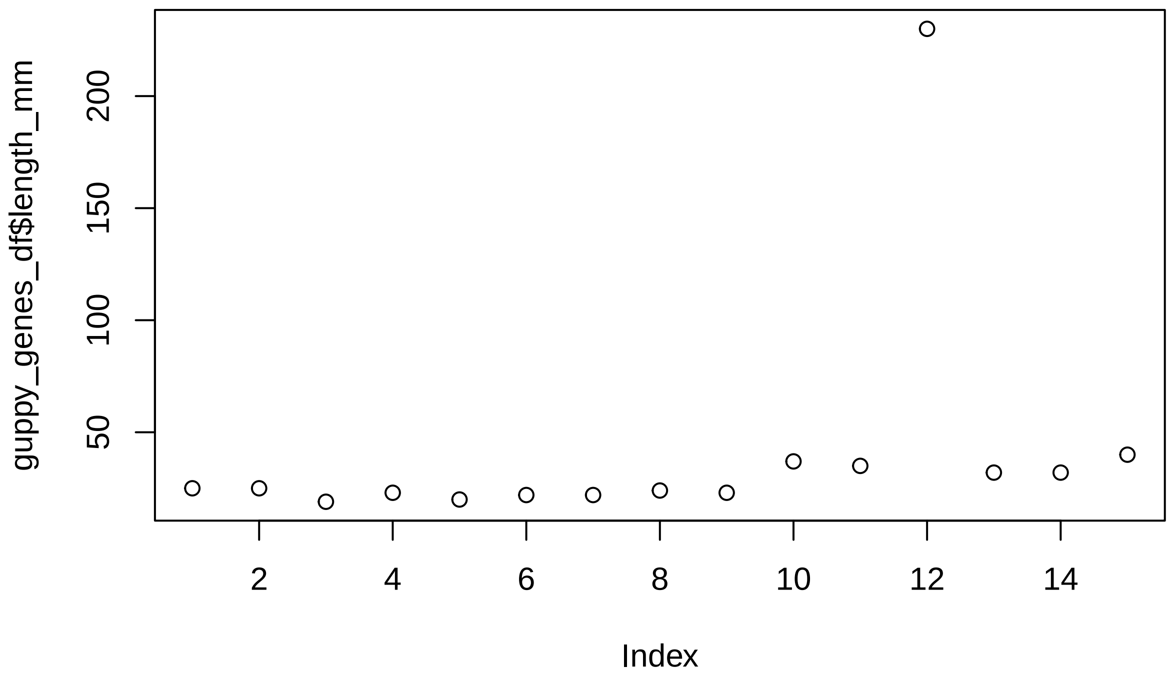
```
guppy_genes_df%>%
  glimpse()
```

```
## Rows: 15
## Columns: 6
## $ survey_date <chr> "7/27/2020", "7/27/2020", "7/27/2020", "7/28/2020", "7/28/~
## $ gene_id     <chr> "FMMJ_GUP_1924", "FMMJ_GUP_1924", "AHTD_GUP_5866", "POIU_G~
## $ location    <chr> "UPPER_STREAM", "UPPER_STREAM", "UPER_STREAM", "UPPER_STRE~
## $ lat         <dbl> 17.124975, 17.124975, 17.124634, 1.712484, 17.124049, 17.1~
## $ lon         <chr> "-88.124909", "-88.124909", "-88.125183", "-88.125748", "-~
## $ length_mm   <dbl> 25, 25, 19, 23, 20, 22, 22, 24, 23, 37, 35, 230, 32, 32, 40
```

glimpse() tells us the names of the columns, provides a preview of the data, and highlights the data class of each column. From here we can see that our lat and lon columns are actually different data types (lat is numeric, lon in character). This could be an issue in downstream analyses!

Simple plots reveal a lot With numerical data, sometimes a simple plot of a single variable can very quickly show you issues.

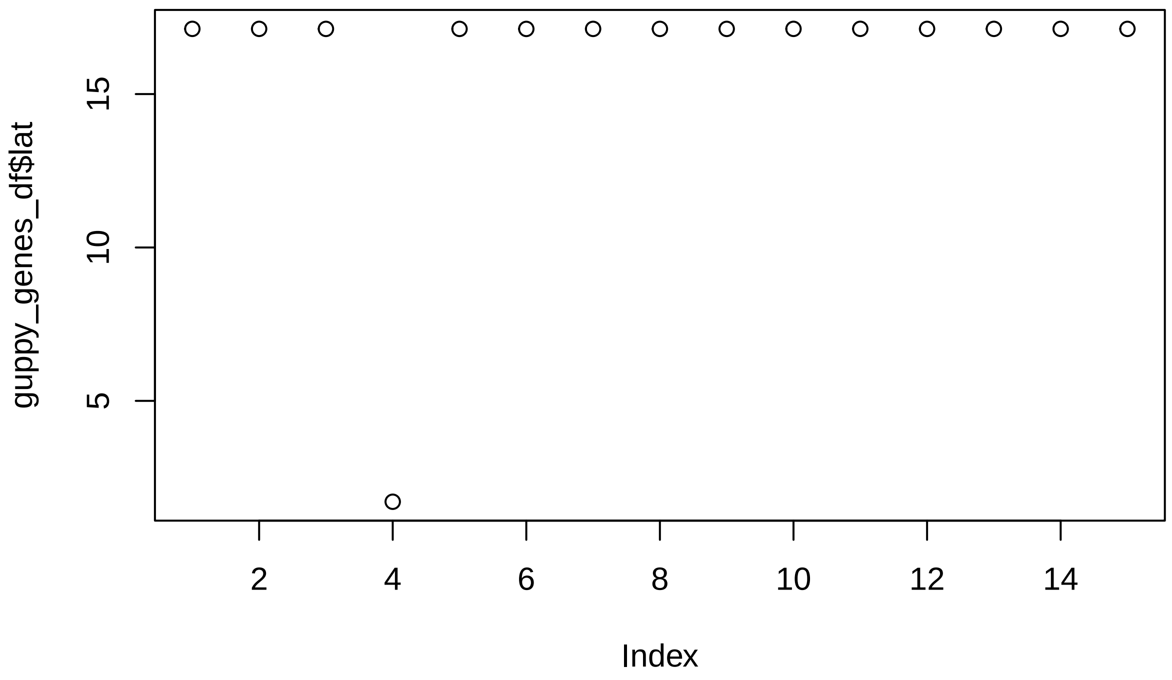
```
plot(guppy_genes_df$length_mm)
```



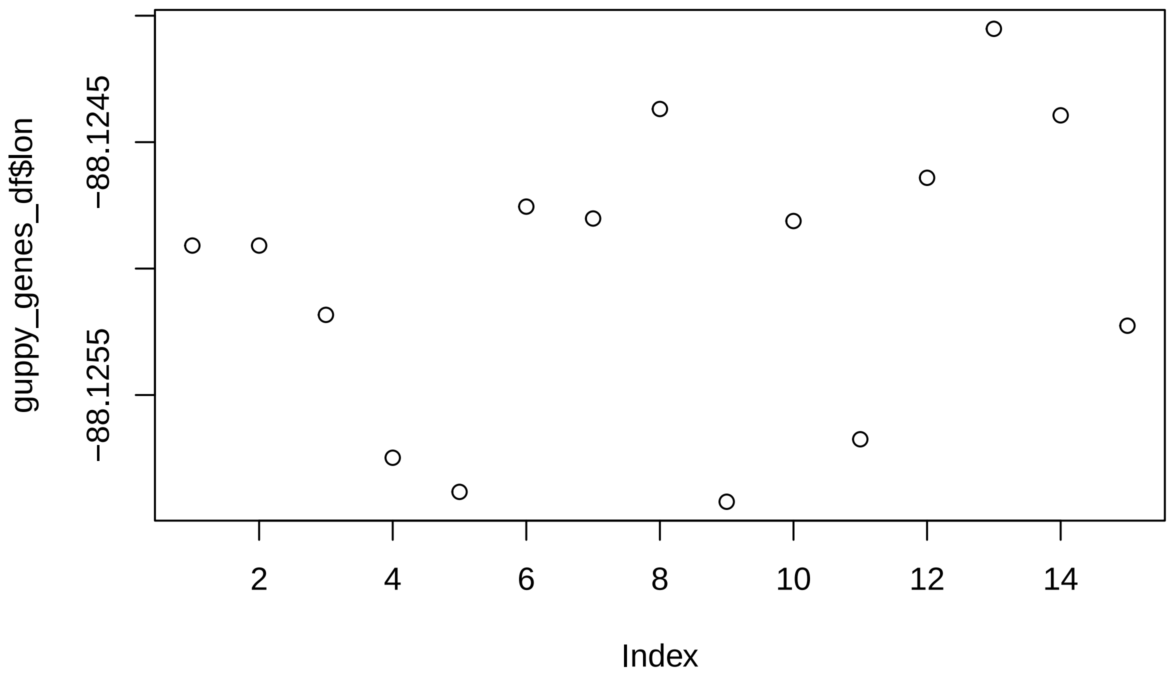
From these plots we identify potential problems through a visual check. It looks like we have an outlier at index 12 that needs to be addressed at some point.

Lets also just plot our lat and lon columns.

```
plot(guppy_genes_df$lat)
```



```
plot(guppy_genes_df$lon)
```



Again, more weird data. In the lat column it looks like the value at index 4 is an issue. For reference, the lon data looks fairly normal. These values may be correct, but its always a good idea to look at any weird points in your data. Often, these are simple data entry issues than can be easily resolved!

4.1.3 unique and Na values

Identifying issues using unique() and is.na()

This section will continue identifying issues in our data using 2 simple functions: **unique()** and **is.na()**.

unique() A great method for identifying typos, synonym data, or inconsistent data entry is by seeing all unique values in a column. We can easily do this with **unique()**.

Lets look for unique values in survey_date. Again, **pull()** pulls the column that we want to feed into unique.

```
guppy_genes_df%>%
  pull(survey_date)%>%
  unique()
```

```
## [1] "7/27/2020" "7/28/2020" "7/30/2020" "7/31/2020" "8/1/2020"  "8/5/2020"
## [7] "8/50/2020" "8/7/2020"
```

It looks like we have an issues with some of our dates. One of them reads as 8/50/2020 which is impossible.

Lets check our location column as well.

```
guppy_genes_df%>%
  pull(location)%>%
  unique()
```

```
## [1] "UPPER_STREAM" "UPER_STREAM"  "LOWER_STREAM"
```

It seems like we have a few typos in the data here as well!

It should be noted that unique will not tell you how many typos there are nor will it reveal where in the data the typo exists. We'll have to investigate our data further to uncover these typos!

is.na() Searching for NA values is always important. NA values typically indicate missing data or import issues. Luckily, we can identify them with **is.na()**.

Using **is.na()** on the entire dataset will search everywhere for NA values.

```
is.na(guppy_genes_df)
```

```
##       survey_date gene_id location   lat   lon length_mm
##  [1,]       FALSE   FALSE    FALSE FALSE FALSE     FALSE
##  [2,]       FALSE   FALSE    FALSE FALSE FALSE     FALSE
##  [3,]       FALSE   FALSE    FALSE FALSE FALSE     FALSE
##  [4,]       FALSE   FALSE    FALSE FALSE FALSE     FALSE
##  [5,]       FALSE   FALSE    FALSE FALSE FALSE     FALSE
##  [6,]       FALSE   FALSE    FALSE FALSE FALSE     FALSE
##  [7,]       FALSE   FALSE    FALSE FALSE FALSE     FALSE
##  [8,]       FALSE    TRUE    FALSE FALSE FALSE     FALSE
##  [9,]       FALSE   FALSE    FALSE FALSE FALSE     FALSE
## [10,]       FALSE   FALSE    FALSE FALSE FALSE     FALSE
## [11,]       FALSE   FALSE    FALSE FALSE FALSE     FALSE
## [12,]       FALSE   FALSE    FALSE FALSE FALSE     FALSE
## [13,]       FALSE   FALSE    FALSE FALSE FALSE     FALSE
## [14,]       FALSE   FALSE    FALSE FALSE FALSE     FALSE
## [15,]       FALSE    TRUE    FALSE FALSE FALSE     FALSE
```

Alternatively you can use it on individual columns.

```
is.na(guppy_genes_df$gene_id)
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE  TRUE
```

TRUE values indicate where NA values are. We can see that our NA values are located in gene_ID (at index position 8 and 15).

When searching for NA values we need to be aware that **is.na()** only finds NA. Many datasets will use different notations to indicate NA values such as NULL or N/A. Make sure you look for these values as well! **unique()** will often highlight those values.

4.2.1 changing data types

Correcting data types We can easily change the data class of an object using the *as.~* suite of functions. These include, **as.numeric(), as.character(), as.data.frame()** etc. etc.

We identified that our lon column is a character class when we used **glimpse()** to look at the data.

```
glimpse(guppy_genes_df)
```

```
## Rows: 15
## Columns: 6
## $ survey_date <chr> "7/27/2020", "7/27/2020", "7/27/2020", "7/28/2020", "7/28/~
## $ gene_id     <chr> "FMMJ_GUP_1924", "FMMJ_GUP_1924", "AHTD_GUP_5866", "POIU_G~
## $ location    <chr> "UPPER_STREAM", "UPPER_STREAM", "UPER_STREAM", "UPPER_STRE~
## $ lat         <dbl> 17.124975, 17.124975, 17.124634, 1.712484, 17.124049, 17.1~
## $ lon         <chr> "-88.124909", "-88.124909", "-88.125183", "-88.125748", "-~
## $ length_mm   <dbl> 25, 25, 19, 23, 20, 22, 22, 24, 23, 37, 35, 230, 32, 32, 40
```

Lets fix that using **mutate()** and **as.numeric()**! **as.numeric()** will simply convert the data in that column into a numeric value.

```
guppy_genes_df%>%
  mutate(lon = as.numeric(lon))
```

```
##    survey_date       gene_id     location       lat       lon length_mm
## 1    7/27/2020 FMMJ_GUP_1924 UPPER_STREAM 17.124975 -88.12491        25
## 2    7/27/2020 FMMJ_GUP_1924 UPPER_STREAM 17.124975 -88.12491        25
## 3    7/27/2020 AHTD_GUP_5866  UPER_STREAM 17.124634 -88.12518        19
## 4    7/28/2020 POIU_GUP_5241 UPPER_STREAM  1.712484 -88.12575        23
## 5    7/28/2020         TT_G_4 UPPER_STREAM 17.124049 -88.12588        20
## 6    7/30/2020 GRRA_GUP_5693  UPER_STREAM 17.124028 -88.12475        22
## 7    7/30/2020 AGTH_GUP_1540 UPPER_STREAM 17.124837 -88.12480        22
## 8    7/30/2020          <NA> UPPER_STREAM 17.124910 -88.12437        24
## 9    7/31/2020 ASDD_GUP_3596 UPPER_STREAM 17.124526 -88.12592        23
## 10    8/1/2020 SDAD_GUP_3114 LOWER_STREAM 17.125321 -88.12481        37
## 11    8/1/2020 THTE_GUP_3944 LOWER_STREAM 17.125925 -88.12568        35
## 12    8/5/2020 SSAS_GUP_1456 LOWER_STREAM 17.124701 -88.12464       230
## 13   8/50/2020 AASD_GUP_1217 LOWER_STREAM 17.125541 -88.12405        32
## 14    8/5/2020 JNKT_GUP_4566 LOWER_STREAM 17.125032 -88.12439        32
## 15    8/7/2020          <NA> LOWER_STREAM 17.124425 -88.12523        40
```

In the output, we can see that lon is now a double, a type of numeric!

Changing dates It also appears that our survey_date column is a character. This may pose no problem for your analysis, however there are situations where you may wish for it to be in the Date format.

as.Date() will convert the character data to date format, however it requires you to input the format the date is in. This is because there are many different standards for order (day/month/year vs month/day/year), format (27 Jul. 1996 vs July 27, 1996) and punctuation (7-27-1996 vs 7/27/1996).

Our dates are in the format month/day/Year, this corresponds to an abbreviated format of "%m/%d/%Y".

```
guppy_genes_df%>%
  mutate(survey_date = as.Date(survey_date,format = "%m/%d/%Y"))
```

```
##    survey_date       gene_id     location       lat        lon length_mm
## 1   2020-07-27 FMMJ_GUP_1924 UPPER_STREAM 17.124975 -88.124909        25
## 2   2020-07-27 FMMJ_GUP_1924 UPPER_STREAM 17.124975 -88.124909        25
## 3   2020-07-27 AHTD_GUP_5866  UPER_STREAM 17.124634 -88.125183        19
## 4   2020-07-28 POIU_GUP_5241 UPPER_STREAM  1.712484 -88.125748        23
## 5   2020-07-28        TT_G_4 UPPER_STREAM 17.124049 -88.125883        20
## 6   2020-07-30 GRRA_GUP_5693  UPER_STREAM 17.124028 -88.124755        22
## 7   2020-07-30 AGTH_GUP_1540 UPPER_STREAM 17.124837 -88.124802        22
## 8   2020-07-30          <NA> UPPER_STREAM 17.124910 -88.124369        24
## 9   2020-07-31 ASDD_GUP_3596 UPPER_STREAM 17.124526 -88.125922        23
## 10  2020-08-01 SDAD_GUP_3114 LOWER_STREAM 17.125321 -88.124812        37
## 11  2020-08-01 THTE_GUP_3944 LOWER_STREAM 17.125925 -88.125675        35
## 12  2020-08-05 SSAS_GUP_1456 LOWER_STREAM 17.124701 -88.124641       230
## 13        <NA> AASD_GUP_1217 LOWER_STREAM 17.125541 -88.124052        32
## 14  2020-08-05 JNKT_GUP_4566 LOWER_STREAM 17.125032 -88.124394        32
## 15  2020-08-07          <NA> LOWER_STREAM 17.124425 -88.125226        40
```

We specify the format as a string in the format argument of **as.Date()**. There are a variety of abbreviations that we use in this string to format the date properly.

The help text for **strptime()** (which is used by **as.Date();** use ?strptime to access it) details all of the abbreviations, however the most common ones are listed here:

- %a - Abbreviated weekday name
- %A - Full weekday name in the current locale.
- %b Abbreviated month name
- %B Full month name
- %d Day of the month as decimal number (01–31).
- %e Day of the month as decimal number (1–31), with a leading space for a single-digit number.
- %H Hours as decimal number (00–23).
- %I Hours as decimal number (01–12).
- %m Month as decimal number (01–12).
- %M Minute as decimal number (00–59).
- %p AM/PM indicator in the locale.
- %S Second as integer (00–61)
- %y Year without century (00–99).
- %Y Year with century.

4.2.2 replacing data directly

Fixing issues using R

In the last lesson we identified potential issues with our dataset. And for only 15 observations, its very messy! Lets start fixing these issues!

Just like we talked about in the best practices lesson, it is almost always better to fix the issues at the source. Typically, this will be in excel or some other spreadsheet software. However, this course will show you how to fix issues using R.

Replacing data directly

Remember that our lat column appeared to have some weird data when we plotted it. Lets look at the values and determine whats going on.

```
guppy_genes_df%>%
  select(lat)
```

```
##          lat
## 1  17.124975
## 2  17.124975
## 3  17.124634
## 4   1.712484
## 5  17.124049
## 6  17.124028
## 7  17.124837
## 8  17.124910
## 9  17.124526
## 10 17.125321
## 11 17.125925
## 12 17.124701
## 13 17.125541
## 14 17.125032
## 15 17.124425
```

Here we can see that the 4th lat coordinate appears to have a misplaced decimal. Lets replace the value manually using our subsetting knowledge.

```
guppy_genes_df[4,'lat'] <- 17.12484
```

Remember, we need to use the arrow to assign data to an object. Since we are subsetting the dataset to the left of the arrow, the value 17.12484 will be inputted at that subset, overwriting the data.

Lets check to see if it worked!

```
guppy_genes_df%>%
  select(lat)
```

```
##         lat
## 1  17.12497
## 2  17.12497
## 3  17.12463
## 4  17.12484
## 5  17.12405
## 6  17.12403
## 7  17.12484
## 8  17.12491
## 9  17.12453
## 10 17.12532
## 11 17.12592
## 12 17.12470
## 13 17.12554
## 14 17.12503
## 15 17.12442
```

Looks good to me! This is of course a tedious process if you need to replace many records. This issue would be much easier to solve by fixing it at the source before its in R! (Yes I will keep hammering this point home)

4.3.1 Fixing typos with replace

Fixing data with replace()

Sometimes the simplest way to fix typos is with a simple find and replace. We can accomplish this by using **mutate()** and the function **replace().**

If we remember, the location data has a few typos.

```
guppy_genes_df%>%
  select(location)
```

```
##        location
## 1  UPPER_STREAM
## 2  UPPER_STREAM
## 3   UPER_STREAM
## 4  UPPER_STREAM
## 5  UPPER_STREAM
## 6   UPER_STREAM
## 7  UPPER_STREAM
## 8  UPPER_STREAM
## 9  UPPER_STREAM
## 10 LOWER_STREAM
## 11 LOWER_STREAM
## 12 LOWER_STREAM
## 13 LOWER_STREAM
## 14 LOWER_STREAM
## 15 LOWER_STREAM
```

We could replace each value manually in a dataset this small. However, what if we had 2,000 rows to fix this typo in?

That's where the **replace()** function comes in.

replace() has three arguments:

- x - A vector to find and replace over. In our case, this will be the location column.
- list - A vector of index positions. We can use logic to get these indices where TRUE values indicate where to replace.
- values - The replacement values

Lets see it in action! Remember, we want to use **mutate()** in order to directly update the location column.

```
guppy_genes_df%>%
  mutate(location = replace(x = location,
                            list = location == "UPER_STREAM",
                            values= "UPPER_STREAM"))
```

```
##    survey_date       gene_id     location       lat        lon length_mm
## 1    7/27/2020 FMMJ_GUP_1924 UPPER_STREAM 17.124975 -88.124909        25
## 2    7/27/2020 FMMJ_GUP_1924 UPPER_STREAM 17.124975 -88.124909        25
## 3    7/27/2020 AHTD_GUP_5866 UPPER_STREAM 17.124634 -88.125183        19
## 4    7/28/2020 POIU_GUP_5241 UPPER_STREAM  1.712484 -88.125748        23
## 5    7/28/2020         TT_G_4 UPPER_STREAM 17.124049 -88.125883        20
## 6    7/30/2020 GRRA_GUP_5693 UPPER_STREAM 17.124028 -88.124755        22
## 7    7/30/2020 AGTH_GUP_1540 UPPER_STREAM 17.124837 -88.124802        22
## 8    7/30/2020          <NA> UPPER_STREAM 17.124910 -88.124369        24
## 9    7/31/2020 ASDD_GUP_3596 UPPER_STREAM 17.124526 -88.125922        23
## 10    8/1/2020 SDAD_GUP_3114 LOWER_STREAM 17.125321 -88.124812        37
## 11    8/1/2020 THTE_GUP_3944 LOWER_STREAM 17.125925 -88.125675        35
## 12    8/5/2020 SSAS_GUP_1456 LOWER_STREAM 17.124701 -88.124641       230
## 13   8/50/2020 AASD_GUP_1217 LOWER_STREAM 17.125541 -88.124052        32
## 14    8/5/2020 JNKT_GUP_4566 LOWER_STREAM 17.125032 -88.124394        32
## 15    8/7/2020          <NA> LOWER_STREAM 17.124425 -88.125226        40
```

Alternatively, you could create a new column if you do not wish to overwrite the data. Just use a name for **mutate()** that isnt in your data already!

```
guppy_genes_df%>%
  mutate(location_clean = replace(x = location,
                            list = location == "UPER_STREAM",
                            values= "UPPER_STREAM"))
```

```
##    survey_date       gene_id     location       lat        lon length_mm
## 1    7/27/2020 FMMJ_GUP_1924 UPPER_STREAM 17.124975 -88.124909        25
## 2    7/27/2020 FMMJ_GUP_1924 UPPER_STREAM 17.124975 -88.124909        25
## 3    7/27/2020 AHTD_GUP_5866  UPER_STREAM 17.124634 -88.125183        19
## 4    7/28/2020 POIU_GUP_5241 UPPER_STREAM  1.712484 -88.125748        23
## 5    7/28/2020         TT_G_4 UPPER_STREAM 17.124049 -88.125883        20
## 6    7/30/2020 GRRA_GUP_5693  UPER_STREAM 17.124028 -88.124755        22
## 7    7/30/2020 AGTH_GUP_1540 UPPER_STREAM 17.124837 -88.124802        22
## 8    7/30/2020          <NA> UPPER_STREAM 17.124910 -88.124369        24
## 9    7/31/2020 ASDD_GUP_3596 UPPER_STREAM 17.124526 -88.125922        23
## 10    8/1/2020 SDAD_GUP_3114 LOWER_STREAM 17.125321 -88.124812        37
## 11    8/1/2020 THTE_GUP_3944 LOWER_STREAM 17.125925 -88.125675        35
## 12    8/5/2020 SSAS_GUP_1456 LOWER_STREAM 17.124701 -88.124641       230
## 13   8/50/2020 AASD_GUP_1217 LOWER_STREAM 17.125541 -88.124052        32
## 14    8/5/2020 JNKT_GUP_4566 LOWER_STREAM 17.125032 -88.124394        32
## 15    8/7/2020          <NA> LOWER_STREAM 17.124425 -88.125226        40
##    location_clean
## 1    UPPER_STREAM
## 2    UPPER_STREAM
## 3    UPPER_STREAM
## 4    UPPER_STREAM
## 5    UPPER_STREAM
## 6    UPPER_STREAM
## 7    UPPER_STREAM
## 8    UPPER_STREAM
## 9    UPPER_STREAM
## 10   LOWER_STREAM
## 11   LOWER_STREAM
## 12   LOWER_STREAM
## 13   LOWER_STREAM
## 14   LOWER_STREAM
## 15   LOWER_STREAM
```

4.3.2 Cleaning data with filter

Cleaning data with filter()

We've used **filter()** in the past to filter a data set via some logical criteria. We could easily filter out our data set based on that same logic.

```
guppy_genes_df%>%
  filter(location == "UPPER_STREAM")
```

```
##   survey_date       gene_id     location       lat        lon length_mm
## 1   7/27/2020 FMMJ_GUP_1924 UPPER_STREAM 17.124975 -88.124909        25
## 2   7/27/2020 FMMJ_GUP_1924 UPPER_STREAM 17.124975 -88.124909        25
## 3   7/28/2020 POIU_GUP_5241 UPPER_STREAM  1.712484 -88.125748        23
## 4   7/28/2020        TT_G_4 UPPER_STREAM 17.124049 -88.125883        20
## 5   7/30/2020 AGTH_GUP_1540 UPPER_STREAM 17.124837 -88.124802        22
## 6   7/30/2020          <NA> UPPER_STREAM 17.124910 -88.124369        24
## 7   7/31/2020 ASDD_GUP_3596 UPPER_STREAM 17.124526 -88.125922        23
```

While absolutely useful, it is fairly straightforward to filter out based on simple logic. However, lets take this as an opportunity to introduce a common issue in datasets: duplicates.

