One code to rule them all (_) O

Jared Adam

2023-06-06

Table of contents

Welcome to an introductory R course for natural scientists

One Code To Rule Them All

This page was originally built to act as an introduction to R and R studio. What it has become is that, plus a little extra. This page will be updated as new information/ lessons are requested. The link will not change, so no trouble.

We suggest starting from the beginning (even if you have used R before) and working through each section. When appropriate, previous sections will be referenced, sometimes linked, and we encourage you to use previous sections when needed. Along with this, there are snippets of code which we encourage you to give a burn and copy each into your script.

If you have any suggestions or concerns, please email me at **Jared.Adam@psu.edu**.

What this course will cover:

Part 1: The Reasoning

- **Introduction:** What is programming and what is R?
- **R as a tool:** How can we use R?

Part 2: The Beginnings

- **Starting a project:** Importing data
- **Basic functionality:** Logic and shortcuts
- **Several methods for calling different variables** Calling data with operators
- **Getting fancy widdit:** What are functions, packages, and how can we use them?
- **Coding etiquette:** How to write simple and reproducible code

Part 3: Buckle up

- **Data wrangling:** Data wrangling with base R and other important packages

Part I

The Reasoning

1 Introduction

1.1 What is programming?

Programming

Computer programming is the process of **writing code** to facilitate actions in a computer, application, or software program, and instructs them on how to perform.

Each 'type' of programming comes with it's own language. A **programming language** is a vocabulary and set of grammatical rules for instructing a computer or computing device to perform specific tasks. Examples of programming languages include, but are not limited to , C, C++, Java, Python, and of course, **R**.

The **purpose** of programming is to find a sequence of instructions that will automate the performance of a task on a computer.

What is R?

At it's root, R is a language and environment for statistical computing and graphic building. R provides a variety of statistical (*linear and nonlinear modeling, classical statistical tests, time series analysis, classification, etc.*) and graphical techniques (*Base R, ggplot, etc.*).

This software **excels** in its ease of producing publication-ready high-quality plots, use of mathematical symbols, implementation of equations and formulas, and much more. Along with this, R is also a free, open-source software available on a wide variety of platforms, including both **Windows and MacOS**.

1.2 Getting started with R

Downloading R

How to download R, by Garrett Grolemund

8

R vs RStudio

R the application is installed on your computer and uses your personal computer resources to process R programming languages.

RStudio integrates with R as an IDE (Integrated Development Environment) to provide further functionality. To reiterate, RStudio acts as a *housing* of sorts to allow for the functionality and script writing of R. Think of saving photos to iCloud. Without a device, your photos would be free-floating and rather inaccessible. *BUT*, with a device (housing), you are able to access these photos. **RStudio** acts similarly with R in that it provides an environment to use the software. There are other text editors and IDEs that are available, **but we recommend starting with RStudio.** RStudio helps you use the version of R on your computer, but it does not come with it's own version of R.

1.3 The big four

Components of RStudio
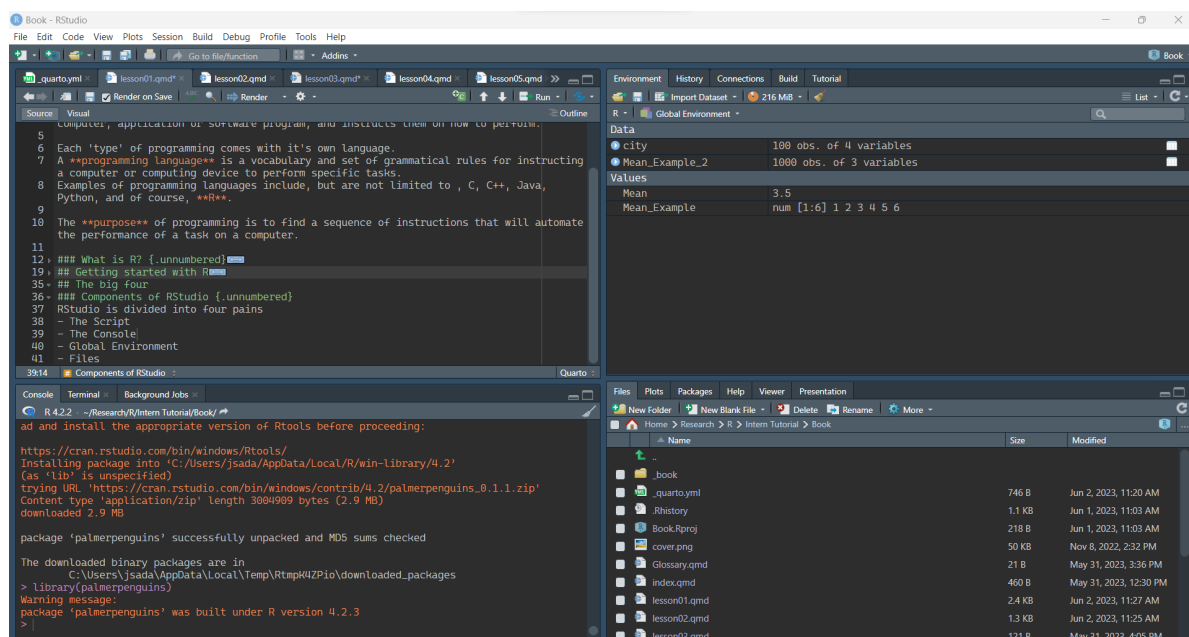
Figure 1.1: A screenshot of my RStudio

1.3.0.0.1 * RStudio is divided into four panes

- The Script (top-left)
- The R Console (bottom-left)
- You Global Environment (top-right)
- Your Files/Plots/Packages/Help/Viewer (bottom-right)

The Script

The section is where your written code will go. Whenever you are giving R commands to complete, this text will be entered in the script.