Cleaning duplicated data with filter() Duplicated data happens often. Data could have been entered twice, there could have been an accidental copy and paste, or the dataset actually should contain duplicated data points!

The process of identifying duplicated data is pretty easy with the **duplicated()** function

```
guppy_genes_df%>%
  duplicated()
```

```
##  [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE
```

TRUE values indicate that the row is a duplicate. In this data, it looks like row 2 is a duplicate of row 1!

```
guppy_genes_df
```

```
##    survey_date       gene_id     location       lat        lon length_mm
## 1    7/27/2020 FMMJ_GUP_1924 UPPER_STREAM 17.124975 -88.124909        25
## 2    7/27/2020 FMMJ_GUP_1924 UPPER_STREAM 17.124975 -88.124909        25
## 3    7/27/2020 AHTD_GUP_5866  UPER_STREAM 17.124634 -88.125183        19
## 4    7/28/2020 POIU_GUP_5241 UPPER_STREAM  1.712484 -88.125748        23
## 5    7/28/2020        TT_G_4 UPPER_STREAM 17.124049 -88.125883        20
## 6    7/30/2020 GRRA_GUP_5693  UPER_STREAM 17.124028 -88.124755        22
## 7    7/30/2020 AGTH_GUP_1540 UPPER_STREAM 17.124837 -88.124802        22
## 8    7/30/2020          <NA> UPPER_STREAM 17.124910 -88.124369        24
## 9    7/31/2020 ASDD_GUP_3596 UPPER_STREAM 17.124526 -88.125922        23
## 10    8/1/2020 SDAD_GUP_3114 LOWER_STREAM 17.125321 -88.124812        37
## 11    8/1/2020 THTE_GUP_3944 LOWER_STREAM 17.125925 -88.125675        35
## 12    8/5/2020 SSAS_GUP_1456 LOWER_STREAM 17.124701 -88.124641       230
## 13   8/50/2020 AASD_GUP_1217 LOWER_STREAM 17.125541 -88.124052        32
## 14    8/5/2020 JNKT_GUP_4566 LOWER_STREAM 17.125032 -88.124394        32
## 15    8/7/2020          <NA> LOWER_STREAM 17.124425 -88.125226        40
```

This could be intentional and should be kept, or its an issue that should be removed.

We can remove duplicated rows very easily using **duplicated()**.

We need to add the period in **duplicated()** as it is within the **filter()** function. The period tells r where to input the object returned from the last pipe.

```
guppy_genes_df%>%
  filter(!duplicated(.))
```

```
##    survey_date       gene_id     location       lat        lon length_mm
## 1    7/27/2020 FMMJ_GUP_1924 UPPER_STREAM 17.124975 -88.124909        25
## 2    7/27/2020 AHTD_GUP_5866  UPER_STREAM 17.124634 -88.125183        19
## 3    7/28/2020 POIU_GUP_5241 UPPER_STREAM  1.712484 -88.125748        23
## 4    7/28/2020        TT_G_4 UPPER_STREAM 17.124049 -88.125883        20
## 5    7/30/2020 GRRA_GUP_5693  UPER_STREAM 17.124028 -88.124755        22
## 6    7/30/2020 AGTH_GUP_1540 UPPER_STREAM 17.124837 -88.124802        22
## 7    7/30/2020          <NA> UPPER_STREAM 17.124910 -88.124369        24
## 8    7/31/2020 ASDD_GUP_3596 UPPER_STREAM 17.124526 -88.125922        23
## 9     8/1/2020 SDAD_GUP_3114 LOWER_STREAM 17.125321 -88.124812        37
## 10    8/1/2020 THTE_GUP_3944 LOWER_STREAM 17.125925 -88.125675        35
## 11    8/5/2020 SSAS_GUP_1456 LOWER_STREAM 17.124701 -88.124641       230
## 12   8/50/2020 AASD_GUP_1217 LOWER_STREAM 17.125541 -88.124052        32
## 13    8/5/2020 JNKT_GUP_4566 LOWER_STREAM 17.125032 -88.124394        32
## 14    8/7/2020          <NA> LOWER_STREAM 17.124425 -88.125226        40
```

We’ve now removed the single duplicate row!

Naturally, we can also use filter to filter by any criteria

4.3.3 putting it all together

Putting it all together

Throughout this section we identified issues in our data, gave some best practices on how to fix them, and taught you how to fix the data yourself using R.

So far, we have just been cleaning the data one step at a time. This section will stitch all those pieces together to create a simple pipeline for cleaning all of the data in one fell swoop!

```
#load in raw data, removes duplicates, NA, fixes date format, latitude typo,
#location typos, length_mm typo, and converts the lon column to a numeric

guppy_genes_df%>%

    filter(!duplicated(.))%>% #duplicates
    filter(!is.na(gene_id))%>% #remove na
    mutate(survey_date, replace(survey_date,survey_date == "8/50","8/5"))%>% #fix dates format
    mutate(lat = replace(lat,lat == "1.712484","17.12484"))%>% #fix typo
    mutate(location = replace(location,location == "UPER_STREAM","UPPER_STREAM"))%>% #fix typo
    mutate(length_mm = replace(length_mm,length_mm == 230,23))%>% #fix typo
    mutate(lon = as.numeric(lon)) -> guppy_genes_df_clean #data class change and output

guppy_genes_df_clean
```

```
##    survey_date       gene_id     location       lat       lon length_mm
## 1    7/27/2020 FMMJ_GUP_1924 UPPER_STREAM 17.124975 -88.12491        25
## 2    7/27/2020 AHTD_GUP_5866 UPPER_STREAM 17.124634 -88.12518        19
## 3    7/28/2020 POIU_GUP_5241 UPPER_STREAM 1.7124844 -88.12575        23
## 4    7/28/2020        TT_G_4 UPPER_STREAM 17.124049 -88.12588        20
## 5    7/30/2020 GRRA_GUP_5693 UPPER_STREAM 17.124028 -88.12475        22
## 6    7/30/2020 AGTH_GUP_1540 UPPER_STREAM 17.124837 -88.12480        22
## 7    7/31/2020 ASDD_GUP_3596 UPPER_STREAM 17.124526 -88.12592        23
## 8     8/1/2020 SDAD_GUP_3114 LOWER_STREAM 17.125321 -88.12481        37
## 9     8/1/2020 THTE_GUP_3944 LOWER_STREAM 17.125925 -88.12568        35
## 10    8/5/2020 SSAS_GUP_1456 LOWER_STREAM 17.124701 -88.12464        23
## 11   8/50/2020 AASD_GUP_1217 LOWER_STREAM 17.125541 -88.12405        32
## 12    8/5/2020 JNKT_GUP_4566 LOWER_STREAM 17.125032 -88.12439        32
##    replace(survey_date, survey_date == "8/50", "8/5")
## 1                                           7/27/2020
## 2                                           7/27/2020
## 3                                           7/28/2020
## 4                                           7/28/2020
## 5                                           7/30/2020
## 6                                           7/30/2020
## 7                                           7/31/2020
## 8                                            8/1/2020
## 9                                            8/1/2020
## 10                                           8/5/2020
## 11                                          8/50/2020
## 12                                           8/5/2020
```

All the hard work from this week can be summed up in just a few lines of code!

This is a perfect example of a well documented pipeline. We can clearly see what each step does, it is well organized, and we can add on new steps as we find more issues. Further, the cleaned data was assigned to a new object guppy_genes_df_clean. This preserves the original as a backup, in case we want to alter our pipeline in the future!

Section 5 - logic, custom functions, and apply()

5.0 Section 5 Overview

Week 5 Overview

At some point you will likely need to perform some type of calculation through an entire dataset. This could be as simple as getting the sum and standard deviation of each rows, or running a suite of statistics and analyses over every object in a list. This may require you to write your own function and have it run over every record in a dataset!

These next few sections will:

- Teach you to make if-else statements
- Describe for-loops and how to construct one
- Introduce the **apply()** function
- Show you how to make simple functions of your own!

The data The dataset for this week describes populations of Jaguars. The scientists are sampling shoulder height between multiple adult individuals in each population. They want to do a simple t.test to see if there are differences in the mean average height between the populations.

This data is constructed using the **rbind()** function and through a quick pipeline to add a population column and then rename the columns. In the course so far, we have learned how to do everything in this code!

```
library(tidyverse)

rbind(c(70,71,72,76,69,74,79,70,70,76),
c(80,70,75,69,76,59,62,70,71,72),
c(60,60,62,64,58,64,69,68,61,62),
c(72,62,64,59,62,48,55,61,63,64),
c(70,68,72,73,69,64,76,70,70,74),
c(80,63,78,59,78,49,62,65,71,62)) ->jag_df


jag_df%>%
  as.data.frame()%>%
  mutate(pop = c("pop_a",'pop_b','pop_c','pop_d','pop_e','pop_f'))%>%
  relocate(pop) -> jag_df


colnames(jag_df) <- c("pop",'indiv_1','indiv_2','indiv_3',
                      'indiv_4','indiv_5','indiv_6',
                      'indiv_7','indiv_8','indiv_9','indiv_10')

jag_df
```

5.1.1 Intro to if-else

Intro to If else statements If else statements are a staple of all programming languages. They create a "fork" in our coding based on some logical criteria. If TRUE, R will run one set of code. If FALSE, R will run another set of code.

Lets give a simple example where if x is larger than y, print "Larger"

```
x = 7
y = 5

if (x > y){
  print("Larger")
}
```

```
## [1] "Larger"
```

In an **if statement,** we first input a logical statement inside the parentheses. If it is TRUE, R runs the code inside the curly brackets {}. This would be the first path in our fork.

We can then add in an **else statement** that tells R what to do if the logical statement is returned FALSE. This would be the second path in our fork. Lets change the values and tell R what to do if Y is less than X.

```
x = 20
y = 25

if(x>y){

  print("x is larger")

}else{

  print("y is larger")

  }
```

```
## [1] "y is larger"
```

Similarly to **if,** the code we want to run is placed inside a new set of curly brackets. Notice that **else** is placed directly after the closing curly bracket from the **if statement**.

5.1.2 Intro to For-Loops

Introduction to For Loops Similarly to **if-else statements**, **for loops** are a common principle in programming. In essence, a **for loop** goes through some object and performs some snippet of code for every nth value in the object.

Take this simple example with a vector named vec that we want to perform a few calculations on.

```
vec <- c(1,5,9,8,7)
```

The for-loop has a few different components we need to explain.

We first need to specify how we want to run the for loop and the object we want to apply the for loop over.

```
for(i in vec){}
```

This so far states says that for every value in the vector vec, perform whatever operations are in the curly brackets. The letter i stands for the index value. This can be any character value.

The index value represents where in the object we are. By default it starts at 1, and increases by 1 each time the loop completes.

Lets add some code to the curly brackets that will print i at every step of the for loop.

```
for(i in vec){print(i)}
```

```
## [1] 1
## [1] 5
## [1] 9
## [1] 8
## [1] 7
```

We now get all the values in the vector! In for-loops with a simple vector, i is simply the value in the vector. The above for loop is essentially the same as running the following code.

```
vec[1]
vec[2]
vec[3]
vec[4]
vec[5]
```

```
## [1] 1
## [1] 5
## [1] 9
## [1] 8
## [1] 7
```

Naturally, we can do more than just print a simple value.

The following output will have 3 columns, the first is the current value, the second is with 5 added to each value, and the third is each value multiplied by 3.

```
for(i in vec){

  print(c(i,
          i+5,
          i*3))}
```

```
## [1] 1 6 3
## [1]  5 10 15
## [1]  9 14 27
## [1]  8 13 24
## [1]  7 12 21
```

5.1.3 Intro to custom functions

Intro to custom functions

Creating functions it not as difficult as it sounds and doing so is really helpful! It dramatically improves your coding ability, makes your scripts easier to understand, and over time can allow you to do more complex analyses easier.

For example, lets say we have 3 vectors. I used **rnorm()** here to create a normal distribution of 100 values.

```
vec_1 <- c(rnorm(100,0,1))
vec_2 <- c(rnorm(100,10,5))
vec_3 <- c(rnorm(100,3,10))
```

Lets say we want to calculate a few summary statistics for each. I included the **print()** function to make the output easier to read.

```
print("Vector 1")
mean(vec_1)
sd(vec_1)
median(vec_1)
range(vec_1)
print("") #print a blank value


print("Vector 2")
mean(vec_2)
sd(vec_2)
median(vec_2)
range(vec_2)
print("")


print("Vector 3")
mean(vec_3)
sd(vec_3)
median(vec_3)
range(vec_3)
print("")
```

```
## [1] "Vector 1"
## [1] 0.06841222
## [1] 0.9662672
## [1] 0.03591471
## [1] -2.014210  2.293079
## [1] ""
## [1] "Vector 2"
## [1] 9.784007
## [1] 5.015307
## [1] 9.892128
## [1] -2.329491 22.857291
## [1] ""
## [1] "Vector 3"
## [1] 3.905776
## [1] 10.12999
## [1] 4.582607
## [1] -23.60923  26.97452
## [1] ""
```

This code is quite long and repetitive. We could instead create a custom function to make our lives easier!

We do so by using **function()**. **function()** contains arguments and objects we want to use within the parentheses. The code we wish to run is contained within curly brackets {}.

In the **return()** function, we place the object from within the function we wish to output. In our case, we combined all the results into an object named output.

```
summary_vect <- function(x){

  output <- c(mean(x), sd(x), median(x), range(x))

  return(output)
}
```

Now look what happens if we run this function for all 3 vectors

```
summary_vect(vec_1)
```

```
## [1]  0.06841222  0.96626723  0.03591471 -2.01421050  2.29307897
```

```
summary_vect(vec_2)
```

```
## [1]  9.784007  5.015307  9.892128 -2.329491 22.857291
```

```
summary_vect(vec_3)
```

```
## [1]   3.905776  10.129989   4.582607 -23.609228  26.974525
```

Writing custom functions helps us keep our code clean, makes our analyses more reproducible, and makes updating your code much easier! If we want to add or remove one summary statistic, we just need to make the change in the function, not throughout the entire script!

In general, keep your functions as simple as possible and no simpler!

5.2.1 for loops on dataframes

For loops on dataframes

Lets give an example where we need to perform some data manipulations and some analyses over our jaguar dataframe.

When iterating through a dataframe object, we need to be explicit about how we iterate. For example, if we just iterate through jag_df, every single value will be returned.

The values are displayed in the console using the **print() function**.

```r
for(i in jag_df){print(i)}
```

```
## [1] "pop_a" "pop_b" "pop_c" "pop_d" "pop_e" "pop_f"
## [1] 70 80 60 72 70 80
## [1] 71 70 60 62 68 63
## [1] 72 75 62 64 72 78
## [1] 76 69 64 59 73 59
## [1] 69 76 58 62 69 78
## [1] 74 59 64 48 64 49
## [1] 79 62 69 55 76 62
## [1] 70 70 68 61 70 65
## [1] 70 71 61 63 70 71
## [1] 76 72 62 64 74 62
```

This may be useful in some circumstances, but typically we want to go through the dataframe row by row or column by column. We can accomplish this by using the **nrow()** function (or **ncol()** for columns).

By placing a "1:" in front of **nrow()**, we tell R that we want to iterate starting from 1 up to the number of rows in the dataset. This will make sure we hit every observation.

Notice, that when we run this chunk, i is no longer the value itself. It is now the index value of each row.

```r
for(i in 1:nrow(jag_df)){print(i)}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
```

Lets run a for-loop that shows how to subset the dataset by row. This will help you get a visual representation of the data we are working with with each iteration of i.

```r
for(i in 1:nrow(jag_df)){

  print(jag_df[i,])

}
```

```
##     pop individual_1 individual_2 individual_3 individual_4 individual_5
## 1 pop_a           70           71           72           76           69
##   individual_6 individual_7 individual_8 individual_9 individual_10
## 1           74           79           70           70            76
##     pop individual_1 individual_2 individual_3 individual_4 individual_5
## 2 pop_b           80           70           75           69           76
##   individual_6 individual_7 individual_8 individual_9 individual_10
## 2           59           62           70           71            72
##     pop individual_1 individual_2 individual_3 individual_4 individual_5
## 3 pop_c           60           60           62           64           58
##   individual_6 individual_7 individual_8 individual_9 individual_10
## 3           64           69           68           61            62
##     pop individual_1 individual_2 individual_3 individual_4 individual_5
## 4 pop_d           72           62           64           59           62
##   individual_6 individual_7 individual_8 individual_9 individual_10
## 4           48           55           61           63            64
##     pop individual_1 individual_2 individual_3 individual_4 individual_5
## 5 pop_e           70           68           72           73           69
##   individual_6 individual_7 individual_8 individual_9 individual_10
## 5           64           76           70           70            74
##     pop individual_1 individual_2 individual_3 individual_4 individual_5
## 6 pop_f           80           63           78           59           78
##   individual_6 individual_7 individual_8 individual_9 individual_10
## 6           49           62           65           71            62
```

Essentially, this for-loop is first running jag_df[1,], then jag_df[2,] then jag_df[3,] and so on until it runs out of rows.

5.2.2 For-loops for analyses

For-loops for Analyses

Lets create a for loop that goes through each row of the dataset and runs a simple **t.test()**. The test will compare the mean jaguar heights of each population to a control population to detect any significant differences.

```
control_pop_heights <- c(71,72,74,73,77,70,73,77,72,75)

for(i in 1:nrow(jag_df)){

    t.test(control_pop_heights,jag_df[i,2:11])%>%
    print()

}
```

```
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and jag_df[i, 2:11]
## t = 0.53775, df = 16.144, p-value = 0.5981
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -2.057493  3.457493
## sample estimates:
## mean of x mean of y
##      73.4      72.7
##
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and jag_df[i, 2:11]
## t = 1.4216, df = 11.537, p-value = 0.1816
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
```

```
##  -1.618501  7.618501
## sample estimates:
## mean of x mean of y
##      73.4      70.4
##
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and jag_df[i, 2:11]
## t = 7.9008, df = 15.752, p-value = 7.24e-07
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##   7.752204 13.447796
## sample estimates:
## mean of x mean of y
##      73.4      62.8
##
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and jag_df[i, 2:11]
## t = 5.8498, df = 11.512, p-value = 9.256e-05
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##   7.759648 17.040352
## sample estimates:
## mean of x mean of y
##      73.4      61.0
##
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and jag_df[i, 2:11]
## t = 2.1489, df = 16.132, p-value = 0.04716
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  0.03962077 5.56037923
## sample estimates:
## mean of x mean of y
##      73.4      70.6
##
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and jag_df[i, 2:11]
## t = 2.0792, df = 10.023, p-value = 0.06422
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.4775583 13.8775583
## sample estimates:
## mean of x mean of y
##      73.4      66.7
```

Results! It seems some populations may differ from the control based on the p-value. This output is useful, but is a bit messy.

Lets do another for loop that prints out the results in an easier to read format, and includes the mean height for each population.

```r
for(i in 1:nrow(jag_df)){

    #here we create a temp variable only for printing the results of the t.test
  #This will get overwritten with every new iteration of the for loop

    temp <- t.test(control_pop_heights,as.numeric(jag_df[i,2:11]))

    population <- jag_df[i,"pop"]

    #cat is a useful function that concats things together to print.
    #The seperator '\n' makes a new line

    #we access results of the t.test using the $ on the temp variable.

    cat("Population:", population,'\n',
         "Mean height is:",mean(as.numeric(jag_df[i,2:11])),'\n',
         "P-value from t.test:", temp$p.value, "\n","\n")

  }
```

```
## Population: pop_a
##  Mean height is: 72.7
##  P-value from t.test: 0.5980817
##
## Population: pop_b
##  Mean height is: 70.4
##  P-value from t.test: 0.1816071
##
## Population: pop_c
##  Mean height is: 62.8
##  P-value from t.test: 7.239574e-07
##
## Population: pop_d
##  Mean height is: 61
##  P-value from t.test: 9.256243e-05
##
## Population: pop_e
##  Mean height is: 70.6
##  P-value from t.test: 0.04715914
##
## Population: pop_f
##  Mean height is: 66.7
##  P-value from t.test: 0.06421611
##
```

The above chunk great use case for a for-loop. Namely, going through a dataset, performing some series of operations, and providing us an easy to read output.

For loops can get considerably more advanced (e.g. doing every nth record instead of every one, dynamically creating outputs, looping everything into a list etc), however, that is beyond the scope of this course!

5.3.1 Apply function

The apply() Function

In the last section we learned about for-loops, which iterate over a dataset and perform some set of functions.

Here we will learn about a cousin of the for loops, the **apply()** function.

apply() functions essentially do the same thing as for loops, however, **apply()** is easier to read, needs less explicit instruction, and in some cases, can be faster than for loops

apply() takes in 3 arguments, *X*, *MARGIN*, and *FUN*

X is the input data that you want to apply over

MARGIN is the direction you want to apply. MARGIN = 1 is for rows, MARGIN = 2 is for columns, MARGIN = c(1,2) applies the function over both rows and columns

FUN is the function you wish to apply.

Lets use **apply()** to get the means of each row in the dataset. Notice X is using the subsetted dataset to exclude the character column pop.

```r
apply(X = jag_df[,2:11],
      MARGIN = 1,
      FUN = mean)
```

```
## [1] 72.7 70.4 62.8 61.0 70.6 66.7
```

A big benefit of **apply()** is that you can return the results of **apply()** to a new object as well. While absolutely possible with for-loops, it is generally more intuitive with apply.

```r
jag_means <- apply(X = jag_df[,2:11],
      MARGIN = 1,
      FUN = mean)
```

To call more complex functions, we need to use **function(x)** in the FUN argument. x in this case is similar to how i is used in the for loop. For example, using **t.test()** just like we did with the for-loops

```r
control_pop_heights <- c(71,72,74,73,77,70,73,77,72,75)

apply(X = jag_df[,2:11],
      MARGIN = 1,
      FUN = function(x) t.test(x,control_pop_heights))
```

```
## [[1]]
##
##  Welch Two Sample t-test
##
## data:  x and control_pop_heights
## t = -0.53775, df = 16.144, p-value = 0.5981
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -3.457493  2.057493
## sample estimates:
## mean of x mean of y
##      72.7      73.4
##
##
## [[2]]
##
##  Welch Two Sample t-test
##
## data:  x and control_pop_heights
## t = -1.4216, df = 11.537, p-value = 0.1816
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -7.618501  1.618501
## sample estimates:
## mean of x mean of y
##      70.4      73.4
##
##
## [[3]]
##
##  Welch Two Sample t-test
##
## data:  x and control_pop_heights
## t = -7.9008, df = 15.752, p-value = 7.24e-07
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -13.447796  -7.752204
## sample estimates:
## mean of x mean of y
##      62.8      73.4
##
##
## [[4]]
##
##  Welch Two Sample t-test
##
## data:  x and control_pop_heights
## t = -5.8498, df = 11.512, p-value = 9.256e-05
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -17.040352  -7.759648
## sample estimates:
## mean of x mean of y
##      61.0      73.4
##
##
## [[5]]
##
##  Welch Two Sample t-test
##
## data:  x and control_pop_heights
## t = -2.1489, df = 16.132, p-value = 0.04716
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -5.56037923 -0.03962077
## sample estimates:
## mean of x mean of y
##      70.6      73.4
##
##
## [[6]]
##
##  Welch Two Sample t-test
##
## data:  x and control_pop_heights
## t = -2.0792, df = 10.023, p-value = 0.06422
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -13.8775583   0.4775583
## sample estimates:
## mean of x mean of y
##      66.7      73.4
```

5.3.2 Tying it all together

Tying it all together

Take a look at the for loop we created to show our **t.test()** results

```
control_pop_heights <- c(71,72,67,68,69,70,71,71,72,71)

for(i in 1:nrow(jag_df)){

    #here we create a temp variable only for printing the results of the t.test This will get overwritt

    temp <- t.test(control_pop_heights,as.numeric(jag_df[i,2:11]))

    population <- jag_df[i,"pop"]

    #cat is a useful function that concats things together to print. The   seperator '\n' makes a new l

    #we access results of the t.test using the $ on the temp variable.

    cat("Population:", population,'\n',
         "Mean height is:",mean(as.numeric(jag_df[i,2:11])),'\n',
         "P-value from t.test:", temp$p.value, "\n","\n")

  }
```

After scraping the hard drive, you discovered 3 more datasets of jaguar populations. Sure we could copy and paste the for loops and change the values needed for each loop. However, this could easily lead to a lengthy, messy script that is difficult to troubleshoot!

```
for(i in 1:nrow(jag_df)){
    temp <- t.test(control_pop_heights,as.numeric(jag_df[i,2:11]))

    population <- jag_df[i,"pop"]

    cat("Population:", population,'\n',
         "Mean height is:",mean(as.numeric(jag_df[i,2:11])),'\n',
         "P-value from t.test:", temp$p.value, "\n","\n")

}


for(i in 1:nrow(jag_df_2)){

    temp <- t.test(control_pop_heights,as.numeric(jag_df_2[i,2:11]))

    population <- jag_df_2[i,"pop"]

    cat("Population:", population,'\n',
         "Mean height is:",mean(as.numeric(jag_df_2[i,2:11])),'\n',
         "P-value from t.test:", temp$p.value, "\n","\n")

}


for(i in 1:nrow(jag_df_3)){


    temp <- t.test(control_pop_heights,as.numeric(jag_df_3[i,2:11]))

    population <- jag_df_3[i,"pop"]

    cat("Population:", population,'\n',
         "Mean height is:",mean(as.numeric(jag_df_3[i,2:11])),'\n',
         "P-value from t.test:", temp$p.value, "\n","\n")

}
```

Look a lengthy, messy script that is difficult to troubleshoot!

Instead, lets pull out the code inside the for loop and turn it into a function!

```
fun_height_summary <- function(x){

  control_pop_heights <- c(71,72,67,68,69,70,71,71,72,71)

  #Run t.test
    t_test_results <- t.test(control_pop_heights,as.numeric(x[2:11]))

  #extract which population is being run
    population <- x["pop"]

  #Output to console
    cat("Population:", population,'\n',
         "Mean height is:",mean(as.numeric(x[2:11])),'\n',
         "P-value from t.test:", t_test_results$p.value, "\n","\n")
  #return t_test_results
    return(t_test_results)
    }
```

Now we can use that function anywhere in our R environment! If we wanted to use that function over **apply()** we now can simply type in the function name. We are applying this function over all of the rows in jag_df.

```
apply(jag_df,1,fun_height_summary)
```

```
## Population: pop_a
##  Mean height is: 72.7
##  P-value from t.test: 0.05555104
##
## Population: pop_b
##  Mean height is: 70.4
##  P-value from t.test: 0.9239286
##
## Population: pop_c
##  Mean height is: 62.8
##  P-value from t.test: 4.607665e-05
##
## Population: pop_d
##  Mean height is: 61
##  P-value from t.test: 0.001097607
##
## Population: pop_e
##  Mean height is: 70.6
##  P-value from t.test: 0.7425727
##
## Population: pop_f
##  Mean height is: 66.7
##  P-value from t.test: 0.298002
##
```

```
## [[1]]
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and as.numeric(x[2:11])
## t = -2.0988, df = 13.246, p-value = 0.05555
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -5.06851405  0.06851405
## sample estimates:
## mean of x mean of y
##      70.2      72.7
##
##
## [[2]]
##
```

```
##  Welch Two Sample t-test
##
## data:  control_pop_heights and as.numeric(x[2:11])
## t = -0.097849, df = 10.308, p-value = 0.9239
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -4.735846  4.335846
## sample estimates:
## mean of x mean of y
##      70.2      70.4
##
##
## [[3]]
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and as.numeric(x[2:11])
## t = 5.9934, df = 12.923, p-value = 4.608e-05
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##   4.731003 10.068997
## sample estimates:
## mean of x mean of y
##      70.2      62.8
##
##
## [[4]]
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and as.numeric(x[2:11])
## t = 4.4797, df = 10.295, p-value = 0.001098
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##   4.641727 13.758273
## sample estimates:
## mean of x mean of y
##      70.2      61.0
##
##
## [[5]]
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and as.numeric(x[2:11])
## t = -0.33541, df = 13.235, p-value = 0.7426
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -2.971742  2.171742
## sample estimates:
## mean of x mean of y
##      70.2      70.6
##
##
```

```
## [[6]]
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and as.numeric(x[2:11])
## t = 1.1009, df = 9.5208, p-value = 0.298
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -3.632493 10.632493
## sample estimates:
## mean of x mean of y
##      70.2      66.7
```

That's a bit of a messy output! This is because our code prints results to the console with **cat()** as well as returns the t_test_results object. What we can do to clean this up, is add a summary argument that if FALSE, does not print the output from **cat()**.

```
fun_height_summary <- function(x, summary = TRUE){

    control_pop_heights <- c(71,72,67,68,69,70,71,71,72,71)

    t_test_results <- t.test(control_pop_heights,as.numeric(x[2:11]))

    population <- x["pop"]

    #cat is a useful function that concats things together to print. The   seperator '\n' makes a new l

    #we access results of the t.test using the $ on the temp variable.
    if(summary == TRUE){

          cat("Population:", population,'\n',
         "Mean height is:",mean(as.numeric(x[2:11])),'\n',
         "P-value from t.test:", t_test_results$p.value, "\n","\n")
      }

return(t_test_results)
}
```

Now the code will only output the t_test_results from **return()** if summary is equal to FALSE. Remember, that we need to use the **function(x)** format for the function with multiple arguments.

```
apply(jag_df,1,function(x) fun_height_summary(x, FALSE))
```

```
## [[1]]
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and as.numeric(x[2:11])
## t = -2.0988, df = 13.246, p-value = 0.05555
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -5.06851405  0.06851405
## sample estimates:
```

```
## mean of x mean of y
##      70.2      72.7
##
##
## [[2]]
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and as.numeric(x[2:11])
## t = -0.097849, df = 10.308, p-value = 0.9239
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -4.735846  4.335846
## sample estimates:
## mean of x mean of y
##      70.2      70.4
##
##
## [[3]]
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and as.numeric(x[2:11])
## t = 5.9934, df = 12.923, p-value = 4.608e-05
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##   4.731003 10.068997
## sample estimates:
## mean of x mean of y
##      70.2      62.8
##
##
## [[4]]
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and as.numeric(x[2:11])
## t = 4.4797, df = 10.295, p-value = 0.001098
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##   4.641727 13.758273
## sample estimates:
## mean of x mean of y
##      70.2      61.0
##
##
## [[5]]
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and as.numeric(x[2:11])
## t = -0.33541, df = 13.235, p-value = 0.7426
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
```

```
##  -2.971742  2.171742
## sample estimates:
## mean of x mean of y
##      70.2      70.6
##
##
## [[6]]
##
##  Welch Two Sample t-test
##
## data:  control_pop_heights and as.numeric(x[2:11])
## t = 1.1009, df = 9.5208, p-value = 0.298
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -3.632493 10.632493
## sample estimates:
## mean of x mean of y
##      70.2      66.7
```

If we we want to run this function over our several jaguar data sets and assign the results of the t tests to an object, that is incredibly easy!

```r
jag_1_ttest <- apply(jag_df_1,1,function(x) fun_height_summary(x, FALSE))

jag_2_ttest <- apply(jag_df_2,1,function(x) fun_height_summary(x, FALSE))

jag_3_ttest <- apply(jag_df_3,1,function(x) fun_height_summary(x, FALSE))
```

This section shows how we can take messy, and redundant code and make it understandable using some fundamental programming elements. Between for loops, if else statements, and creating your own custom functions, there is no task you R you cannot complete!

Section 6 - Plotting with ggplot2

6.0 Section 6 Overview

Section 6 Overview

This is last section of the R course! At this stage you are well on your way to become an R wizard! This week will focus entirely on creating graphics using ggplot2. ggplot2 is a standalone package, however it comes included with tidyverse! ggplot2 has a bit of a learning curve, however it offers unparalleled customization of your visualizations and is by far the most popular visualization package in the R ecosystem.

This section will cover:

- how to make and customize plots with ggplot
- how to create multiple plots of your data using faceting
- how to craft your own custom ggplot themes

The data and loading tidyverse The data for this section describes results from a series of fruit fly experiments. I used a series of functions to create more realistic looking data sets for this section. Each is briefly explained below.

group = This a column to create 3 groups. The function **rep**() simply repeats "group_1", "group_2", "group_3" 200 times each to create 600 rows of data.

lifespan_days = The lifespan of the fruit flies in days. The function **rnorm**() produces values from a normal distribution given a mean and a standard deviation. The code here created 200 values and it was run 3 times (one for each group) for 600 data values.

length_mm = The length of each fly in millimeters. **rnorm**() was used in a similar way as lifespan.

eye_color = 4 eye colors repeated 150 times each for 600 data values.

wing_size = The size of the wing either reported as "none" "small" or "normal". The **sample**() function allows us to sample each of those reported values based on some probability that each is picked. We ran 3 iterations of this function, sampling 200 values each, with different probabilities for each iteration.

```r
library(tidyverse)

set.seed(1234) #set seed ensures our distributions are the same every time

fruit_fly_df <- data.frame(
  group = rep(c("group_1", "group_2",'group_3'), each = 200),

  lifespan_days = c(rnorm(200,7.5,.7),
                    rnorm(200,9,.6),
                    rnorm(200,10,.3)),

  length_mm =c(rnorm(200,3,2)+1,
                     rnorm(200,4,1)+1,
                     rnorm(200,5,2)+1),
  eye_color = rep(c("green","red","black","brown"),150),

  wing_size = c(sample(c("none","small","normal"),200,
                       prob = c(.4,.5,.1), replace = TRUE),
                sample(c("none","small","normal"),200,
                       prob = c(.1,.5,.1), replace = TRUE),
                sample(c("none","small","normal"),200,
                       prob = c(.4,.1,.5), replace = TRUE)))
```

6.1.1 Intro to ggplot2

Intro to ggplot2

ggplot2 is a package specifically for data visualization and is included within the tidyverse package. ggplot2 allows for full customization of every element in your plot as well as layering multiple graphics over one another.

Plots made in ggplot2 utilize 3 main components: data, aesthetics, and geometric layers

data is the data object being fed into ggplot, typically a data frame.

aesthetics describes how the variables within our data can be mapped to different aesthetic elements of our plot. For example, we can map which data goes on the x axis and which data goes on the y axis.

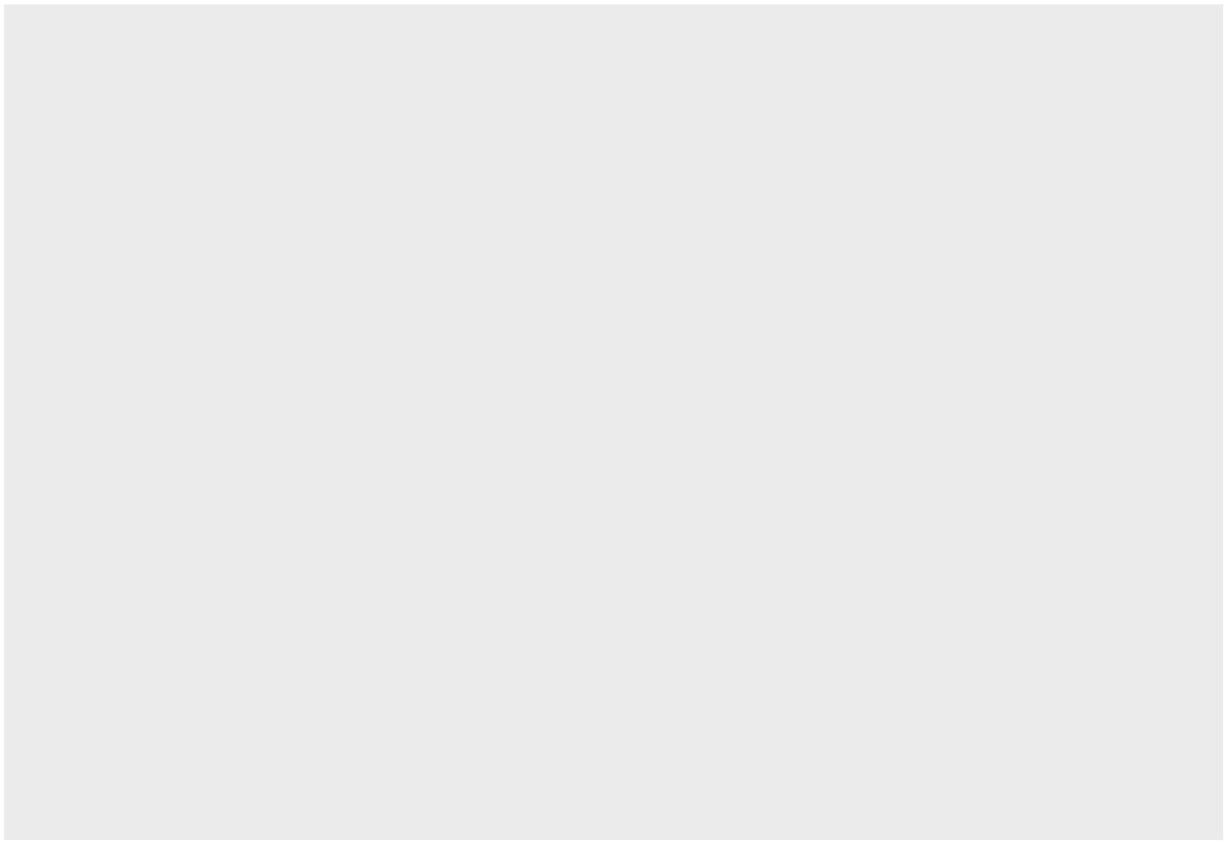
geometric layers (geom layers) describe exactly how the data will displayed. There are a wide variety of geom layer types available. geom_point creates a scatter plot, geom_boxplot creates a boxplot, and geom_line creates line plots to name a few.

Lets introduce each of these major components

Data The main function we will use from ggplot2 is **ggplot**().

Using **ggplot**(), lets first load in the data and attempt to create a plot.

```
ggplot(data = fruit_fly_df)
```



Nothing happened!

Exactly as intended. The data is loaded via the ggplot function, however r doesnt know what to do with that data yet as there are many unknowns.

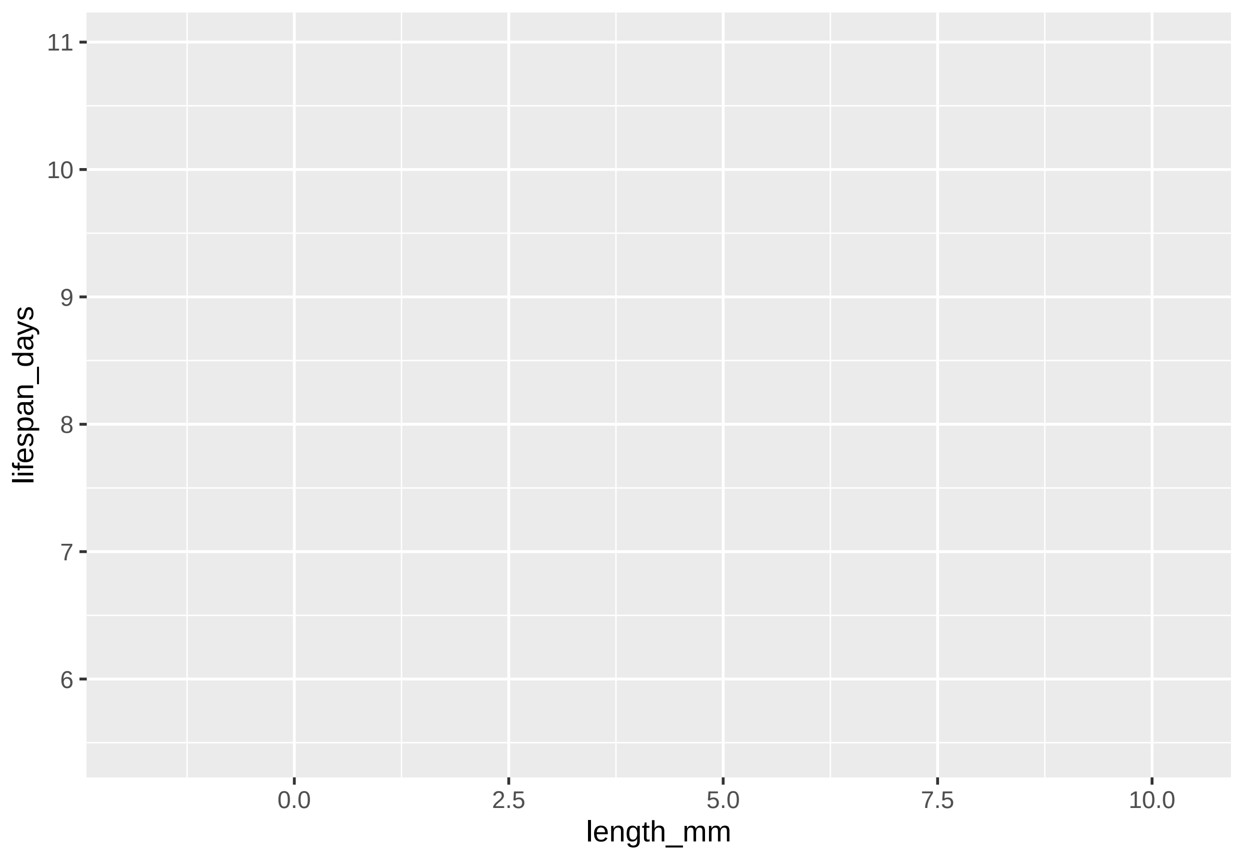
Which variables are actually going to be plotted? Are there variables we can use to group our data? Is there a column we can use to label our data points?

That's where aesthetic mapping comes into play!

Aesthetic mapping In order to use the data, we need to map its variables to aesthetics using **aes**(). **aes**() will tell r how to use our data in the visualization. Lets add it to our **ggplot**() function to map a few variables.

Here we are mapping the length_mm variable to the x axis (or as an independent variable) and the lifespan_days variable to the y axis (or as a dependent variable).

```
ggplot(data = fruit_fly_df, aes(x = length_mm, y = lifespan_days))
```

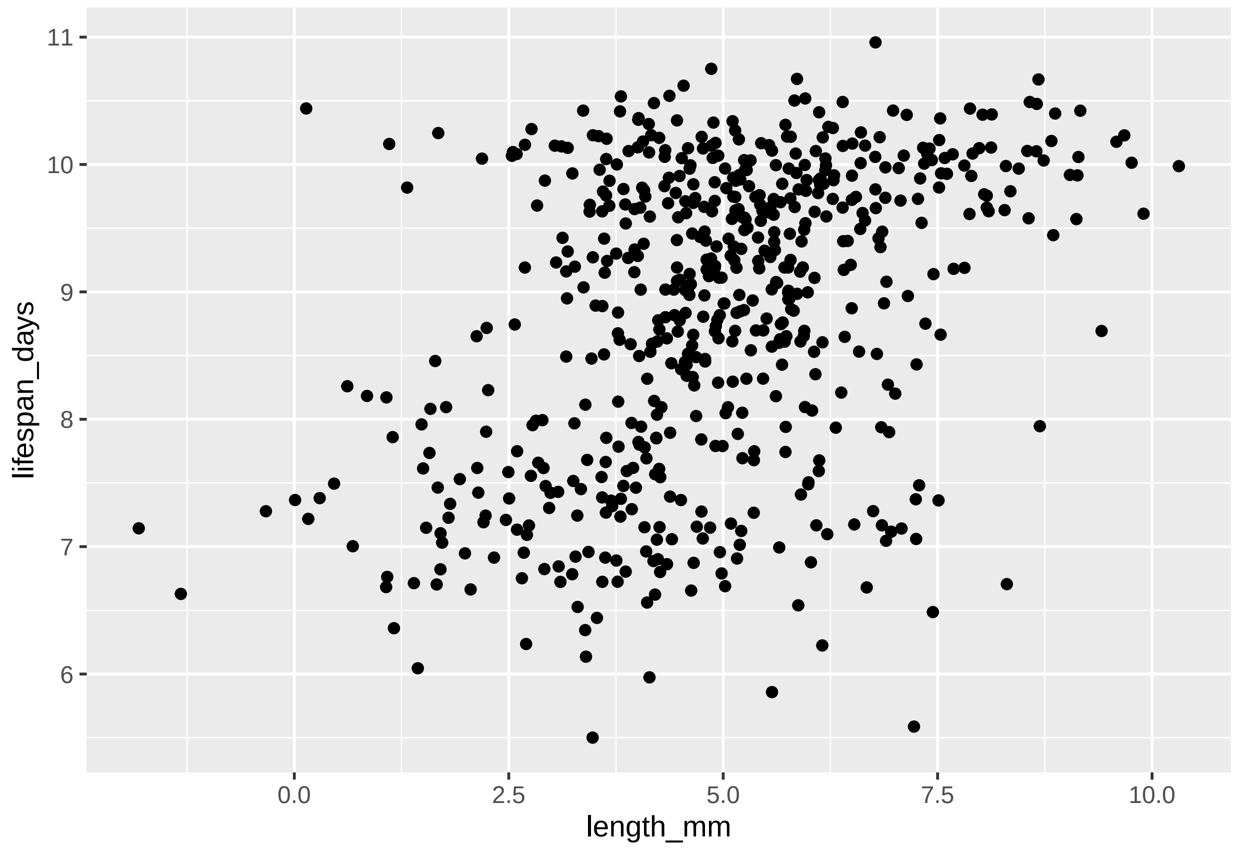


If we run this code, we get a plot but none of of the data was displayed. This is because we have so far only loaded the ggplot() layer. This layer simply loads the data and sets up the default aesthetic mappings. Since we supplied it with 2 variables, it already created a base plot window based off of that data (notice the x and y axis limits/labels).