Along with this, the Script is where any open R files will be housed. This allows you to navigate between scripts with ease.

The R Console
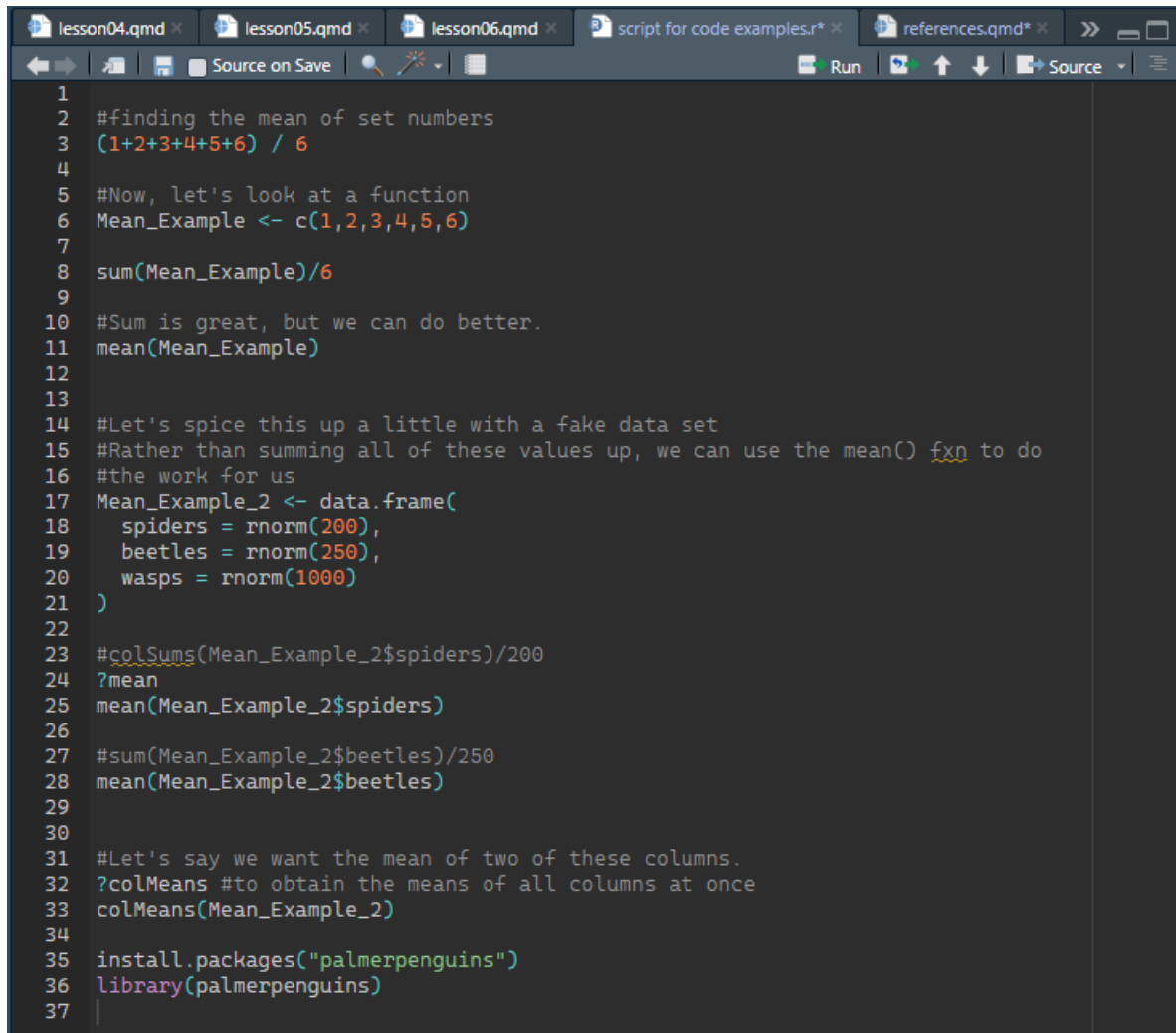
This section is where your outputs will be printed. Whenever you run a line in the script, the console will produce an output, or an error message if the line was unable to be run. As you can see in the picture below, the console's output is both the line I ran, paired with the respective output.