Geometric Layer If we want to specify how to visualize our data, we need to include a geom layer. Lets add in just the **geom_point**() layer to create a scatter plot.

We specify the next layer of our ggplot using the plus symbol (+) and each layer is its own self contained function. Here we want to add another layer, geom_point, that creates the scatter plot.

```
ggplot(data = fruit_fly_df, aes(x = length_mm, y = lifespan_days))+
  geom_point()
```



A very simple scatterplot! Hooray!

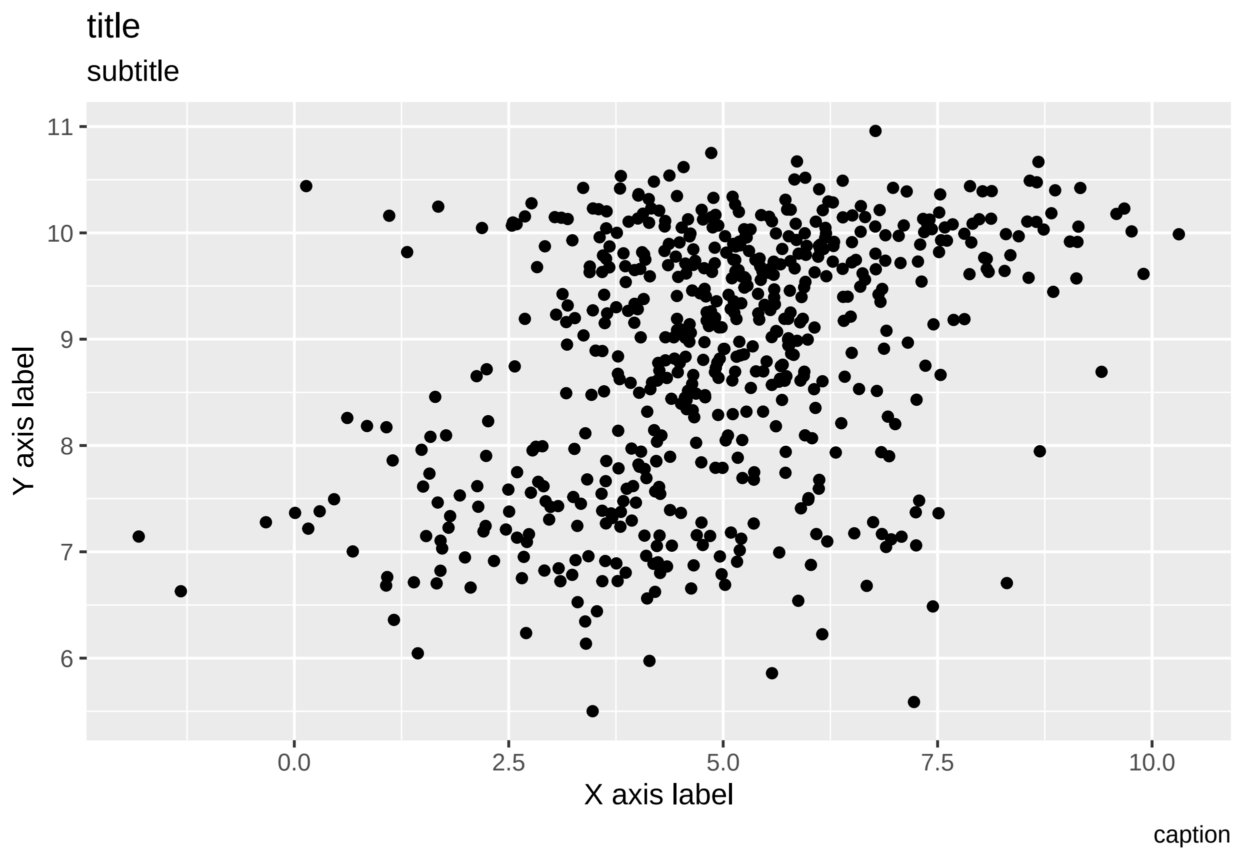
These three elements of the ggplot: the data, the aesthetics, and the geometric layers; form the basis for every single ggplot you will make from here on out. Through the next few sections you will learn how to customize every element of the above plot!

6.1.2 labeling layers and ggobjects

Labeling, layers, and gg objects

labeling using labs() Lets now handle some basics, such as labeling. We can specify our labels all in the **labs**() function. We can customize the title, subtitle, caption, x-axis label, y-axis label, and even the alt text all within **labs**()!

```
ggplot(data = fruit_fly_df, aes(x = length_mm, y = lifespan_days))+
  geom_point()+
  labs(title = "title", subtitle = "subtitle", caption = "caption",
       x = "X axis label", y = "Y axis label",
       alt = "Simple alt text describing the graph")
```

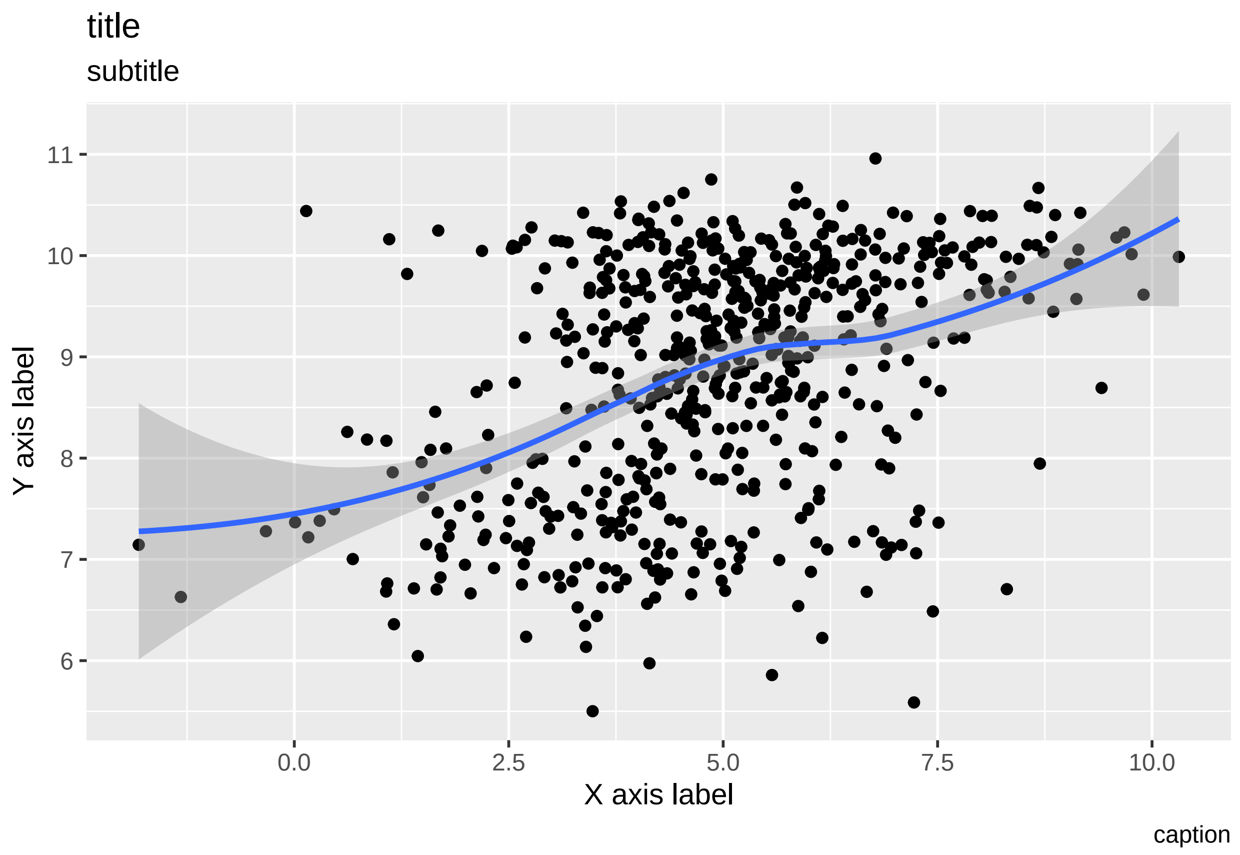


New layers As mentioned earlier, ggplot works with layers. Lets say we want to add a new layer on top of the geom_point layer that shows a trendline.

We can do so by adding a + and creating a new layer. Lets add a layer which creates a trend line based on the data. To do this, we will use **geom_smooth**()

```
ggplot(data = fruit_fly_df, aes(x = length_mm, y = lifespan_days))+
   labs(title = "title", subtitle = "subtitle", caption = "caption",
       x = "X axis label", y = "Y axis label",
       alt = "Simple alt text describing the graph")+
  geom_point()+
  geom_smooth()
```

```
## ‘geom_smooth()‘ using method = ’loess’ and formula ’y ~ x’
```



Straight forward! We can move the layers around as we please, but know that the the topmost layer is what is loaded first, then the second is loaded on top of the first, and so on and so forth.

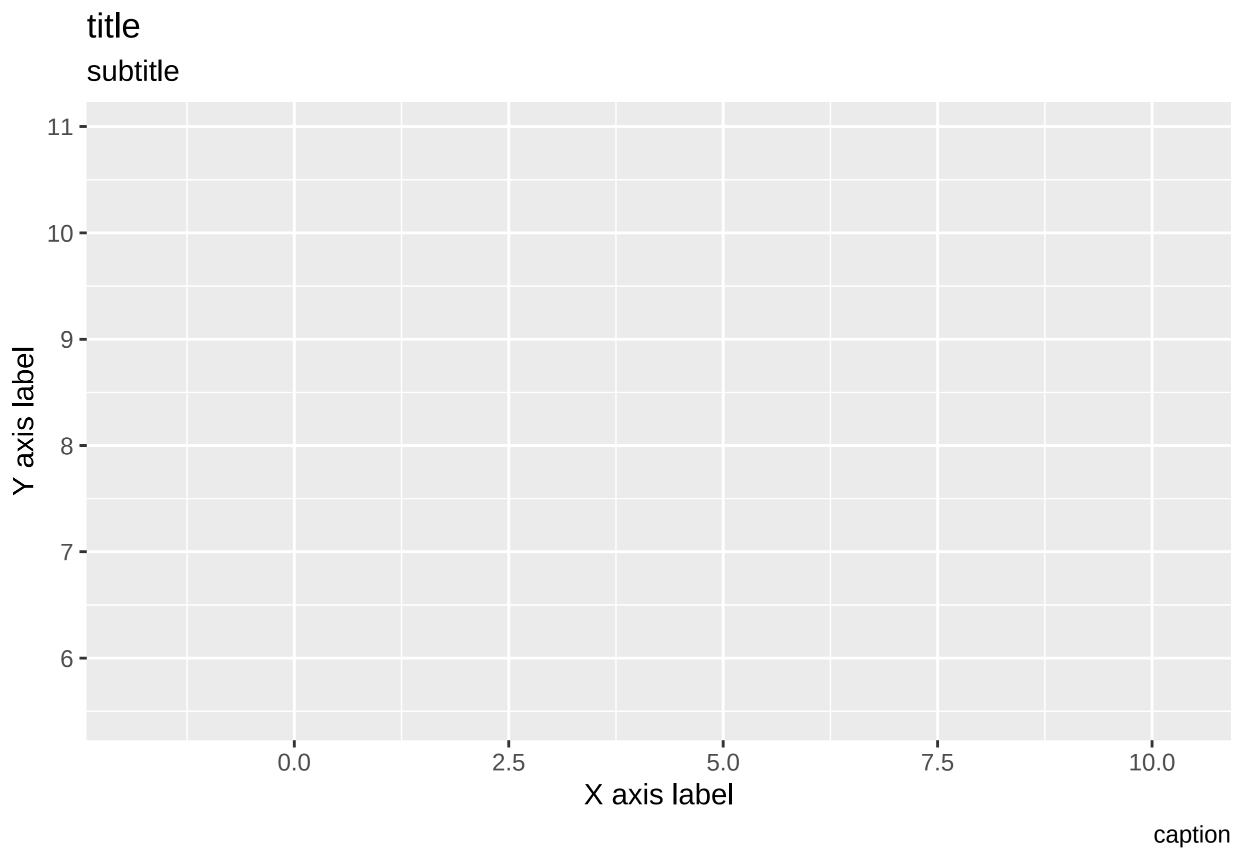
Creating gg objects A huge benefit of ggplot is that we can save our ggplot code as an object!

For example, lets save the **ggplot**() and **labs**() portions into a single object named gg.

```
gg <- ggplot(data = fruit_fly_df, aes(x = length_mm, y = lifespan_days))+
   labs(title = "title", subtitle = "subtitle", caption = "caption",
       x = "X axis label", y = "Y axis label",
       alt = "Simple alt text describing the graph")
```

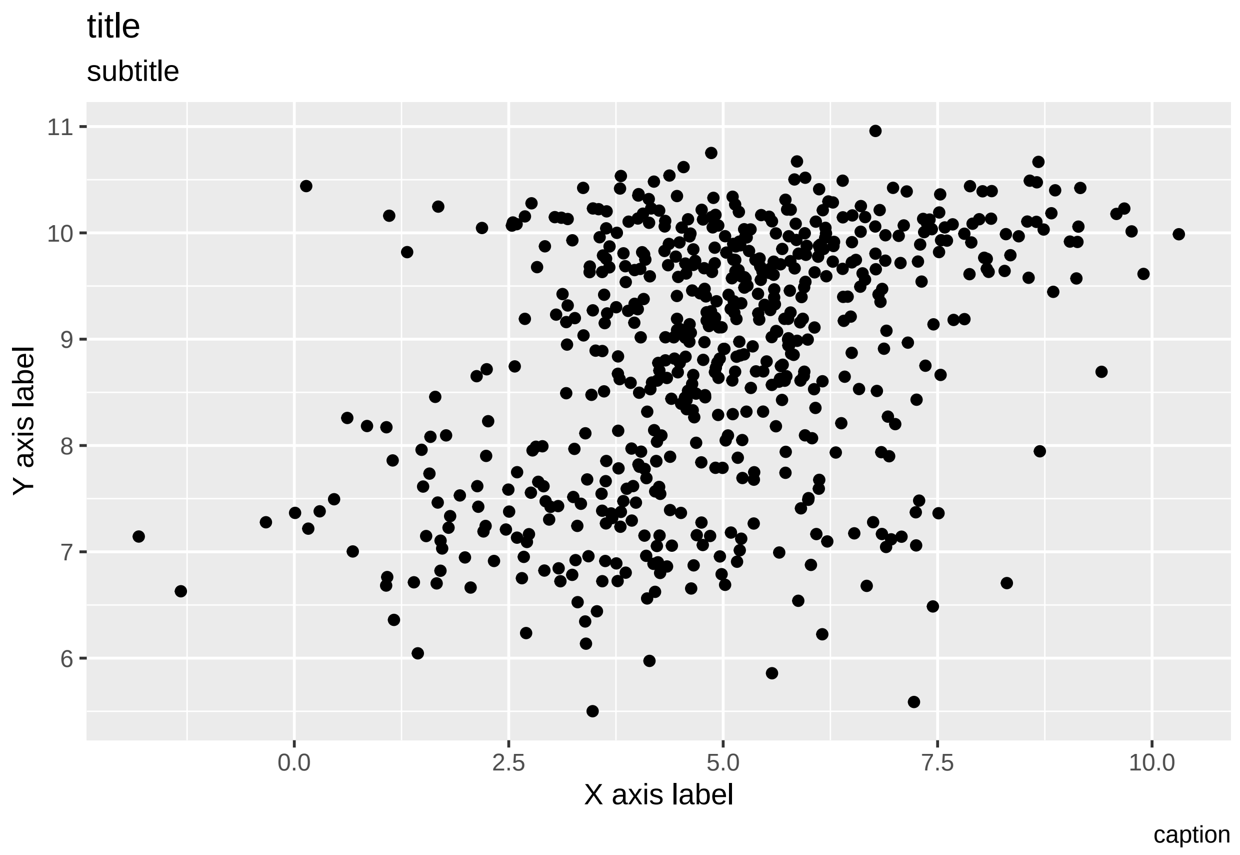
If we run gg, we will get that base plot once again.

```
gg
```



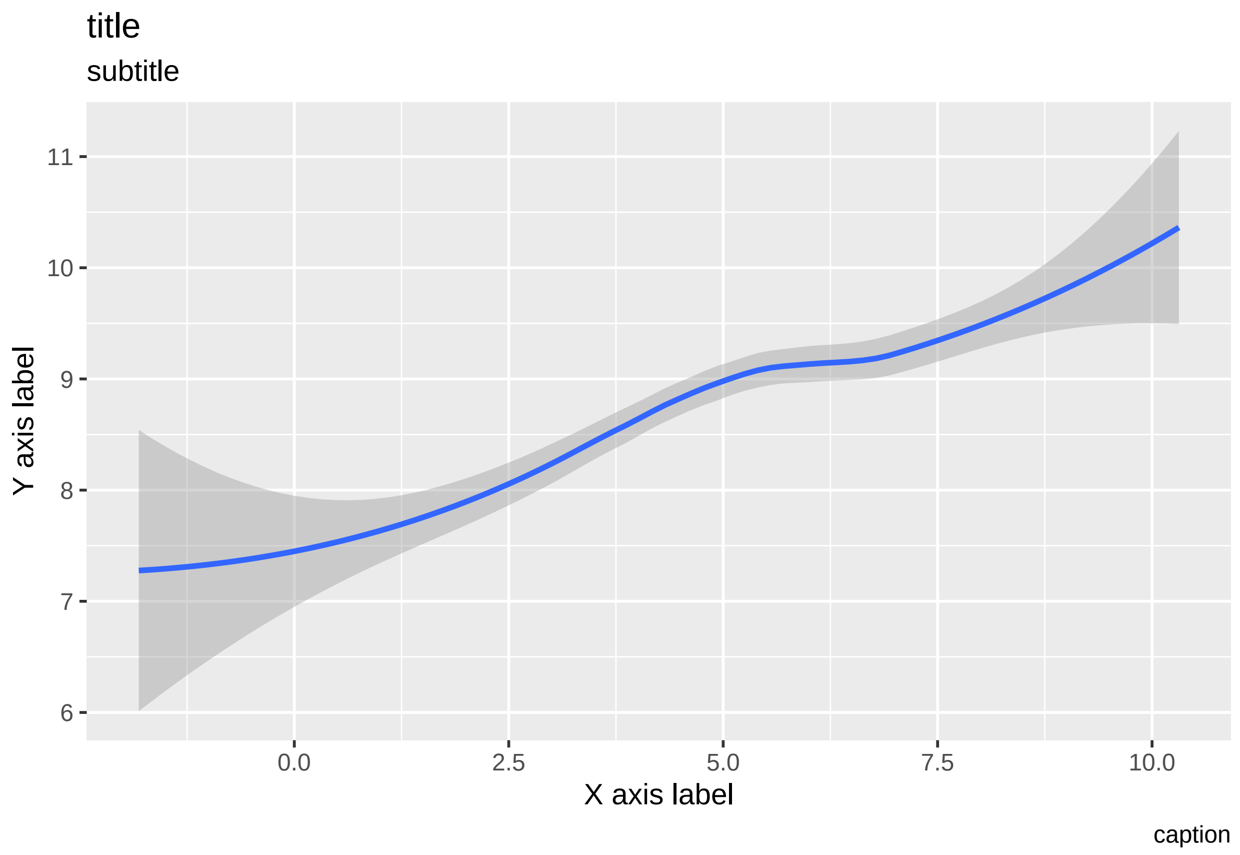
From there we can easily add new layers without needing to repeat information! Lets quickly create 3 plots. Each will inherit the same information from gg to create their plot.

```
gg+
  geom_point()
```



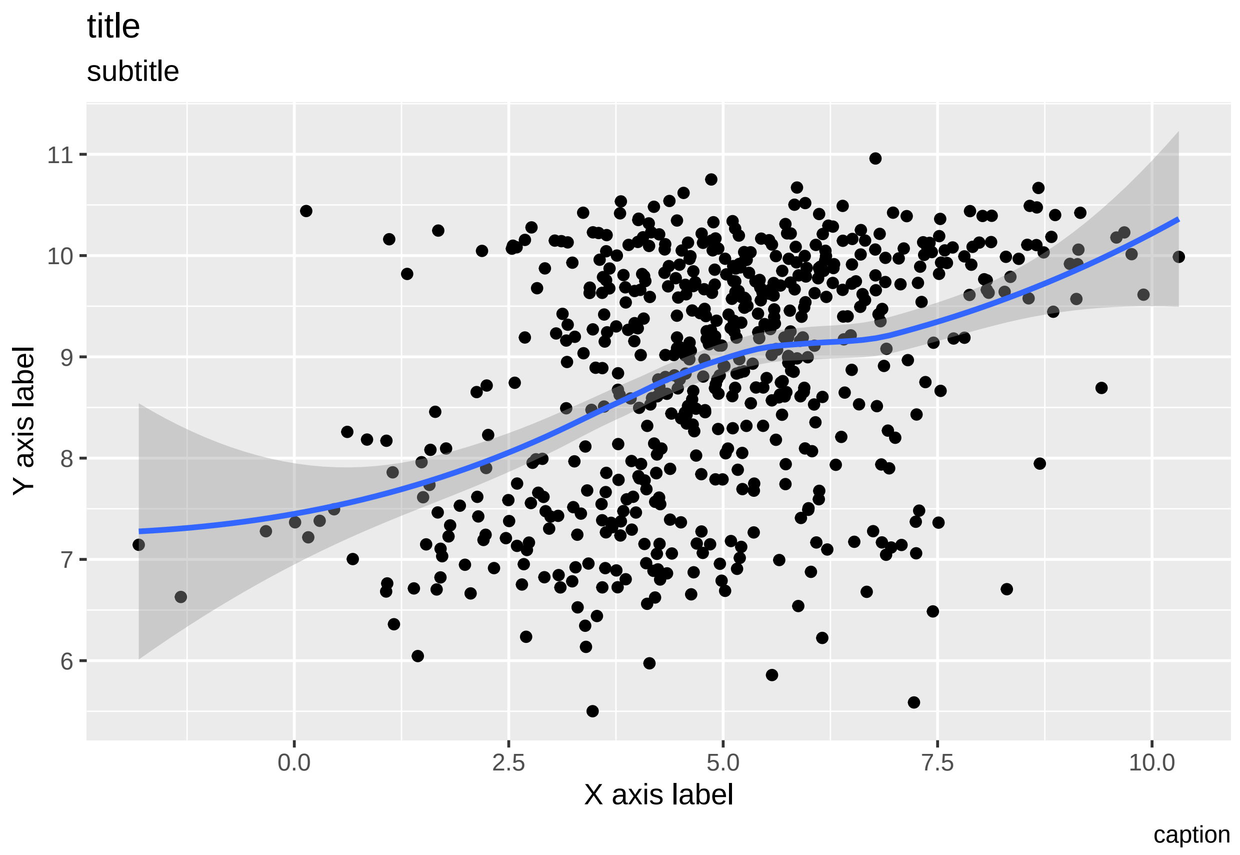
```
gg+
  geom_smooth()
```

```
## ‘geom_smooth()‘ using method = ’loess’ and formula ’y ~ x’
```



```
gg+
  geom_point()+
  geom_smooth()
```

```
## ‘geom_smooth()‘ using method = ’loess’ and formula ’y ~ x’
```



Creating gg objects will keep your code clean and easy to understand. Further, it is incredibly helpful when you need to make a series of plots that contain similar information!

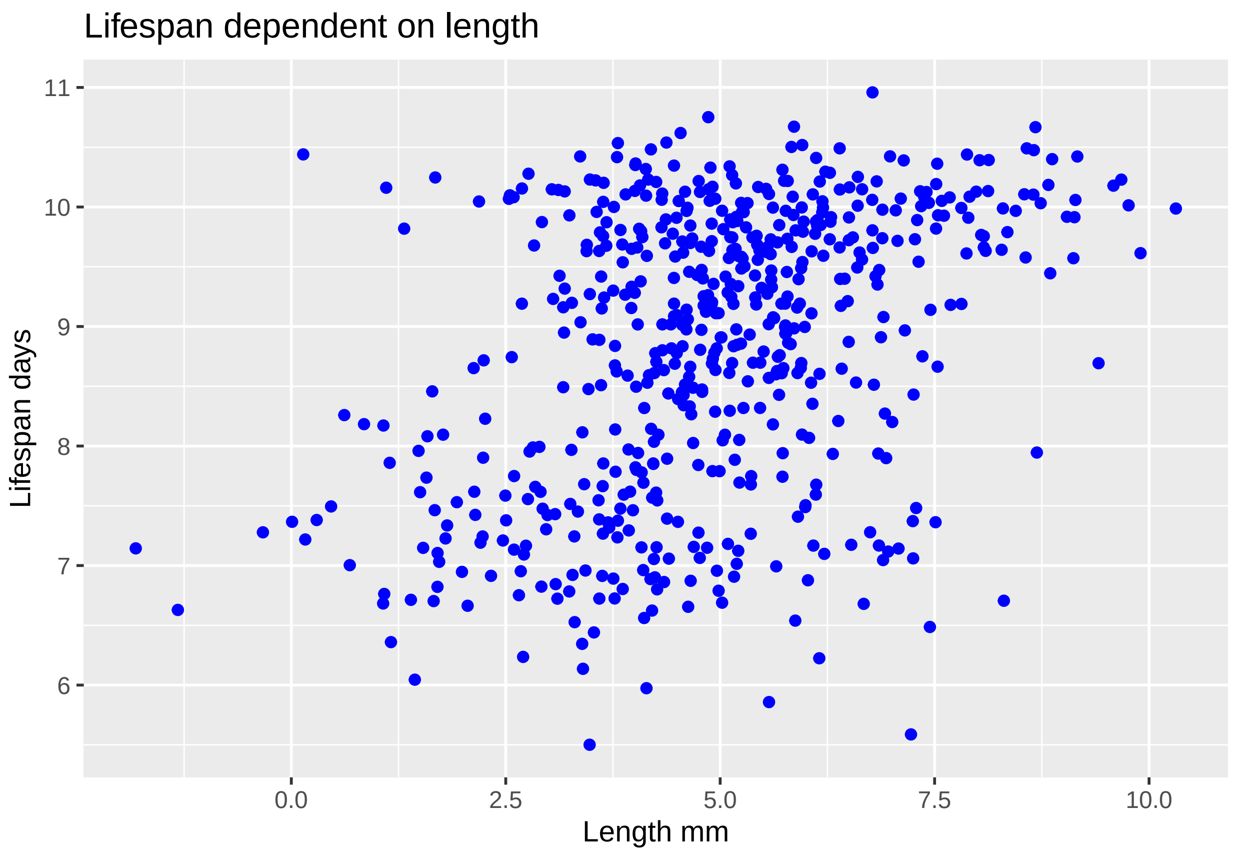
6.2.1 color and fill

---

color and fill We can change the color of our data using 2 different arguments, color and fill.

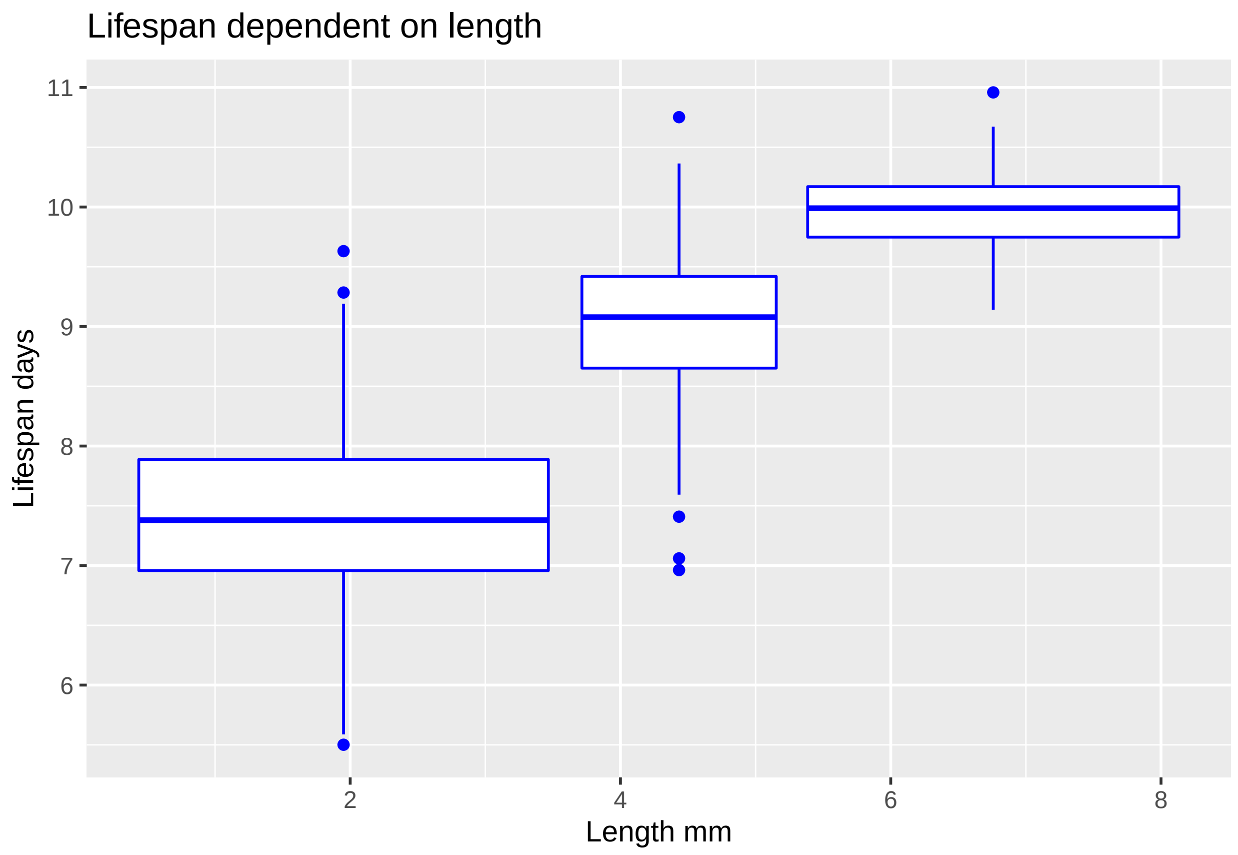
color determines the color of 1-dimensional data visualizations (points and lines)

```
#create a new gg object to be used for the lesson
gg <- ggplot(data = fruit_fly_df, aes(x = length_mm, y = lifespan_days))+
            labs(title = "Lifespan dependent on length", x = "Length mm", y = "Lifespan days")

gg+geom_point(color = "blue")
```



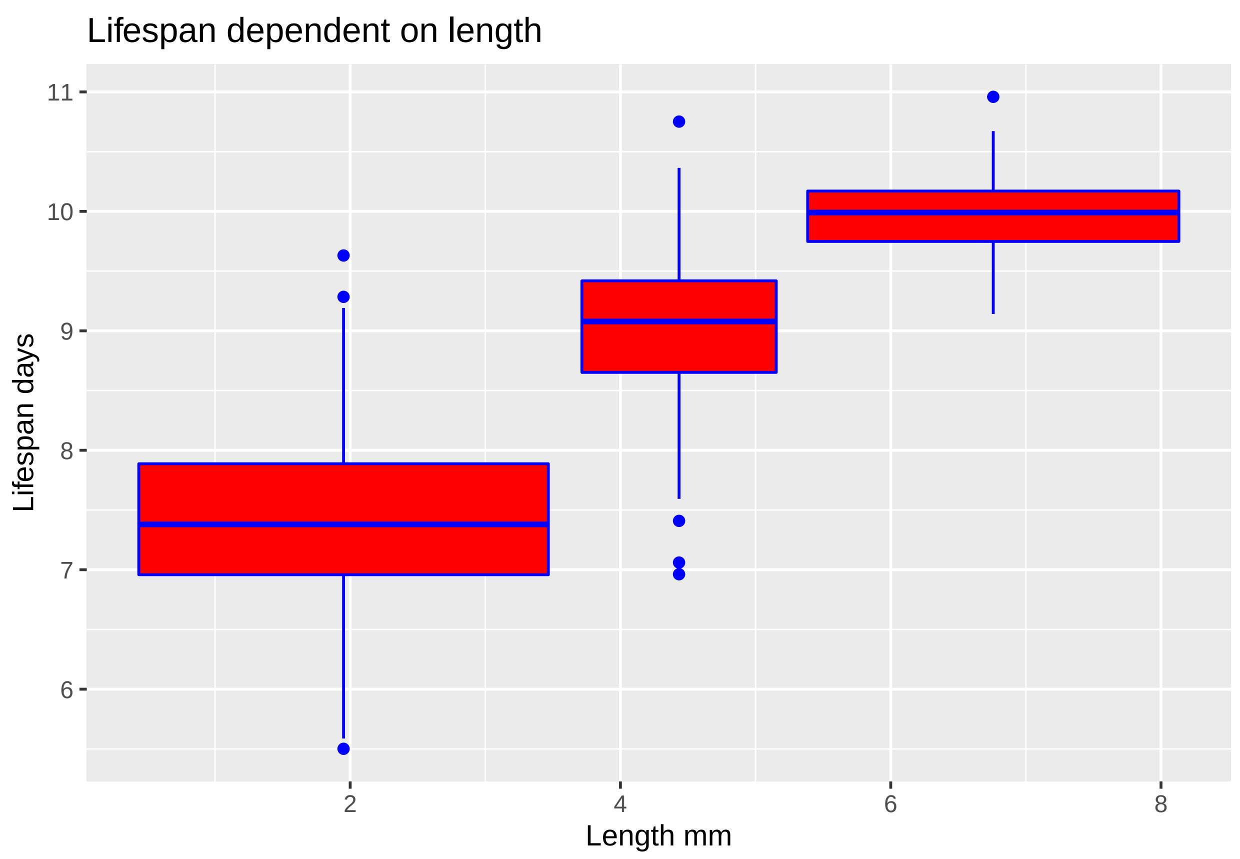
color also determines the border color of 2-dimensional data visualizations (e.g. boxplots, bar charts)

```
#To create boxplots we need to group by our grouping variable in aes().
gg+geom_boxplot(aes(group = group),
                color = "blue")
```



fill determines what color 2 dimensional data visualizations are filled in with. This example fills in the box plot red, while keeping the border color blue

```
gg+geom_boxplot(aes(group = group), fill = "red", color = "blue")
```



As a reminder, in the bonus documents section I have included resources for all the different color formats.

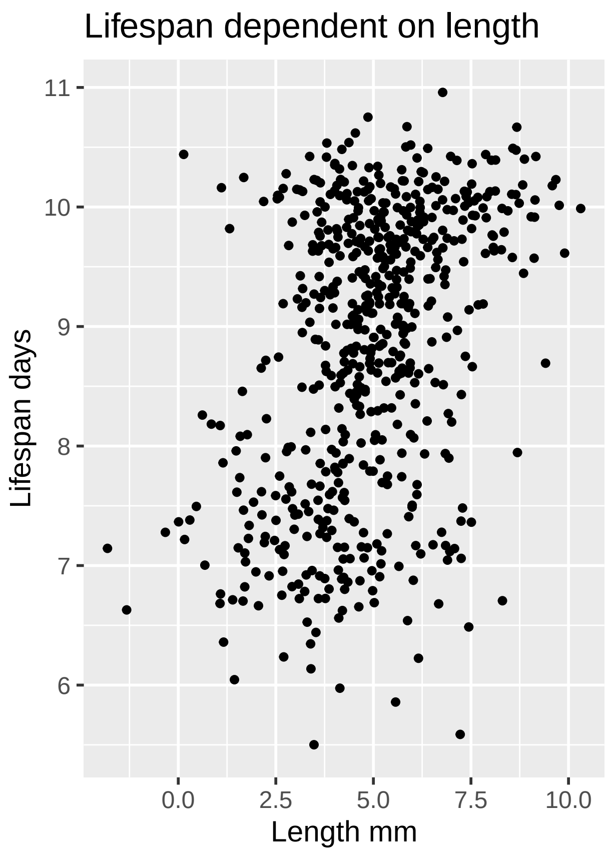
6.2.2 Size shape and transparency

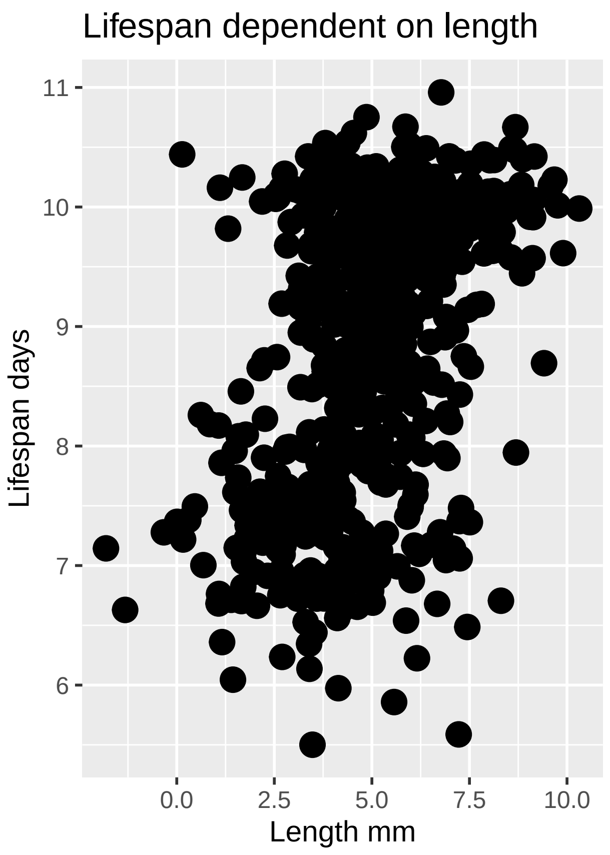
Size, Shape, Transparency, and Linetype

Color and fill are not the only arguments we can specify. This lesson will give a brief rundown of the most commonly used aesthetic arguments.

Size Size controls the size of the data points. Size takes numerical inputs in millimeters.

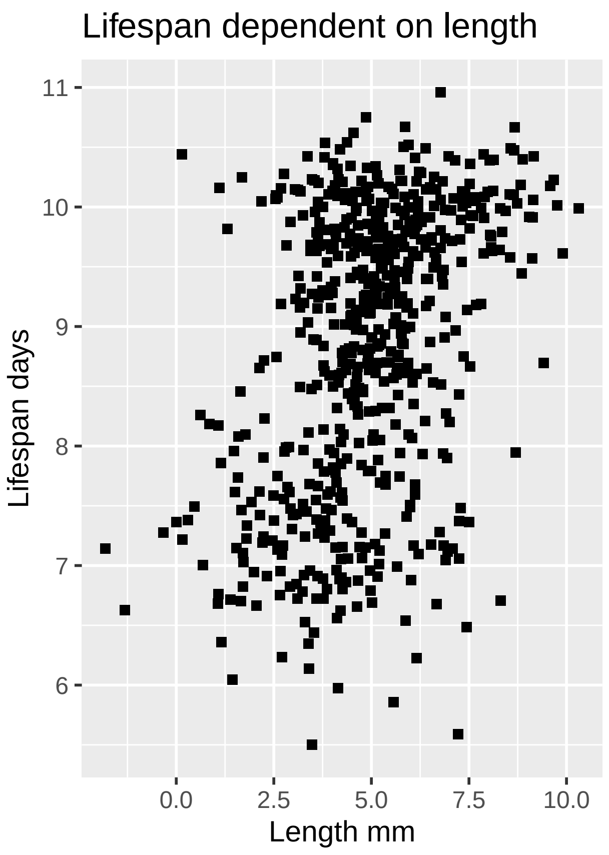
```r
#Left Plot
gg + geom_point(size = 1)

#Right plot
gg + geom_point(size = 4)
```

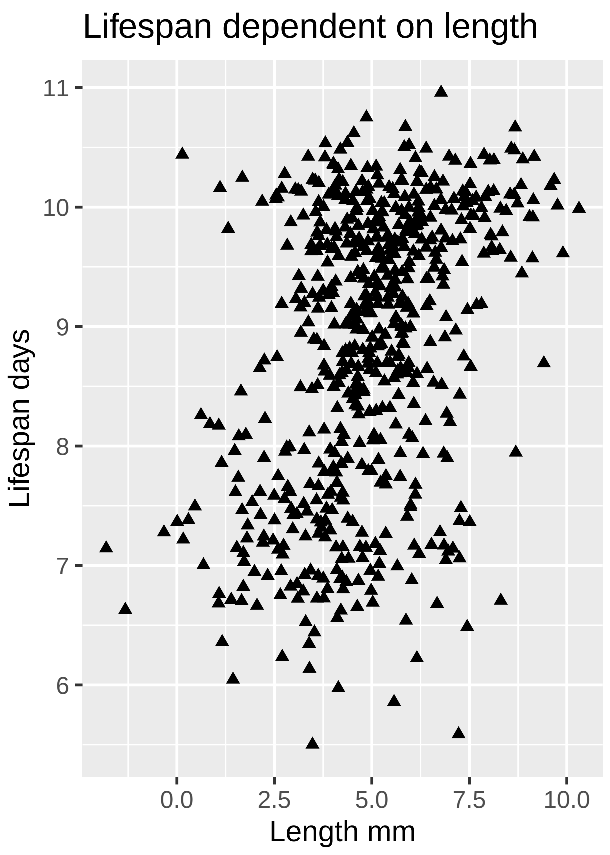




Shape Shape controls the shape of the points. Shapes can be a character value (e.g. "square") or an integer that corresponds to a shape. For example, 17 is a triangle.

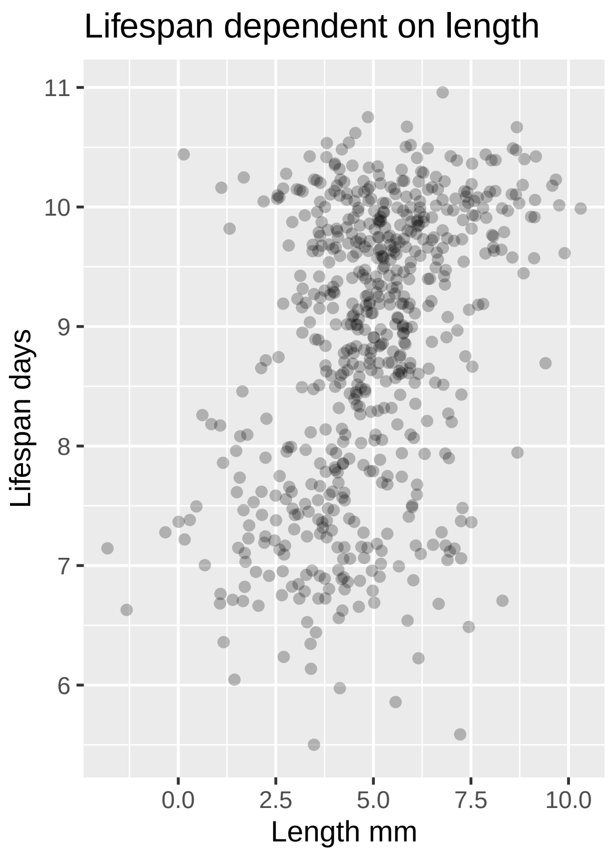
```
#Left Plot
gg + geom_point(shape = "square")

#Right Plot
gg + geom_point(shape = 17)
```

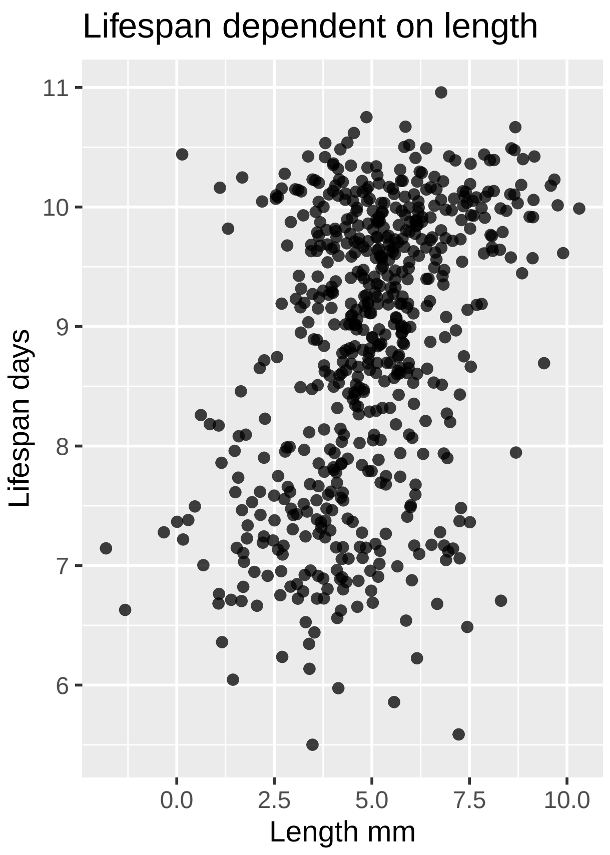




Alpha Alpha controls the transparency of the points. Alpha values are decimals on a scale of 0-1, with 1 being completely opaque, and 0 being completely transparent.

```
#Left Plot
gg + geom_point(alpha =.25)

#Right Plot
gg + geom_point(alpha = .75)
```





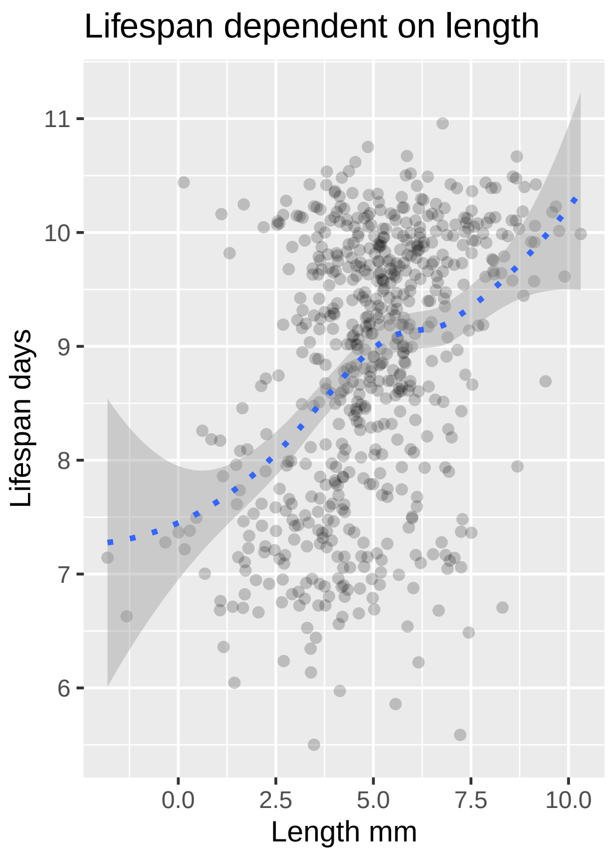
Line type When we plot lines, we can specify the type of line using the linetype argument. linetype takes in either characters or a numeric value. There are seven types linetype allows.