1.3.0.1 The Global Environment

This pane is where any of your imported or created objects will go. These could include, but are not limited to, data sets, functions, vectors, values, etc. If you wish to view your full data set, you can click on the the object. If you wish to view the the column and row names, but not view the full object, you can select the blue and white arrow on the left-hand side of the object name.
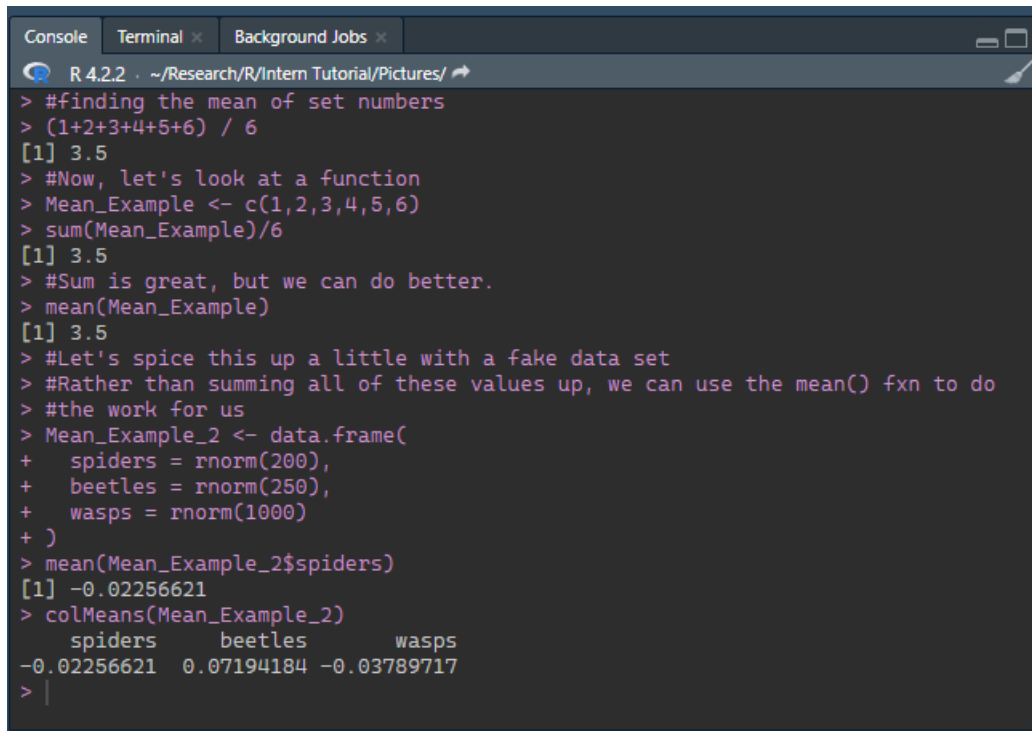
Your Files/Plots/Packages/Help/Viewer

This pane of RStudio is where a lot of information can be found. You can navigate your computers files, view the plots you've developed, install packages, and find helpful information and examples within an easy-to-use search bar.
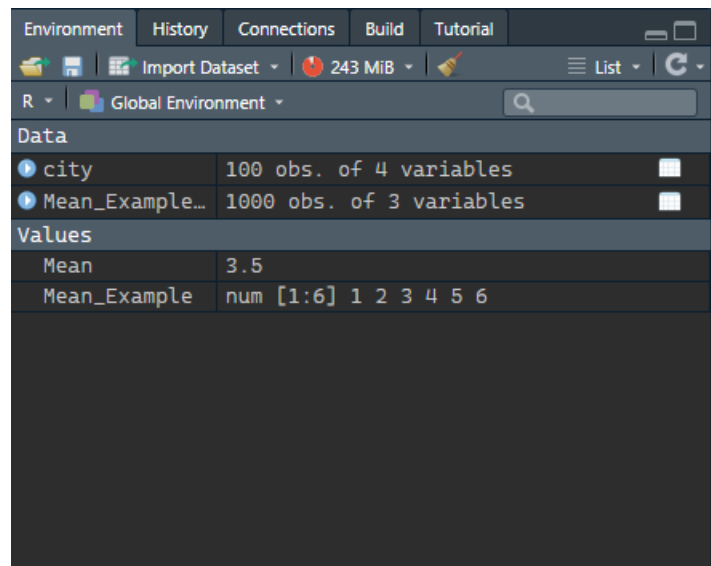
```
1
2   #finding the mean of set numbers
3   (1+2+3+4+5+6) / 6
4
5   #Now, let's look at a function
6   Mean_Example <- c(1,2,3,4,5,6)
7
8   sum(Mean_Example)/6
9
10  #Sum is great, but we can do better.
11  mean(Mean_Example)
12
13
14  #Let's spice this up a little with a fake data set
15  #Rather than summing all of these values up, we can use the mean() fxn to do
16  #the work for us
17  Mean_Example_2 <- data.frame(
18    spiders = rnorm(200),
19    beetles = rnorm(250),
20    wasps = rnorm(1000)
21  )
22
23  #colSums(Mean_Example_2$spiders)/200
24  ?mean
25  mean(Mean_Example_2$spiders)
26
27  #sum(Mean_Example_2$beetles)/250
28  mean(Mean_Example_2$beetles)
29
30
31  #Let's say we want the mean of two of these columns.
32  ?colMeans #to obtain the means of all columns at once
33  colMeans(Mean_Example_2)
34
35  install.packages("palmerpenguins")
36  library(palmerpenguins)
37
```

Figure 1.2: A screenshot of my Script

Figure 1.3: A screenshot of my Conolse. This is what an my ran script output looks like.