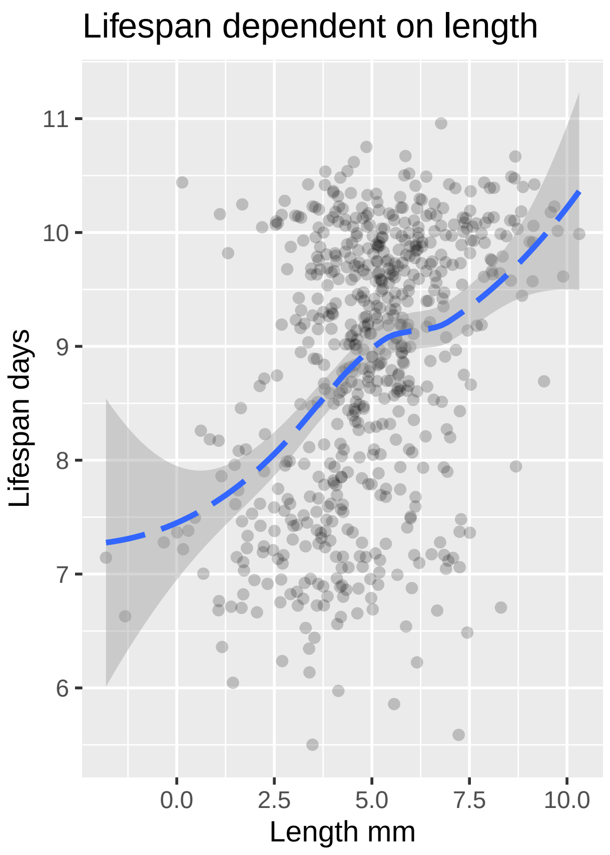
- 0 - no line
- 1 - solid ———————-
- 2 - dashed - - - - - - - - - - - - - - -
- 3 - dotted . . . .
- 4 - dotdash -.-.-.-.-.-.-.-.-.-.-.-.-.-.
- 5 - longdash – – – – – – – – – – –
- 6 - twodash – - – - – - – - – - – -

I set the alpha value low on these graphs to show the lines better.

```
gg+geom_point(alpha = .2)+
  geom_smooth(linetype = 3)

gg+geom_point(alpha = .2)+
  geom_smooth(linetype = "longdash")
```

```
## ‘geom_smooth()‘ using method = 'loess' and formula 'y ~ x'
## ‘geom_smooth()‘ using method = 'loess' and formula 'y ~ x'
```



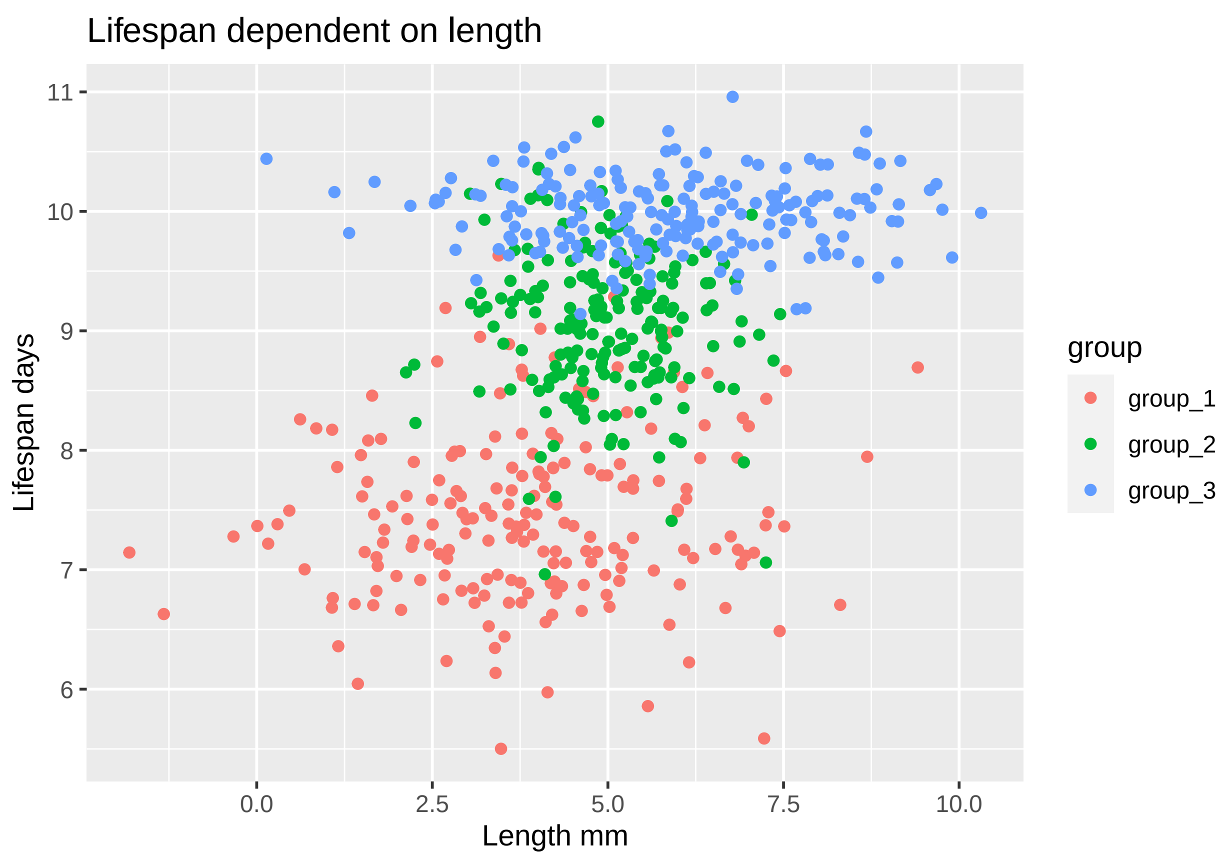


Naturally, there are many aesthetics that we wish to customize. The bonus documents "Intro the Themes" and "Intro to elements" cover in depth how to fully edit every component of your plot!

6.2.3 visualizing using data and aes

Visualizing using data and aes() Using the **aes**() function, we have the capability to let our data determine how the visualization looks. For example, lets create the scatterplot, but let the group variable determine the color.

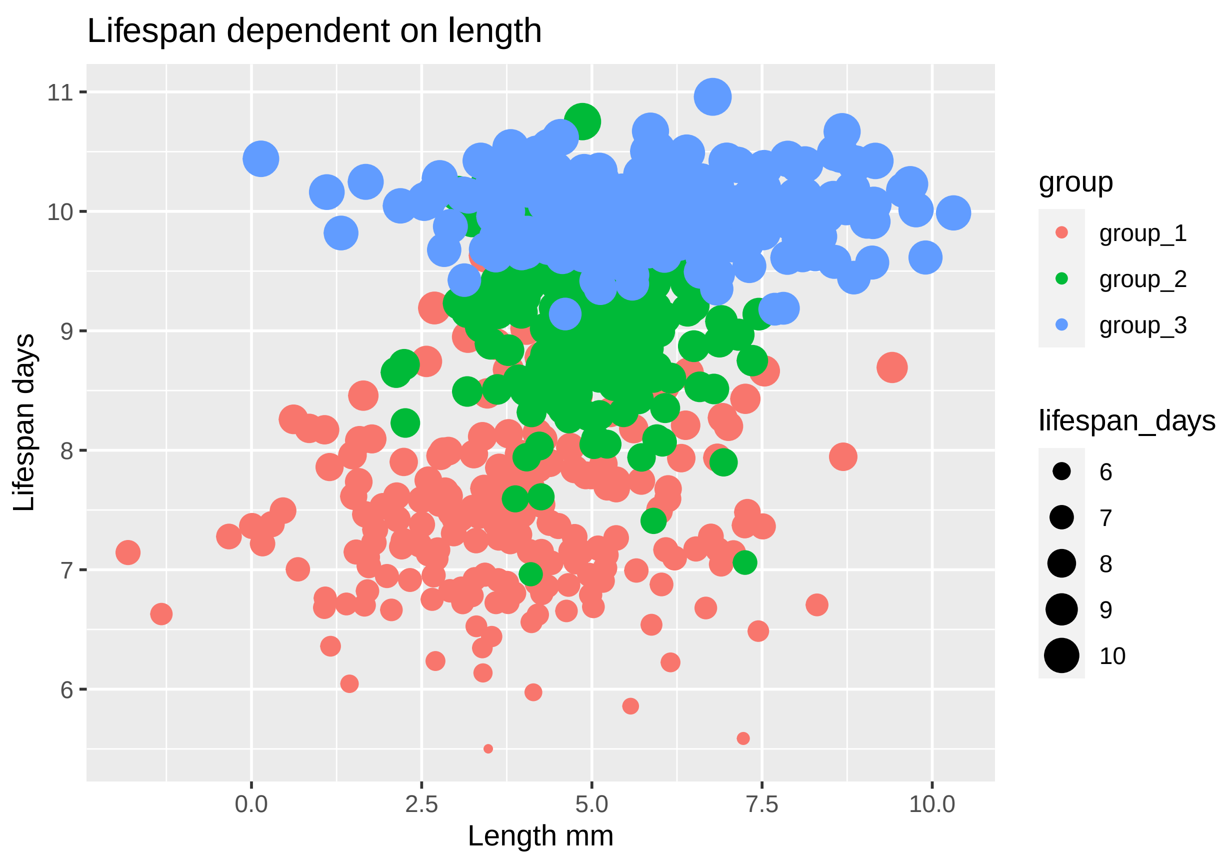
```
gg + geom_point(aes(color = group))
```



Now we can actually see some cool trends!

Naturally, we can do this procedure with any variable. Lets now determine the size of our points according to length.

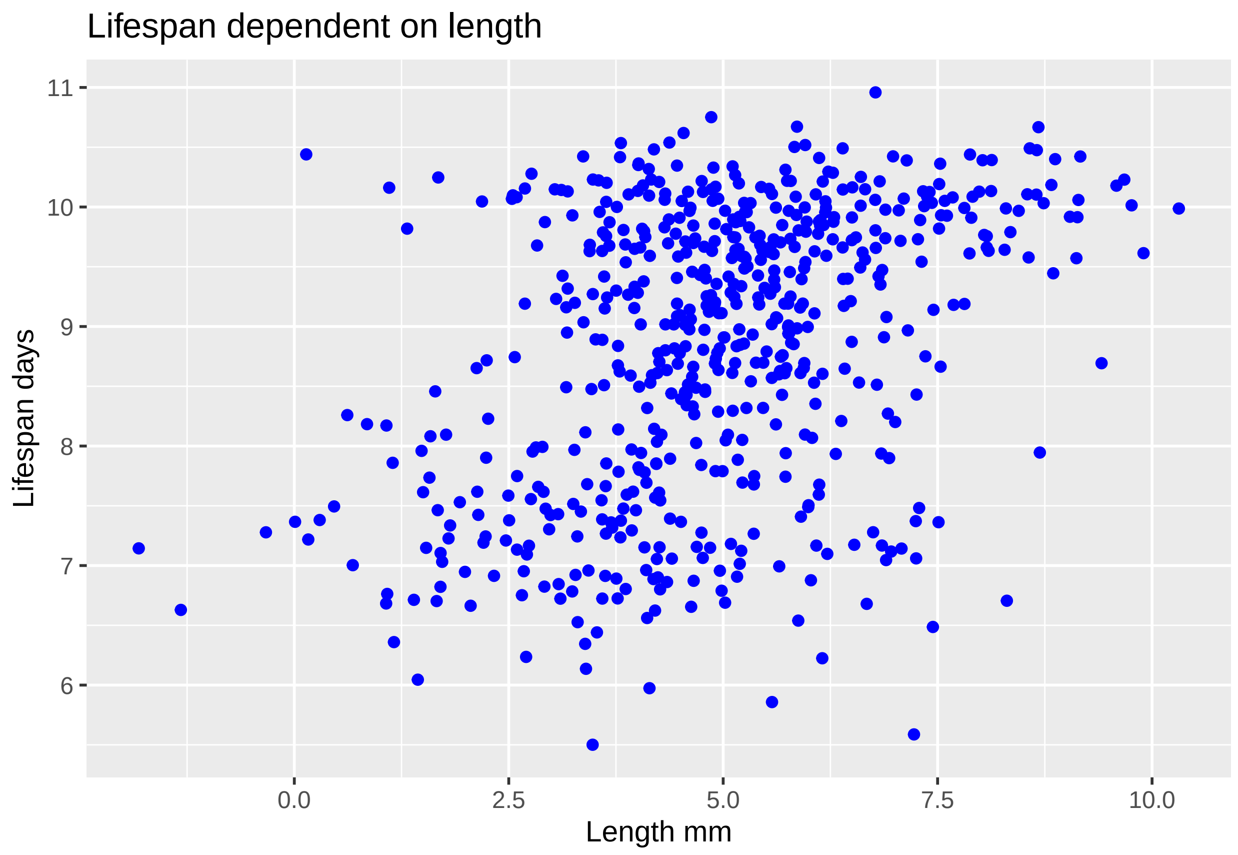
```
gg+ geom_point(aes(size = lifespan_days,
                   color = group))
```



Using **aes**() properly can lead to some stunning visualizations! However, I want to specifically point out some common mistakes when using it.

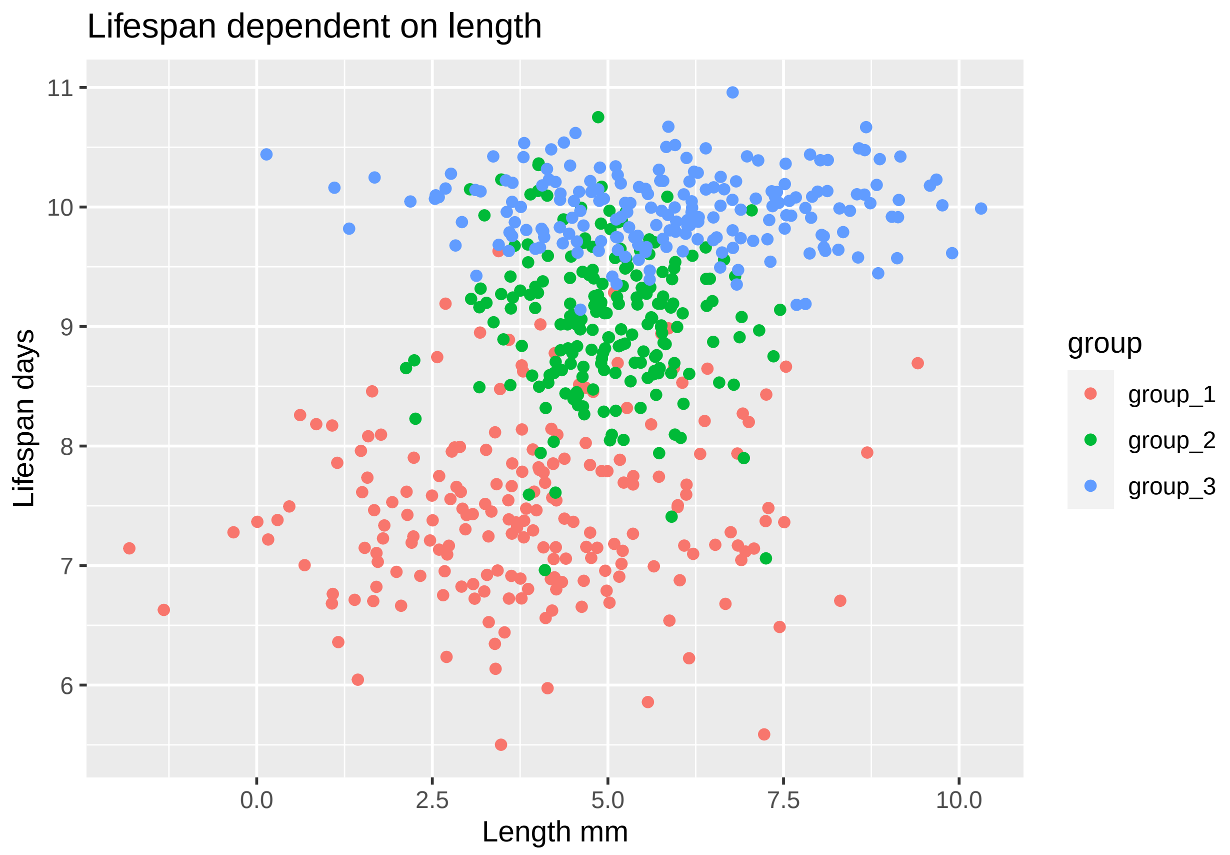
Common mistakes with aes() When we specified the color of our points without using data, we specified it ***outside*** of **aes**().

```
gg + geom_point(aes(), color = "blue")
```



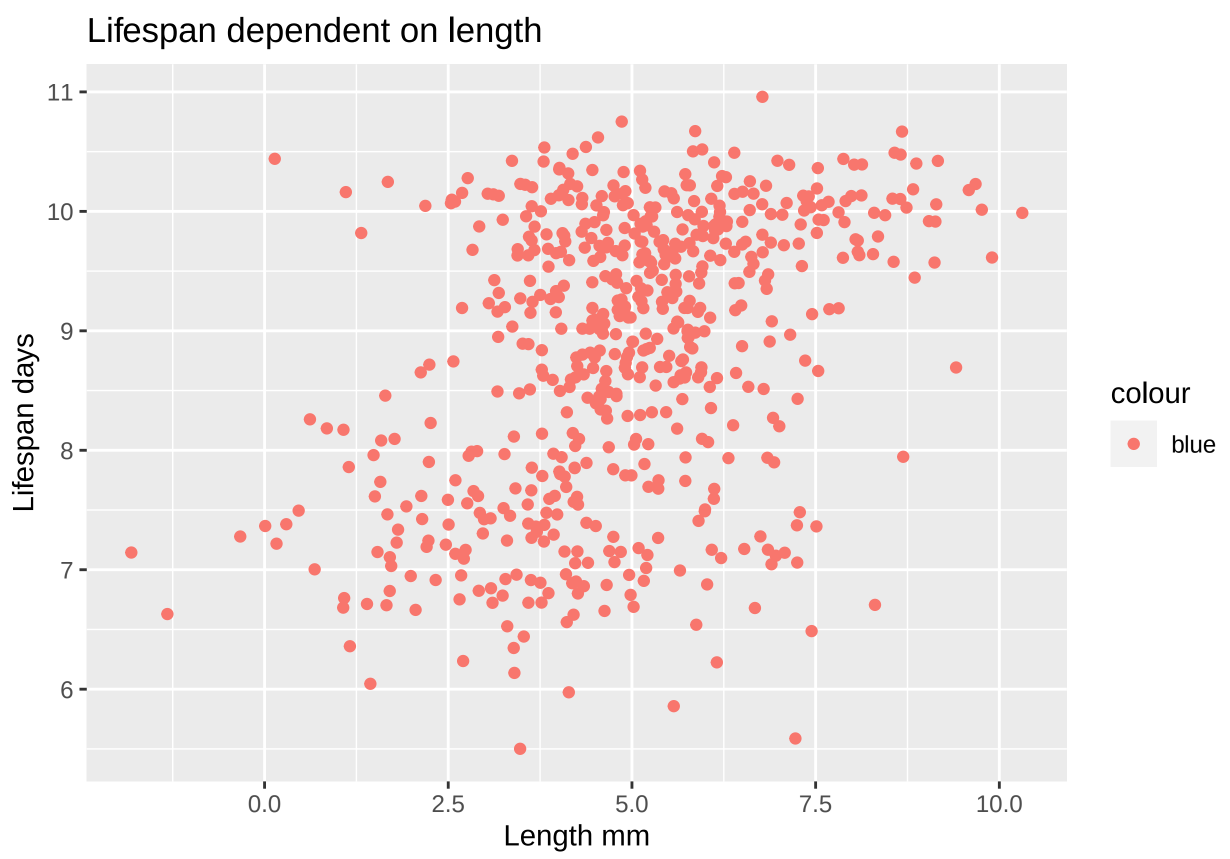
When we determined the color of our points according to our data, we placed the *color* argument ***inside*** of **aes**().

```
gg + geom_point(aes(color = group))
```



A very common mistake is placing the exact color we want to use ***inside*** of aes(). For example:

```
gg + geom_point(aes(color = "blue"))
```



As we can see, the points are not blue.

By placing **color = "blue"** inside of aes, we are telling r to visualize the data according to some variable "blue", which does not exist in our data. This is why "blue" shows up on the legend.

Red just happens to be the first default color ggplot2 uses.

Conversely, specifying data we wish to map to an aesthetic ***outside*** of aes() results in an error message.

```
gg+geom_point(color = group)
```

```
## Error in layer(data = data, mapping = mapping, stat = stat, geom = GeomPoint, : object 'group' not f
```

This can be very confusing for newcomers to ggplot2. Just remember, if we want the aesthetics to relate to data in our dataset, put the arguments in *aes*().

If we do not want to relate to our data, then keep our arguments outside of *aes*().

6.3.1 faceting with facet_wrap

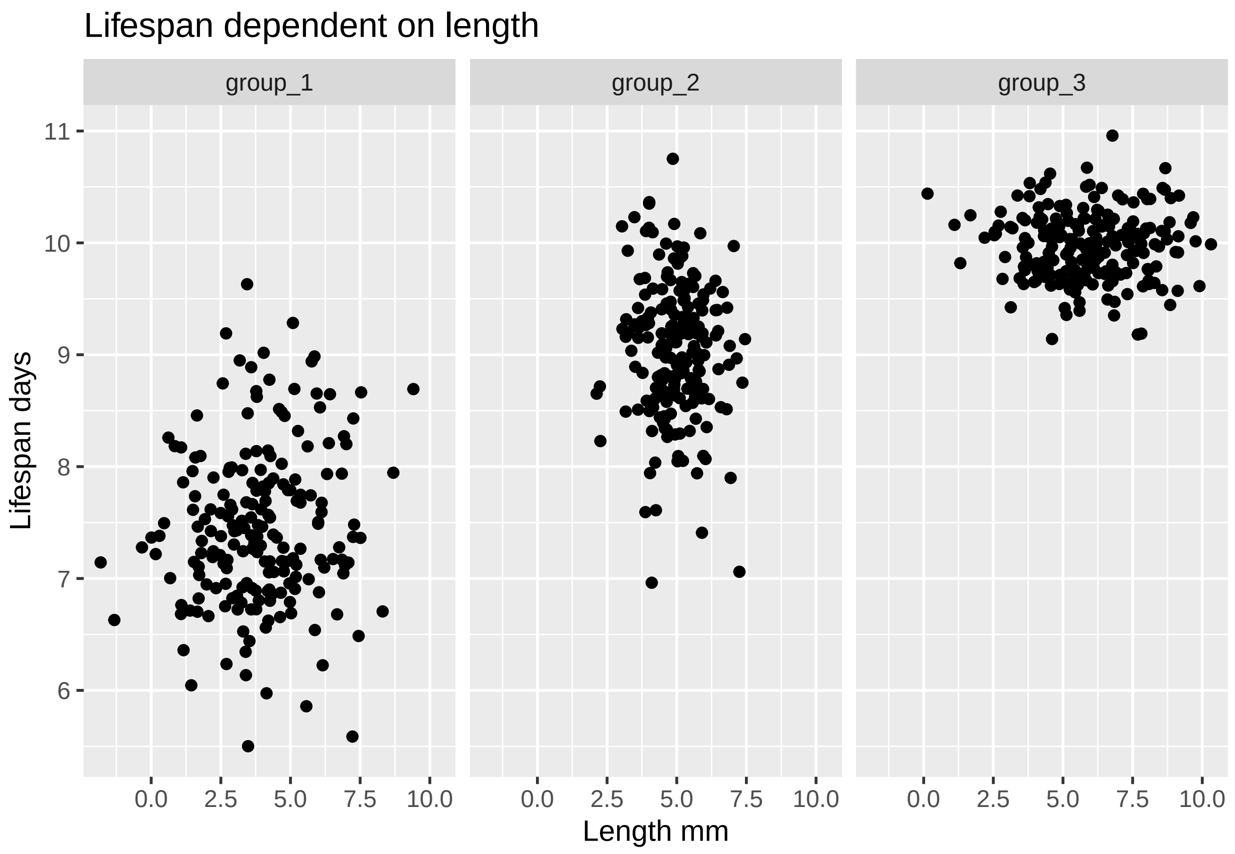
Faceting with ggplot

This section will introduce faceting. Faceting allows us to make many plots at once based on some grouping variable in our data.

Lets make a simple scatterplot, but this time including **facet_wrap**(). We'll split our data according to the 'group' variable.

Here we need to include the tilde (~) in order for facet_wrap to work correctly.