Figure 1.4: Global Environment
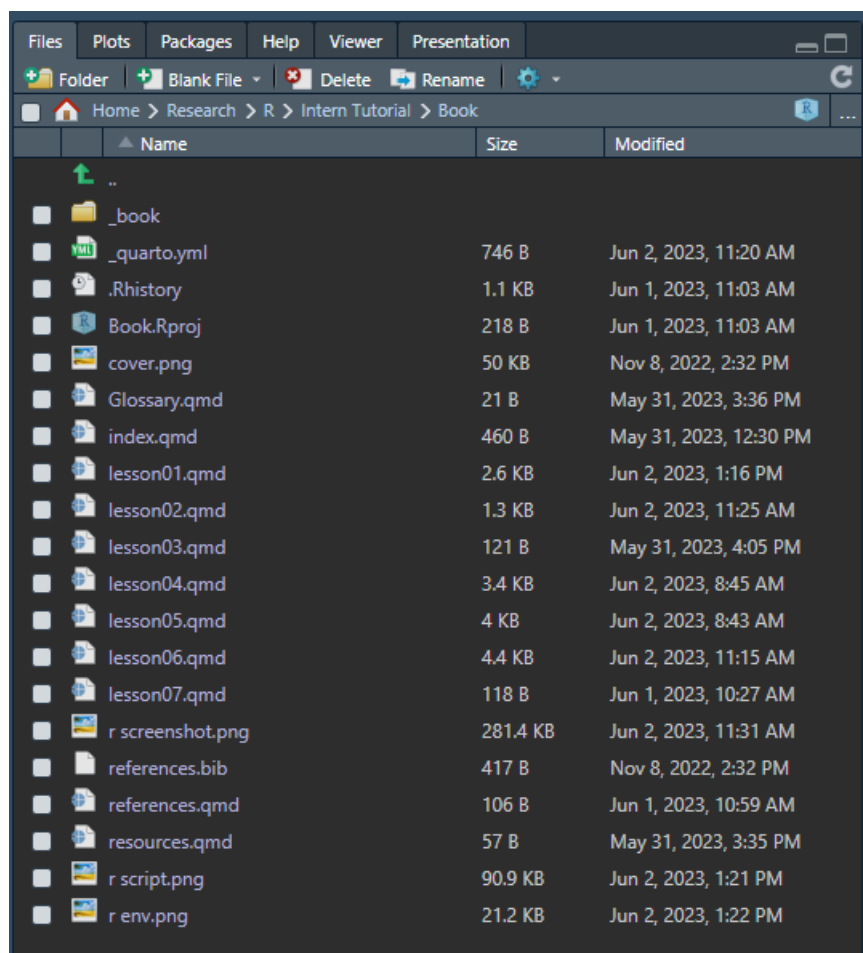
Figure 1.5: A screenshot of my Files and such

2 R as a tool

2.1 Excel vs. R and why we should care

Excel vs R

When choosing between R and Excel, it is important to understand how both solutions can get you the results you need. However, one can make it an easy, reputable, convenient process, whereas the other can make it an extremely frustrating, time-consuming process **prone to human errors**.

When opening Excel and applying data manipulation techniques to your data, are you **easily** able to tell what manipulations have been made without clicking on the column or cells? If you were to share these Excel sheets with colleagues are they easily able to **replicate** your analyses without you telling them where to click or which formulas were applied?

With R all of these are possible. You automatically have all the code visible and in front of you in the form of scripts. Reading and understanding the code is possible because of its easy-to-use, easy-to-read syntax which allows you to track what the code is doing without having to be concerned about any hidden functions or modifications happening in the background.

When we consider our programming methods, we must strive for two goals: **simple and reproducible**. R makes both of these goals achievable.

Let's keep talking about this

I want to inform you of something. This is entirely objective and bias-free (as if that is even possible).

Let's talk excel data sheets for a moment. *Excel* has some great features. The most flexible of these is the *cell.* An excel cell can be extremely flexible as they can store various data types (numeric, logical, and characters).

This is great! We can store our data here in a nice and organized manner and scroll through and view it all with relative ease.

Not so fast. Lettuce think about a data set with 5,000 or 20,000, or 100,000, or 500,000, or 1,000,000 rows and 100+ columns. Now. Imagine **scrolling** through all of this looking for

errors. Or double checking formulas written within new columns. Imagine saving this file over and over upon each rendition. What were to happen if an **error was missed** after formula was run and you continued to work and save new files? This could mean big trouble when it came time for a real analysis. Personally, that sounds like a **nightmare.**

	A	B	C	D	E	F	G	H
1	So	much	data	omg	and	look	here	ItNeverEnds
542	541	543	545	547	549	551	553	555
543	542	544	546	548	550	552	554	556
544	1	545	547	549	551	553	555	557
545	544	2	548	550	552	554	556	558
546	545	547	3	551	553	555	557	559
547	546	548	550	4	554	556	558	560
548	547	549	551	553	5	557	559	561
549	548	550	552	554	556	6	560	562
550	549	551	553	555	557	559	7	563
551	550	552	554	556	558	560	562	8
552	551	553	555	557	559	561	9	565
553	552	554	556	558	560	10	564	566
554	553	555	557	559	11	563	565	567
555	554	556	558	12	562	564	566	568
556	555	557	13	561	563	565	567	569
557	556	14	560	562	564	566	568	570
558	15	559	561	563	565	567	569	571
559	558	16	562	564	566	568	570	572
560	559	561	17	565	567	569	571	573
561	560	562	564	18	568	570	572	574
562	561	563	565	567	19	571	573	575
563	562	564	566	568	570	20	574	576
564	563	565	567	569	571	573	21	577
565	564	566	568	570	572	574	576	22
566	565	567	569	571	573	575	577	579

Now that I got that off my chest. Let's chat about R. Within R are some great options for viewing our data. We can look in our environment. We can call certain base R functions (See functions section here) to view different sections.

Here are some examples of these functions.

The structure (**str**) to view the nature of our data set.

```
# The str() function to view the structure of the data set
str(Penguins)
```

```
tibble [344 × 8] (S3: tbl_df/tbl/data.frame)
 $ species          : Factor w/ 3 levels "Adelie","Chinstrap",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ island           : Factor w/ 3 levels "Biscoe","Dream",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ bill_length_mm   : num [1:344] 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
 $ bill_depth_mm    : num [1:344] 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
 $ flipper_length_mm: int [1:344] 181 186 195 NA 193 190 181 195 193 190 ...
 $ body_mass_g      : int [1:344] 3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
 $ sex              : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
 $ year             : int [1:344] 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 ...
```

The **head** function to view the first several rows of a data set.

```
# The head() function to view the several rows
head(Penguins)
```