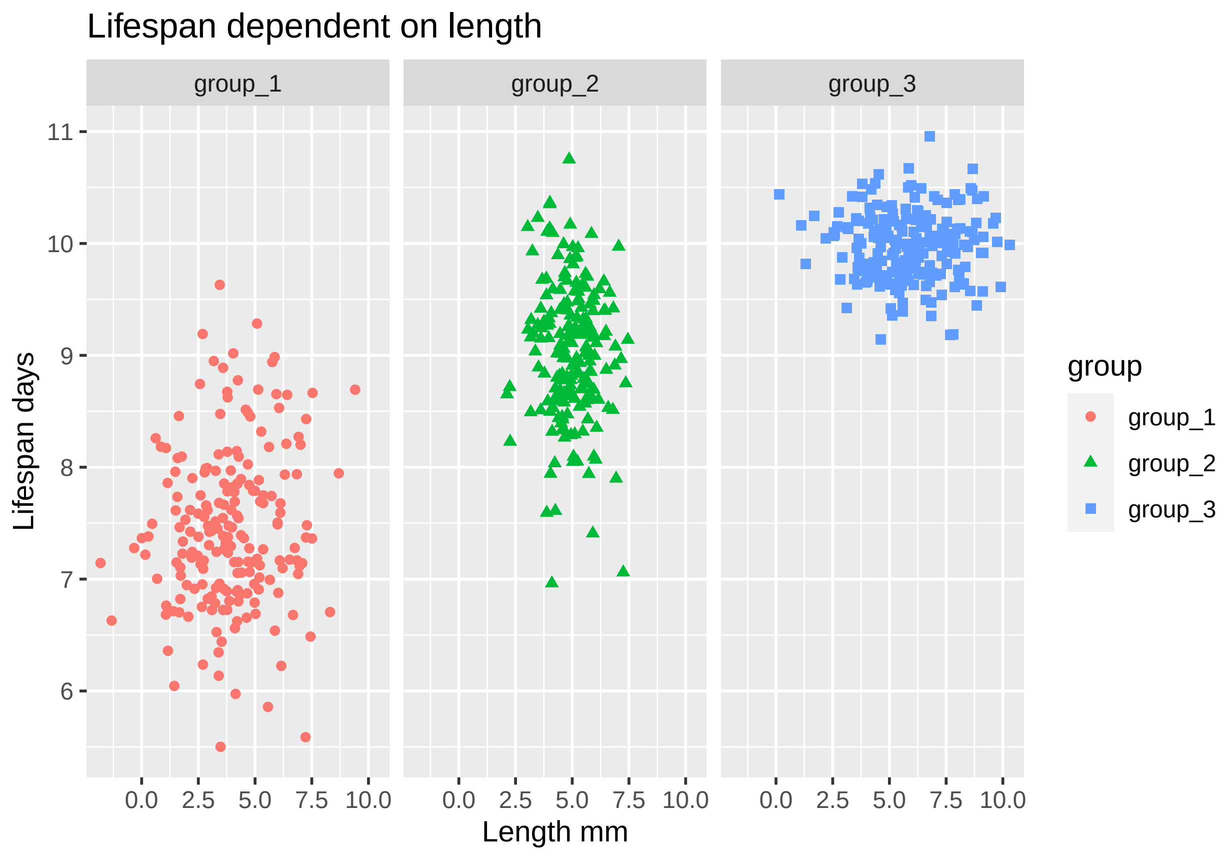
```
gg +
  geom_point()+
  facet_wrap(~group)
```



We now have three plots, one for each group!

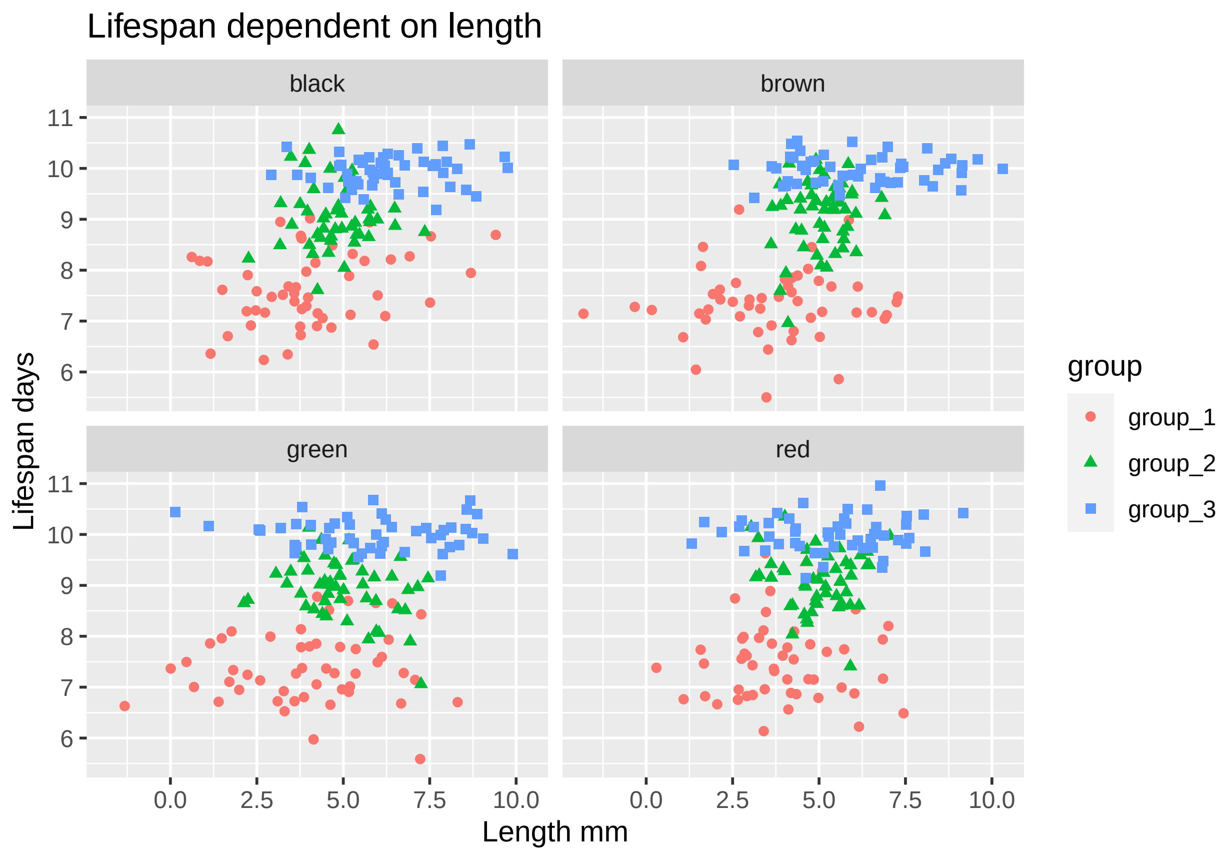
We can also supply color and shape using group into aes().

```
gg +
  geom_point(aes(color = group, shape = group))+
  facet_wrap(~group)
```

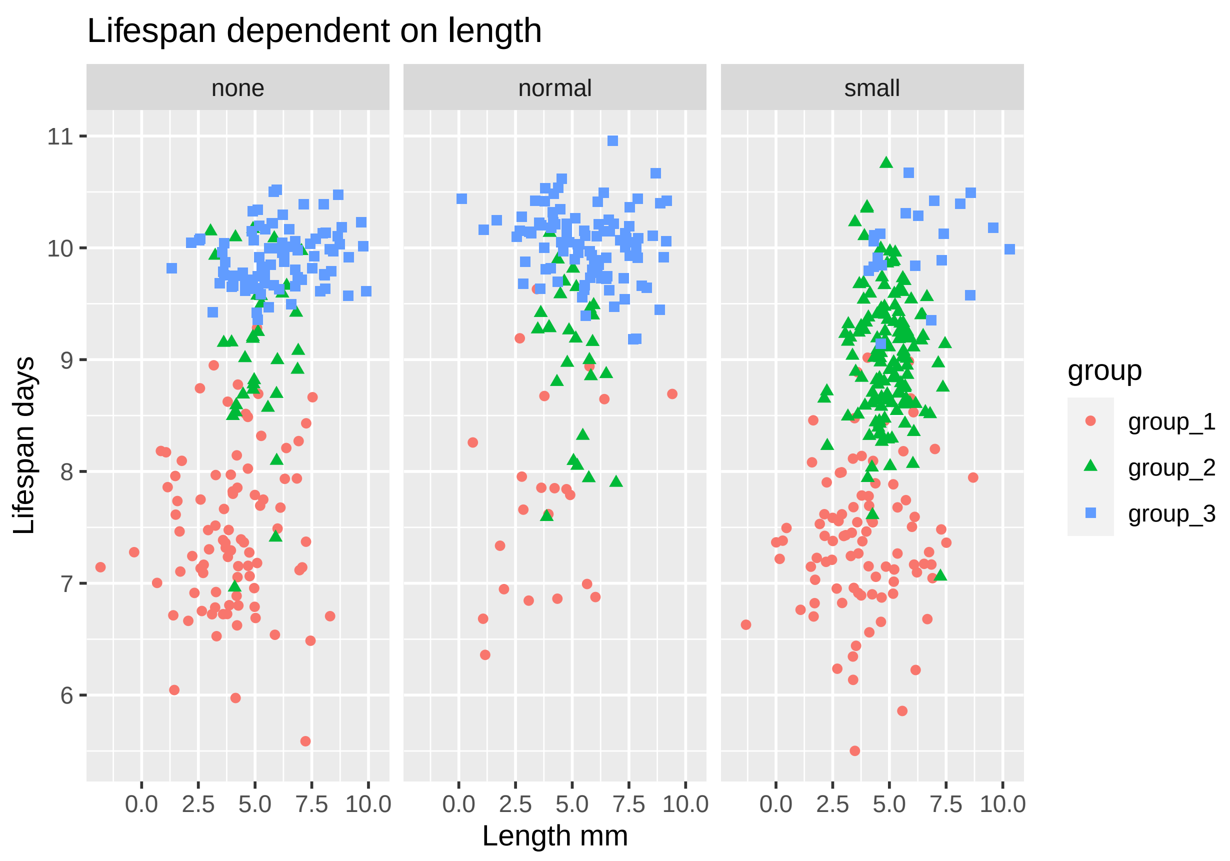


Facet according to 2 variables We can of course facet on another grouping variable, such as eye color or wing size

```
gg +
  geom_point(aes(color = group, shape = group))+
  facet_wrap(~eye_color)
```

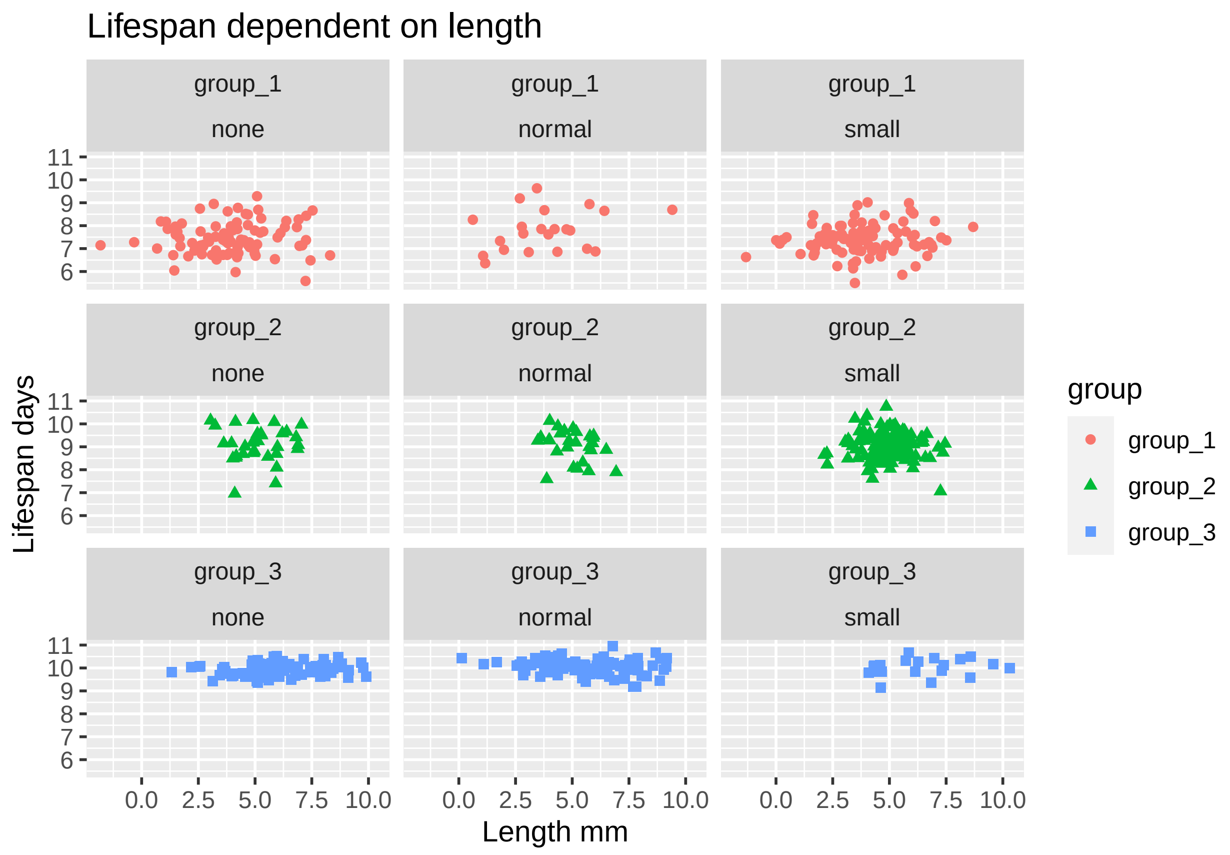


```
gg +
  geom_point(aes(color = group, shape = group))+
  facet_wrap(~wing_size)
```



However, If we want to facet according to 2 variables, we can do so by placing another variable to the left of the tilde (~). Lets split it between group as well as wing size.

```
gg +
  geom_point(aes(color = group, shape = group))+
  facet_wrap(group~wing_size)
```



The variables on either side of the tilde (~) correspond to either rows or columns with the format being **rows~columns**. For example, the above plot makes the groups the rows and the wing size the columns (group ~ wing_size).

Faceting is an incredible useful tool in ggplot and should be in any data analysts toolkit!

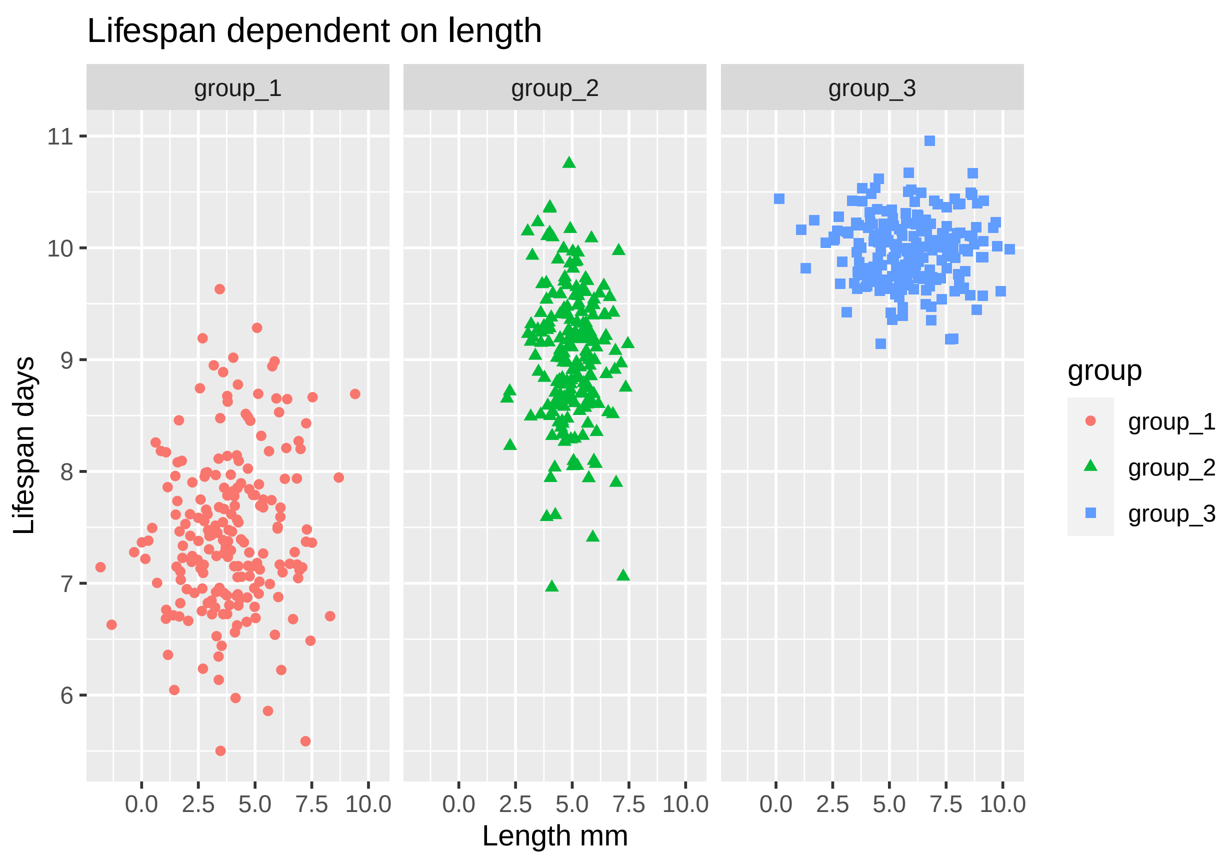
6.3.2 saving visualizations using ggsave

Saving visualizations using ggsave()

Saving visualizations created in ggplot is easy using the ggsave function.

Lets create a visualization and save it to the object "output".

```
output <- gg +
  geom_point(aes(color = group, shape = group))+
  facet_wrap(~group)

output
```



We can now use ggsave() in a very similar manner as the save() function we learned a few weeks ago. Lets save our figure as a png image.

```
ggsave(output, filename = "BestVisualization.png")
```

```
## Saving 6.5 x 4.5 in image
```

Again, ensure you input the file extension (in this case .png) when saving your plot. You can also save your plot as a PDF file (.pdf) by changing the file extension.

```
ggsave(output, filename = "BestVisualization.pdf")
```

```
## Saving 6.5 x 4.5 in image
```

As a reminder, the plots will save to your currently set working directory.

Bonus: Intro to ggplot themes

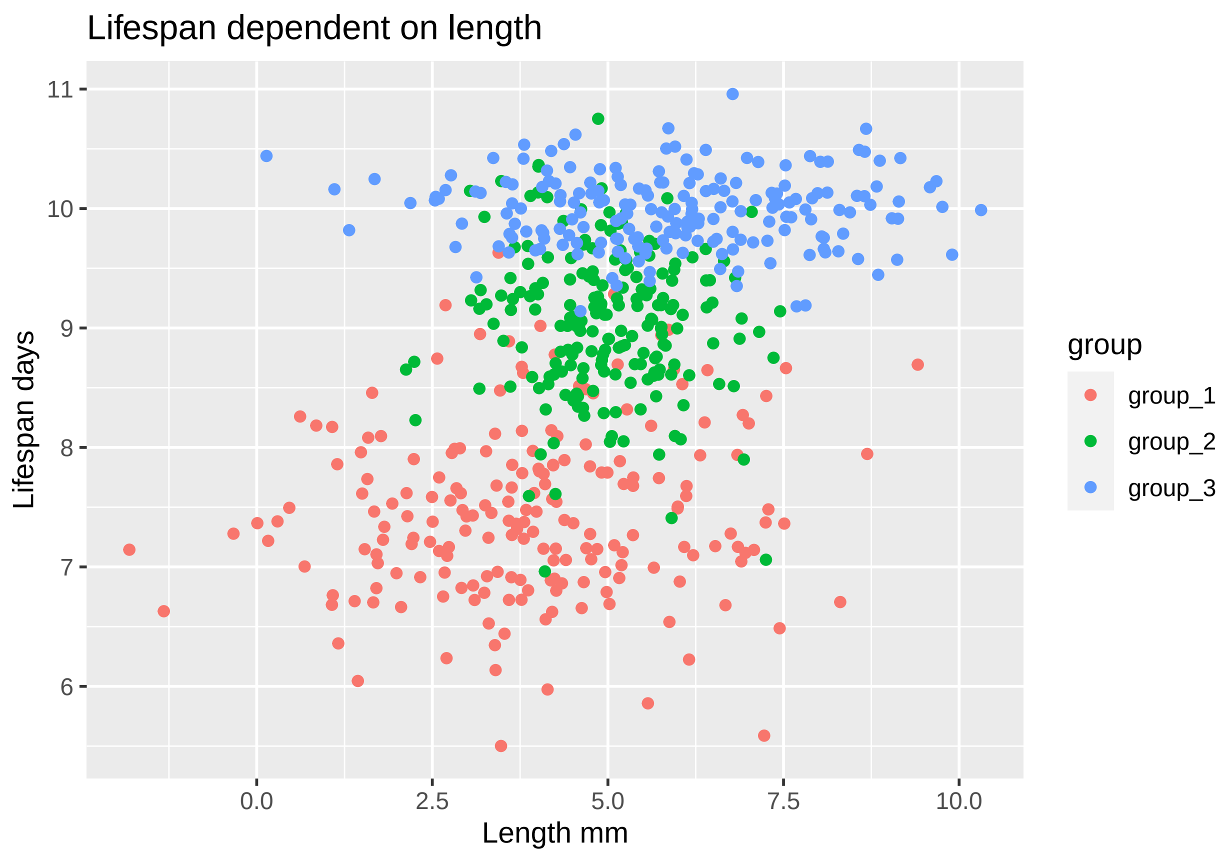
Intro to ggplot themes

When we want to customize specific aspects of how our ggplot looks, we will often want to edit the theme. The theme controls the overall appearance of non-data elements in our plot (axes, titles, legend etc.).

Luckily, there are a wide variety of built in themes (as well as many packages that add themes).