A tibble: 6 × 8

species <fctr>	island <fctr>	bill_length_mm <dbl>	bill_depth_mm <dbl>	flipper_length_mm <int>	body_mass_g <int>	sex <fctr>	year <int>
Adelie	Torgersen	39.1	18.7	181	3750	male	2007
Adelie	Torgersen	39.5	17.4	186	3800	female	2007
Adelie	Torgersen	40.3	18.0	195	3250	female	2007
Adelie	Torgersen	NA	NA	NA	NA	NA	2007
Adelie	Torgersen	36.7	19.3	193	3450	female	2007
Adelie	Torgersen	39.3	20.6	190	3650	male	2007

6 rows

The **tail** function to view the last several rows of our data set.

```
# The tail () functoin to view the last several rows
tail(Penguins)
```

A tibble: 6 × 8

species <fctr>	island <fctr>	bill_length_mm <dbl>	bill_depth_mm <dbl>	flipper_length_mm <int>	body_mass_g <int>	sex <fctr>	year <int>
Chinstrap	Dream	45.7	17.0	195	3650	female	2009
Chinstrap	Dream	55.8	19.8	207	4000	male	2009
Chinstrap	Dream	43.5	18.1	202	3400	female	2009
Chinstrap	Dream	49.6	18.2	193	3775	male	2009
Chinstrap	Dream	50.8	19.0	210	4100	male	2009
Chinstrap	Dream	50.2	18.7	198	3775	female	2009

6 rows

The **colnames** function to view the names of our columns.

```
#The colnmaes() fcuntion to view my column names
colnames(Penguins)
```

```
[1] "species"        "island"        "bill_length_mm"  "bill_depth_mm"  "flipper_length_mm" "body_mass_g"   "sex"          "year"
```

As we are starting to see, when compared to Excel with examples of only viewing data, R is beginning to appear more versatile. We will continue to build on the capabilities of R in future sections and work through functions, etiquette, data wrangling, plotting, and much more.

Part II

The Beginnings

3 Starting a project

3.1 Starting a new project

To begin, we must first open a new project.

1. To open a new project, you first select the RStudio app on your computer. Unless immediately prompted, select *New Project* under the *File* tab.

2. Next you will be prompted select which Directory type. Select *New Directory*.

3. Next, in the *Project Type* screen, you will select *New Project*.

4. Once selected, you will be prompted to name the Directory. Make this name unique and choose where you would like it to be saved.

3.2 R-compatible data sets

When importing a data set from excel into R, the file type must be a **.csv**, rather than the typical **.xlsx**.

For further instruction on preparing an R-ready .csv, please see the link below.

Building an R-friendly .csv

3.3 How to get data into R

Importing data

Working directory

Your working directory is where you will have R pull data sheets from. There are two common ways of doing this.
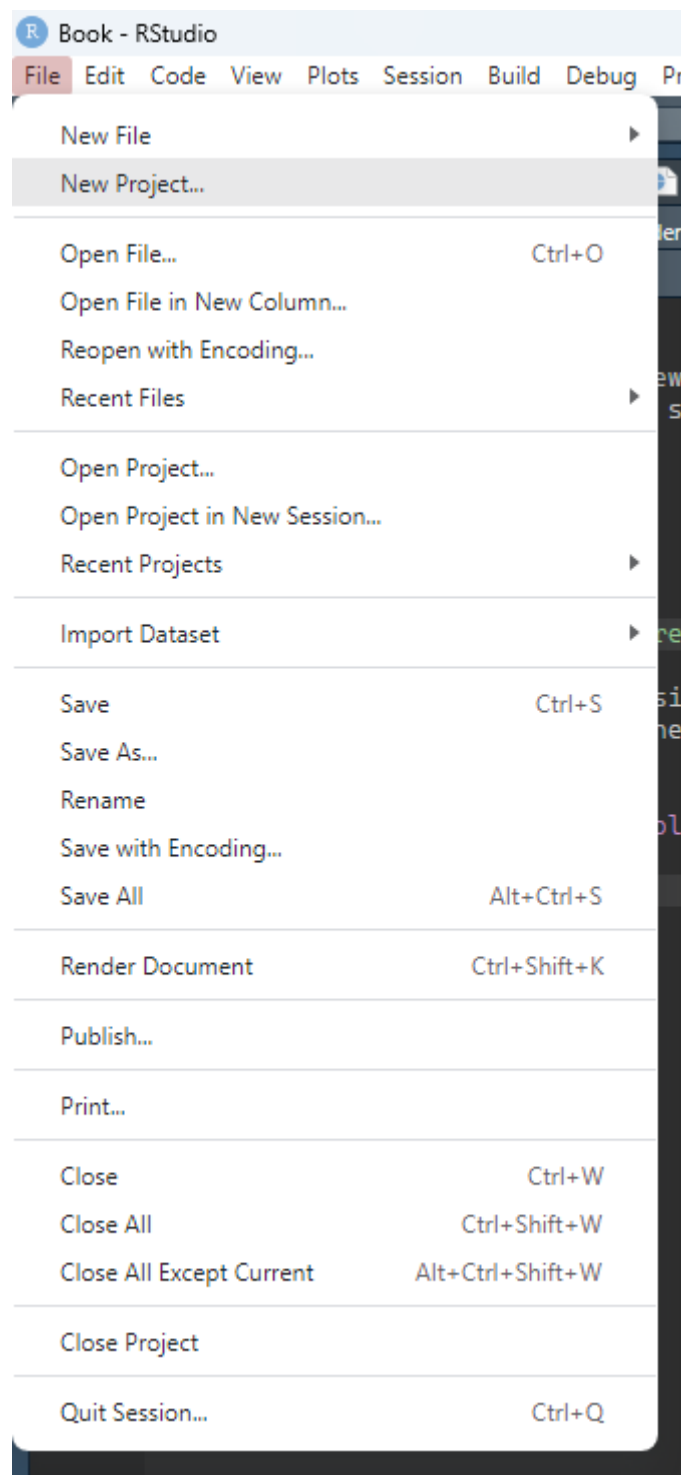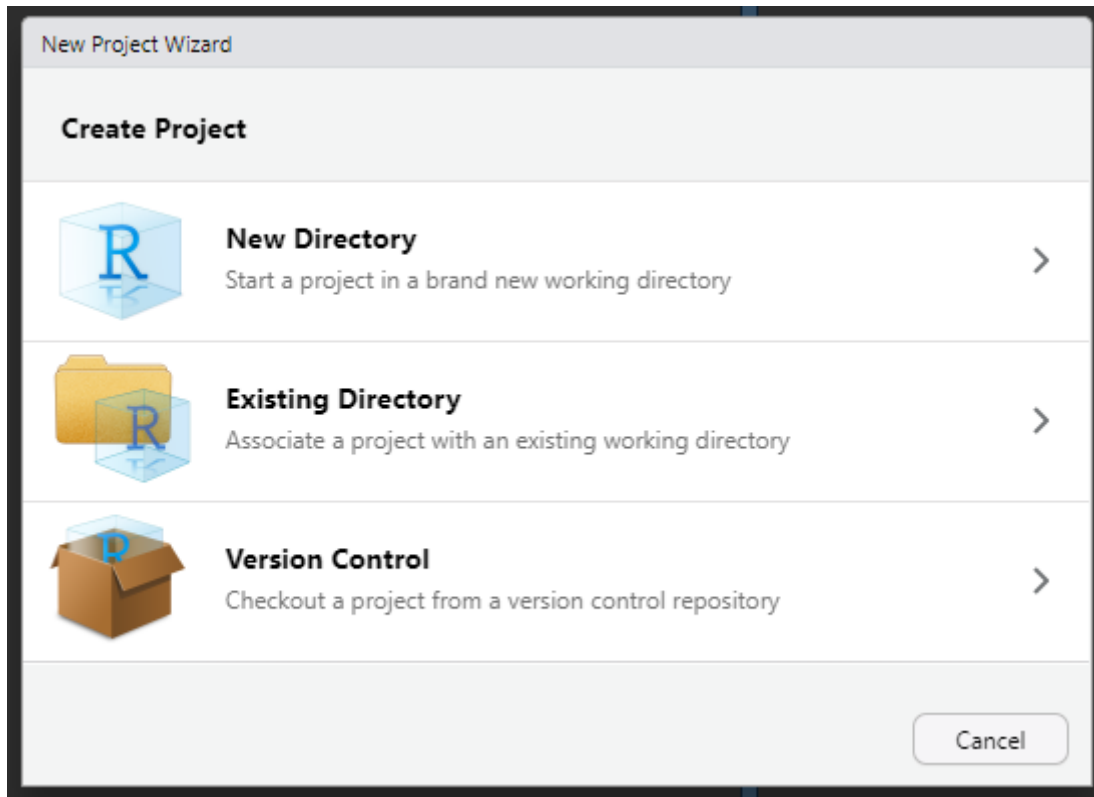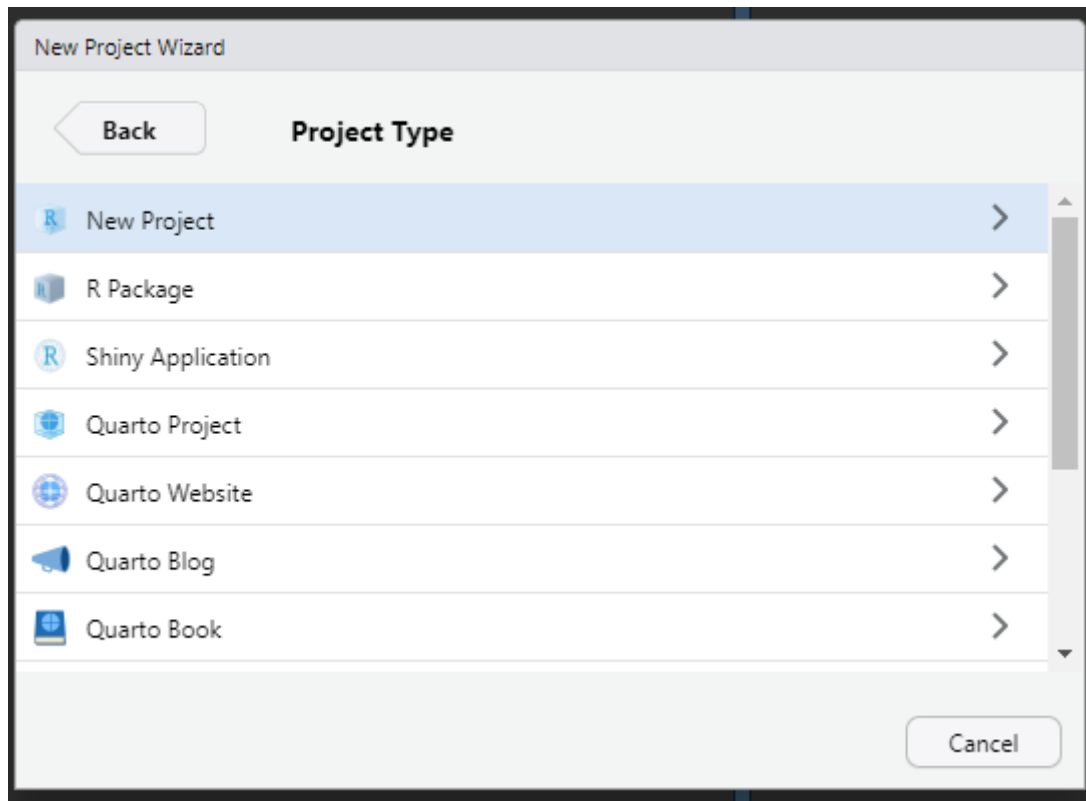
New Project Wizard