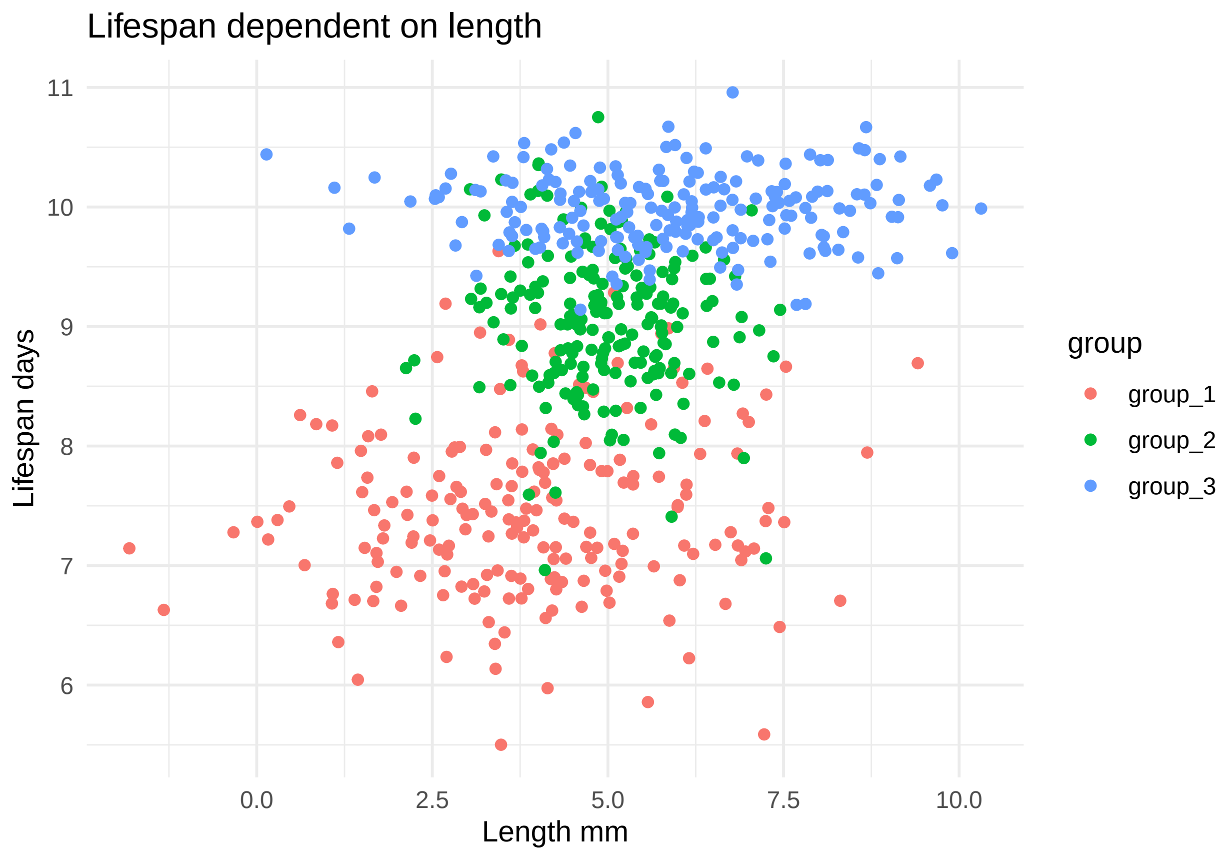
Lets make our scatter plot, coloring by group and use a few different themes. First the normal plot

```r
gg+
  geom_point(aes(color = group))
```



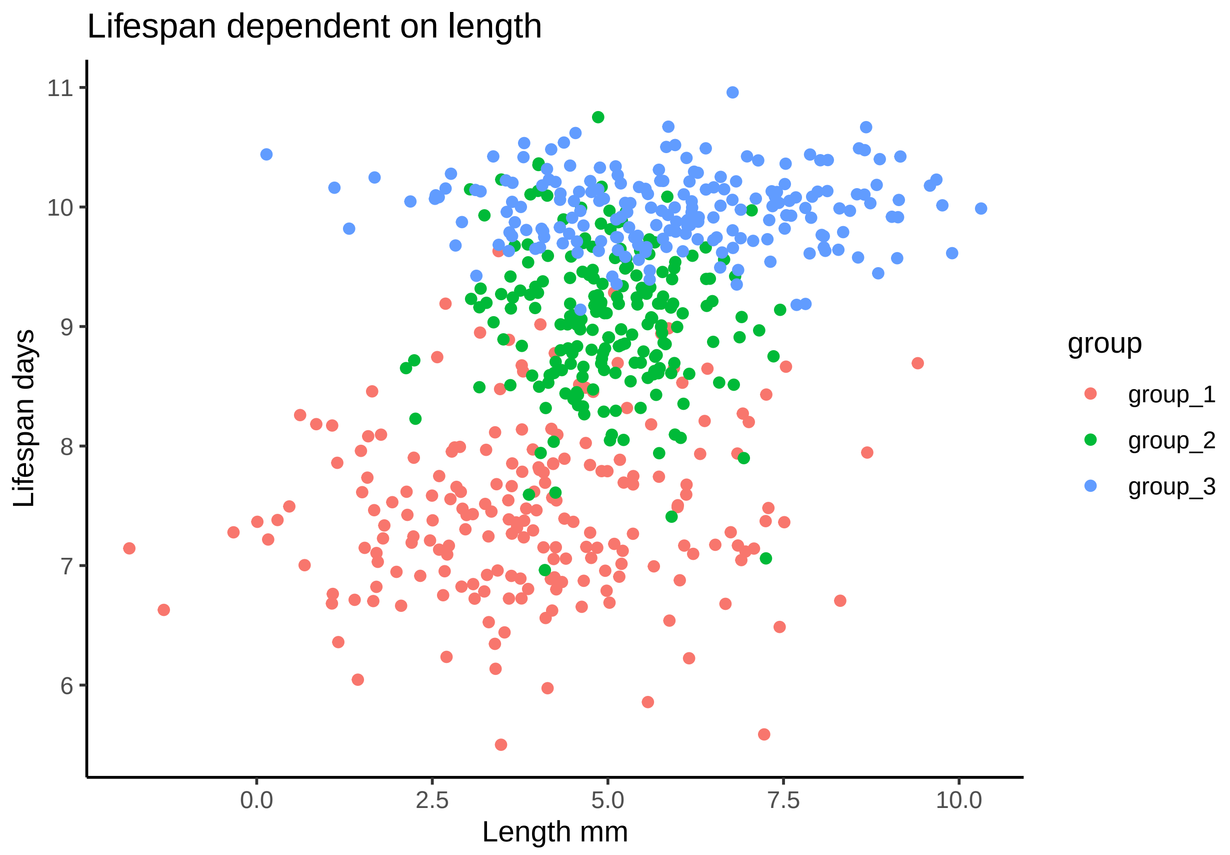
The minimal theme is quite common. Stripping away most unneeded elements

```r
gg+
  geom_point(aes(color = group))+
  theme_minimal()
```



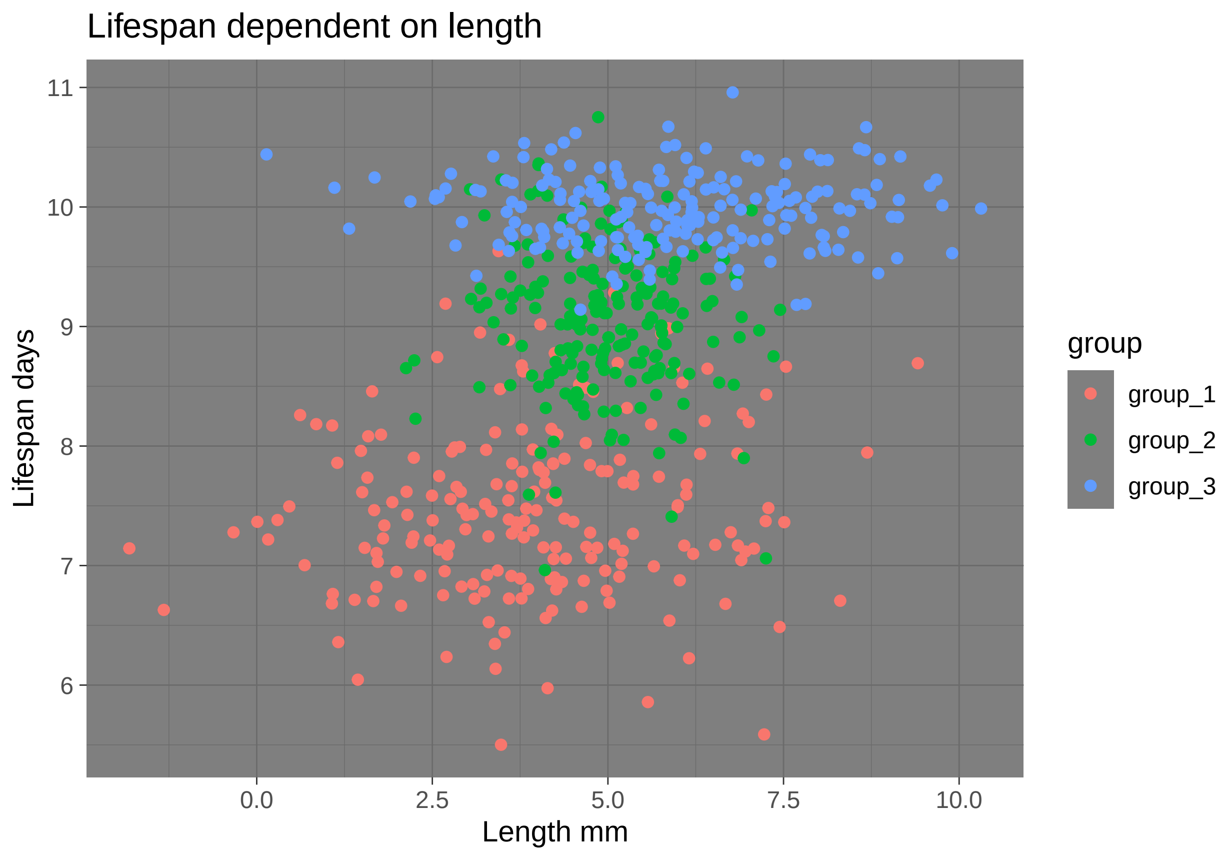
A classic theme is theme_classic()

```
gg+
  geom_point(aes(color = group))+
  theme_classic()
```



To add in something totally different, we'll show the dark theme.

```
gg+
  geom_point(aes(color = group))+
  theme_dark()
```



These simple themes can dramatically speed up your data visualization procedure.

Luckily, we can also edit every single component of the theme using the theme() function. Every component of the theme is broken into elements that we can edit. For example, on the x-axis we can edit the elements that relate to the tick-marks, the data labels, and the axis title.

The next document will cover how to edit each of those elements!

Bonus: customizing themes with elements

Customizing themes with elements

Regardless of which base theme we start with, we can fully edit every element using the **theme**() function.

Elements allow us to specify particular aspects of our ggplot. Lets start with **element_text**(), which controls the size, color, position, and just about everything related to text.

First lets create a gg object of our faceted plot from before.

```
gg <- gg +
  geom_point(aes(x = length_mm, y = lifespan_days, color = group, shape = group))+
  geom_smooth(aes(x = length_mm, y = lifespan_days))+
  facet_wrap(group~wing_size)
```

There are nearly 100 different elements we can edit within **theme**(). Naturally, we do not need to specify the format for every single element, only the ones we wish to change.

The full list of editable elements as well as their element class (explained below) can be found via the help documentation for **theme**() (?**theme**).

This lesson will focus on how to use the 4 main element classes to edit any of those 100 elements.

Elements When we edit a component of our plot, we must use one of the element functions to make this happen.

The most common element classes include:

- **element_text**() for text such as axis labels or the title.
- **element_rect**() for rectangular elements such as the plot background,
- **element_line**() for line elements such as the axis line.
- **element_blank**() which returns a blank element. This is useful for removing elements all together.

Each of these functions contain a variety of arguments that let you specify the exact format of the element.

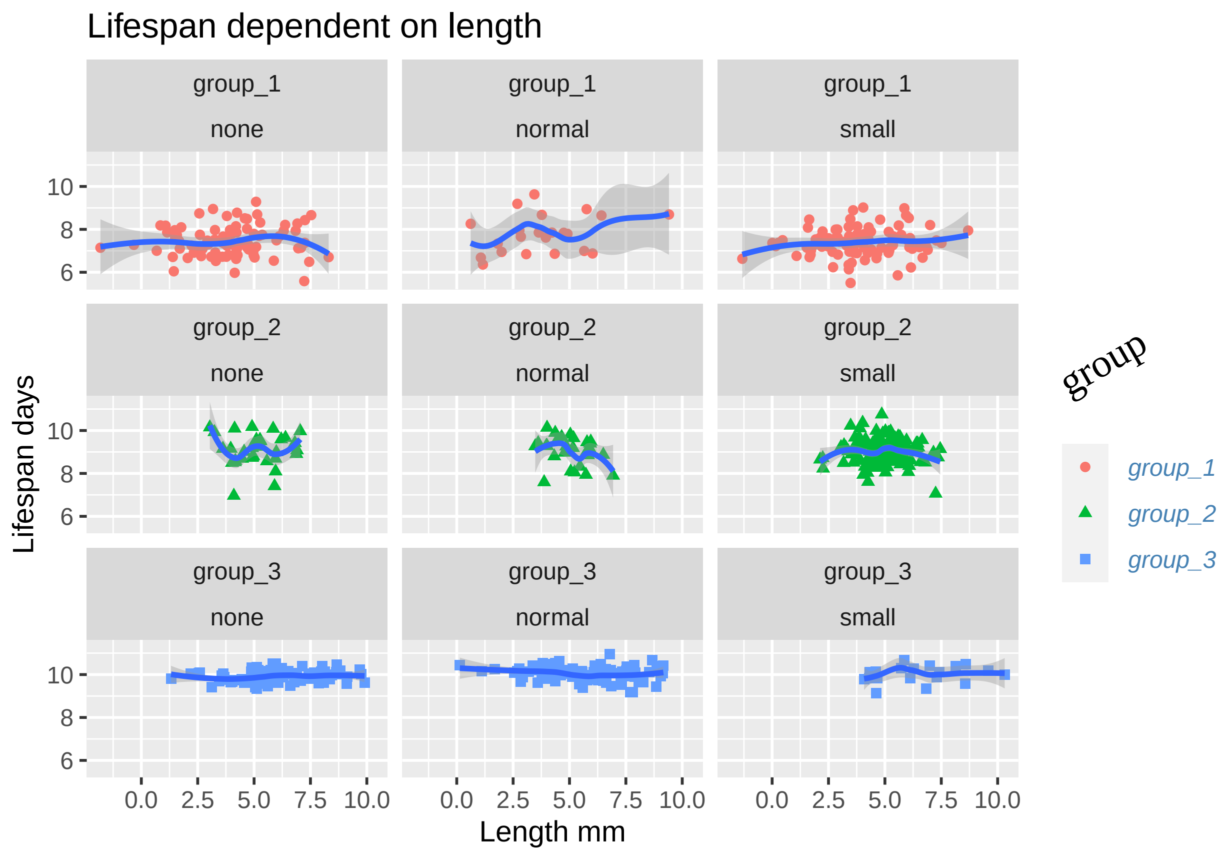
element_text() element_text() allows us to edit text elements. A few common arguments we specify include:

- family = what font to use
- face = controls bolding, italics, underlining etc
- color = font color
- size = font size
- angle = the angle of the text

Lets change the text and the title of our legend using these arguments

```
gg +  theme(legend.title = element_text(size = 15, angle = 30, family = "serif"),
            legend.text = element_text(face = "italic", color = "steelblue"))
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



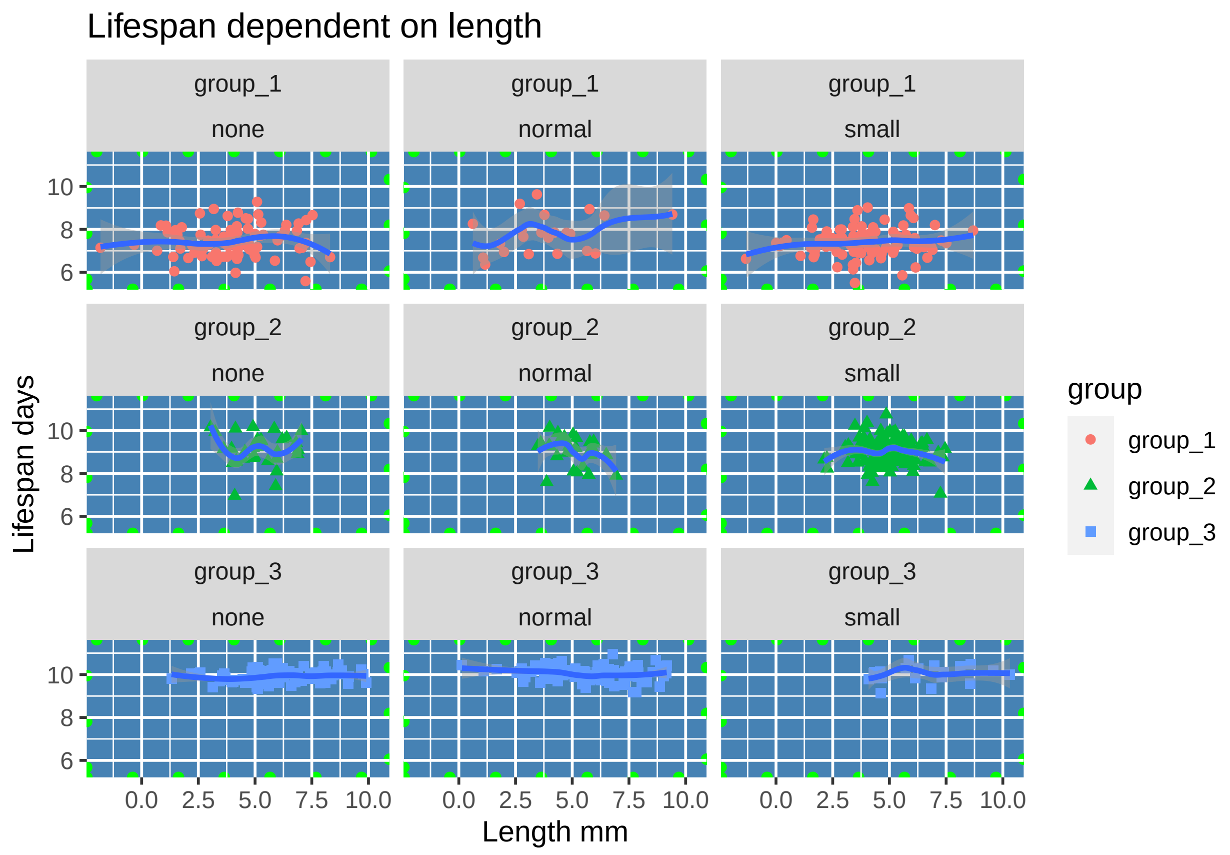
element_rect() Rectangular elements include backgrounds and borders. The main arguments include:

- fill =The color of the fill
- color = the color of the border
- size = the size of the border line
- linetype = specifies the format of the border line (solid, dashed, dotted etc.)

Lets change the format of the panels using element_rect()

```
gg +
  theme(panel.background = element_rect(color = "green",
                                        linetype = "dotted",
                                        size = 2,
                                        fill = "steelblue"))
```

```
## ‘geom_smooth()‘ using method = ’loess’ and formula ’y ~ x’
```



Well its an ugly plot but its a plot nonetheless!

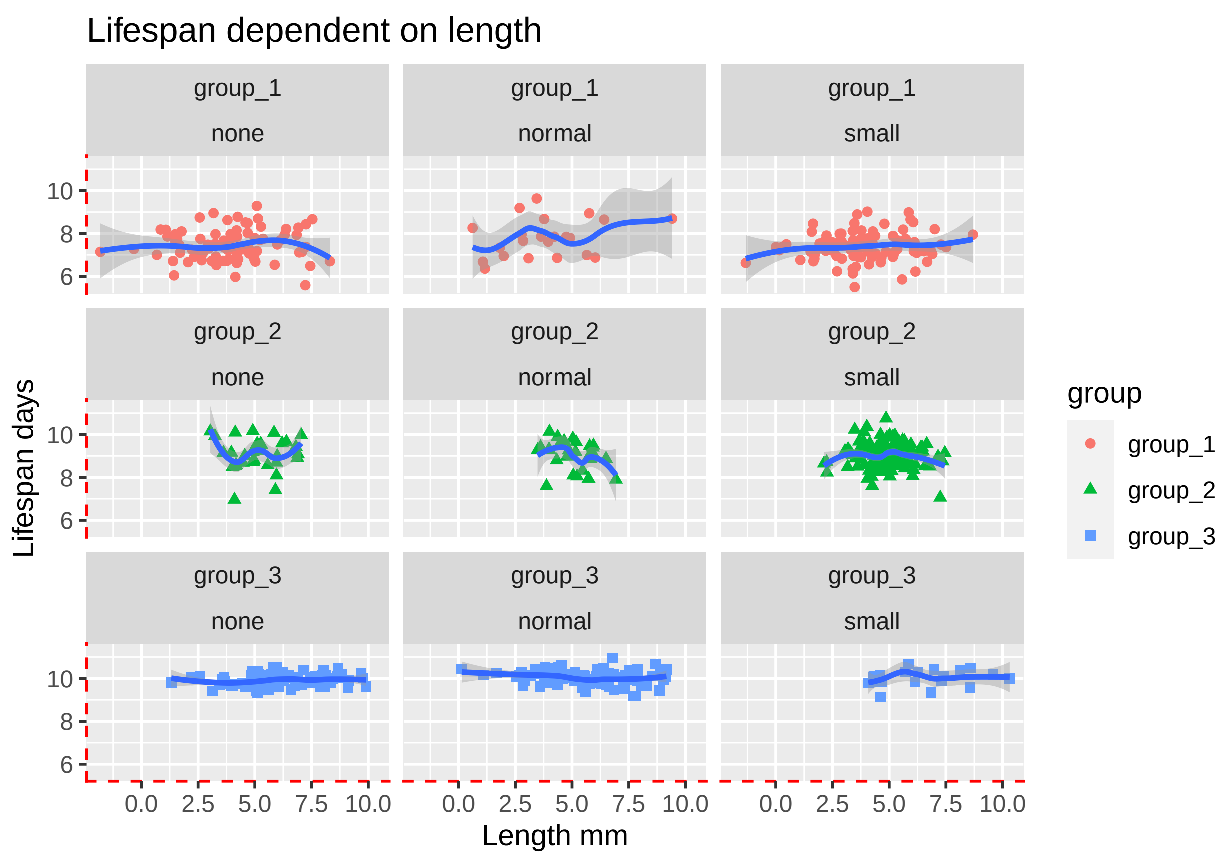
element_line() element_line determines the format of line elements. Common arguments include:

- color = the color of the line
- size = size of the line
- linetype = format of the line (solid, dashed, dotted etc.)
- lineend = style of the end of the line (round, butt, square)
- arrow = Arrow specifications using the arrow() function

Here lets edit the axis lines using element_line()

```
gg+ theme(axis.line = element_line(color = "red",
                                   size = .5,
                                   linetype = "dashed",
                                   lineend = "square"))
```

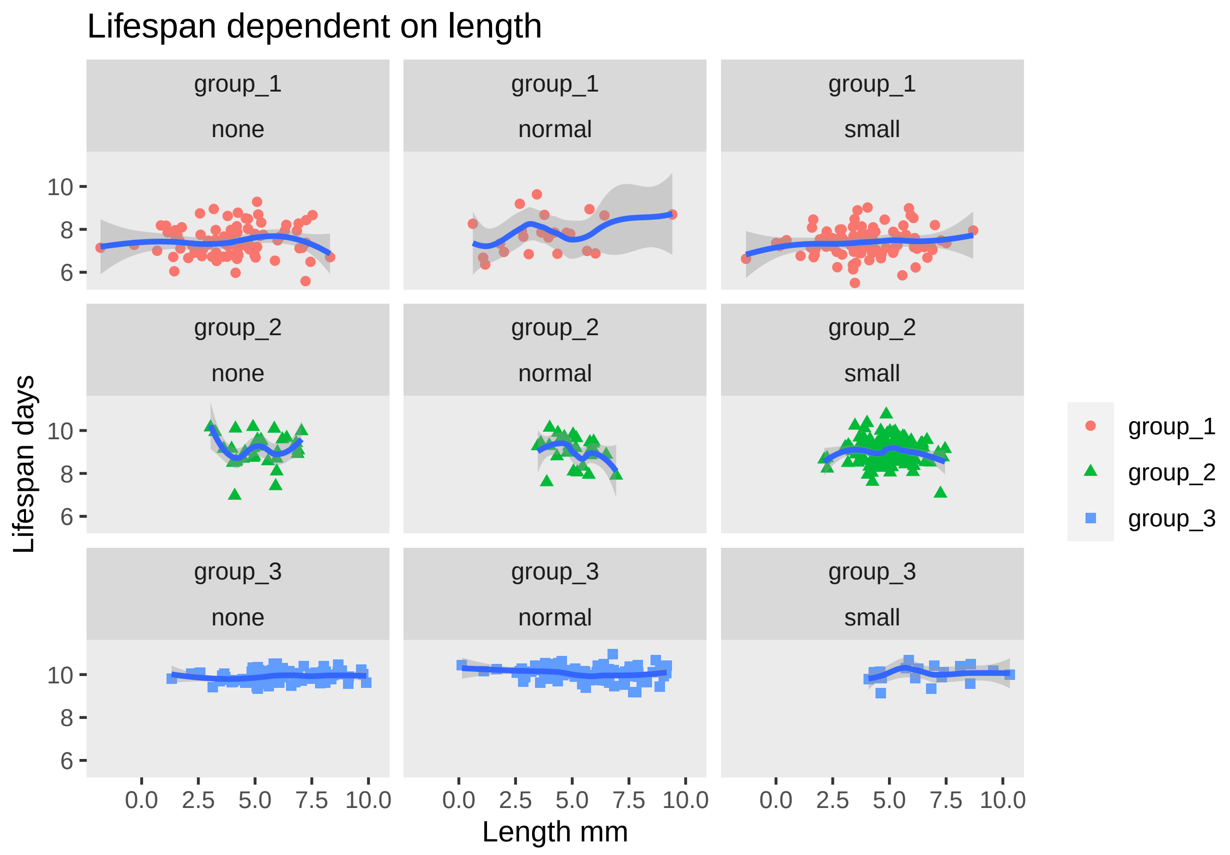
```
## ‘geom_smooth()‘ using method = ’loess’ and formula ’y ~ x’
```



element_blank() element_blank() takes no arguments but allows us to remove specific elements. Lets use this to remove the legend title and the grid lines in each panel.

```
gg+
  theme(legend.title = element_blank(),
        panel.grid = element_blank())
```

```
## ‘geom_smooth()‘ using method = ’loess’ and formula ’y ~ x’
```



As we can see, using themes and elements allow us to edit any aspect of our ggplot to our specifications! This week should have made you comfortable with working in ggplot and making the edits you need to make!