Back   **Project Type**

R New Project	>
R Package	>
R Shiny Application	>
Quarto Project	>
Quarto Website	>
Quarto Blog	>
Quarto Book	>

Cancel

The first step is to determine where R currently thinks our working directory is. To do this, we use the **getwd** function.

```
getwd()

#output:"C:/Users/jsada/OneDrive - The Pennsylvania State University/Documents"
```

As you can see, the output shows my computer pathway, or source, of where R will be obtaining files.
If we wish to change this, we have two options.

First, we will use the **setwd** package. Within the parentheses of this function, we will write out the desired pathway. Let's say I wish to be more specific than just the *Documents* folder.

```
setwd("C:/Users/jsada/OneDrive - The Pennsylvania State University/Documents/Research/R/In
```

The second method for setting your working directory is done through the *Session* menu at the top. You will then hover over *Set Working Directory* and then select *Choose Directory...* From here, you will navigate to the folder you wish to pull data from.
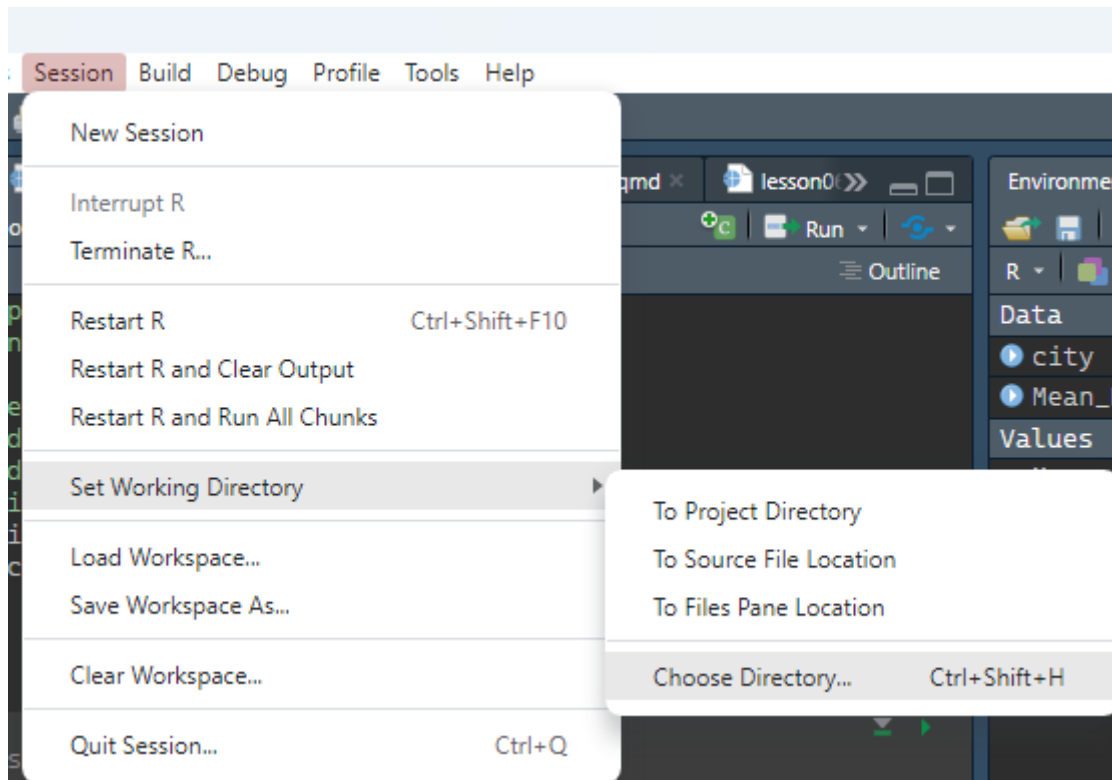
Figure 3.1: Setting my working directory

Now that we have our working directory set, we can take a look into the folder.

To see how many files are in my directory, I will use the **length** function.

```
length(list.files())
#output: 24
```

To see the first five files within my directory, I will use the **head** function.

```
head(list.files())
#output:

#[1] "_book"        "_quarto.yml"   "Book.Rproj"     "cover.png"
#[5] "directory.png" "Glossary.qmd"
```

The last command we will run to investigate our working directory is the **%in%** operator. We will use this operator to see if there is specific file within our directory. This operator will provide us with a logical out (TRUE or FALSE). When calling a specific object, we must use either half, or full parentheses.

```
'Book.Rproj' %in% list.files()

#output: TRUE
```

Github

In this book, I will not cover using the Github platform for data storage. If you wish to explore this further, please see the linked tutorial below.

[A Github tutorial by Callum Arnold](#)

3.4 Types of data

Now that we have our data set imported into R, we can begin looking our data. The first step is gaining an understanding of the *type* of data we have. Within R, there are 5 main types of data. These include:

Data type	Example
numeric	(10.5,55,680)
integer	(1L, 55L, 100L, where the letter "L" declares this an integer)
complex	(9+3i, where "i" is the imaginary part)

Data type	Example
character	(Also known as strings - "k", "bugsRcool", "11.5", "etc.")
logical	(TRUE and FALSE)

When it comes to data manipulations, statistical tests, model building, and developing plots, it is incredibly important that our data are classified as the correct data type. To determine this for single variables or values, we use the **class** function.

Copy these examples into your script to try this function out.

```r
# numeric
x <- 10
class(x)

# integer
x <- 10L
class(x)

# complex
x <- 9i + 3
class(x)

#character/string
X <- "Boy howdy, this is rivetting stuff"
class(x)

# logical
x <- TRUE
class(x)
```

4 Basic functionality

4.1 Shortcuts, arithmetic commands, logic, and other helpful tools

Shortcuts

Run

Task	Windows & Linux	Mac
Run	Ctrl + Enter	Cmd + Return

Shortcuts for editing

Task	Windows & Linux	Mac
Copy	Ctrl + C	Cmd + C
Paste	Ctrl + V	Cmd + V
Undo	Ctrl + Z	Cmd + Z
Redo	Ctrl + Shift + Z	Cmd + Shift + Z
Select All	Ctrl + A	Cmd + A
Indent	Tab	Tab
Outdent	Shift + Tab	Shift + Tab
Insert pipe operator	Cmd + Shift + M	Cmd + Shift + M

Arithmetic operators

Task	Operator	Example
Addition	+	x + y
Substraction	–	x – y
Multiplication	*	x * y
Division	/	x / y
Exponent	^	x ^ y

Logic commands

Task	Operator	Example
Equal	==	x == y
Not equal	!=	x != y
Greater than	>	x > y
Less than	<	x < y
Greater than or equal to	>=	x >= y
Less than or equal to	<=	x <= y

Other important operators

Task	Operator	Example
Call for the help menu for the respective function	?	?mean
Assigning a name or value	<-	Name <- "Chop"
To access one variable in a data set	$	DataSet$IWantThisColumn
Searching or calling exact characters	""	"Chop Spring"
Concatenate (c), combines arguments, both numbers or words, into a vector	c()	Name <- c("Jared", "Daniel", "Chop") Numbers <- c(1,2,3)
Another form of concatenation used when writing long scripts of code which span multiple lines	+	This line + this line too

5 Several methods for calling different variables

5.1 Background of the operators

When calling specific variable, list, or object in R, there are three main ways. These include the $ operator, the [] (bracket), and the %>% (pipe) operator. These operators work in similar ways in that they allow you to access specific *pieces* of your data frame.

In this section we will investigate all three.

The dollar sign: $

This operator is used in R to access the list of a data frame. You can use this operator to access variables, add values or objects, update (e.g., change a class), and delete variables from a data frame.

Let's start by creating a data frame.

```r
score <- 1:4 #score column with values of 1-4

insect <- c('wasp', 'beetle', 'ant', 'TrueBug') #insect column with a list of names

field <- c('corn', 'beans', 'corn', 'beans') #field column with a list of names

Avg_Weight_mg <- c(2, 7, 0.5, 3) #assigning weight values to each insect

Sample_DF <- data.frame(score, insect, field, Avg_Weight_mg) #using the data.frame functio

print(Sample_DF) #using the print function to view my new data frame
```

```
  score  insect field Avg_Weight_mg
1     1    wasp  corn           2.0
2     2  beetle beans           7.0
3     3     ant  corn           0.5
```

```
4       4 TrueBug beans           3.0
```

Now that we have our data frame, let's use the **$** operator to investigate our data.

First, let's say we want to look at the *insect* column. The **$** operator here pulls out the just the values of this column. **Notice**, to use this, we need to specify a source. The source here is '*Sample_DF*'. Try this operator out to see which variations of column names will, and will not provide you with an output.

```
Sample_DF$insect #show me the insect column within Sample_DF
```

```
[1] "wasp"    "beetle"  "ant"     "TrueBug"
```

So, now we can see the functionality of this operator as a means of viewing data. Let's now investigate adding a new column to this existing *Sample_DF.*

In this example, we will be adding a column to specify whether the insect was an adult, or not. We will do this by adding our new object name following our source, *Sample_DF*, and then specifying the values of this column. *Remember*, we need to have a source data frame when we use this operator.

```
Sample_DF$adult <- c('yes', 'no', 'no', 'yes') #naming a new column IN Sample_DF with the

print(Sample_DF) #printing this updated df
```

```
  score  insect field Avg_Weight_mg adult
1     1    wasp  corn           2.0   yes
2     2  beetle beans           7.0    no
3     3     ant  corn           0.5    no
4     4 TrueBug beans           3.0   yes
```

We can now see our new column was added to the right-hand side of the existing data frame.

Lettuce look at one more example of how we can use this operator. Here, we will be changing the *class* of an object within Sample_DF.

In this example, I want to change the class of *score* from integer, to numeric. Notice, I have to call the data source in the beginning to tell R I **do not** want to create a new object, I only want to change my existing data frame. Next, I need to specify the source for the function, **as.numeric**.

```r
Sample_DF$score <- as.numeric(Sample_DF$score) #within my df, in the score column, change
```

If you wanted to create a new object with this change, we can simply change the name of the object.

We can do this, like so.

```r
New_DF <- as.numeric(Sample_DF$score) #within a new df, in the score column, change the cl

print(New_DF) #show me the new df
```

```
[1] 1 2 3 4
```

Notice here, the new object **only** houses the values from the *score* column.

Single dimension square bois: []

Brackets [], in R, work similarly to that of the dollar sign ($). Brackets are especially useful when we want to extract single elements from an object. Let's start by creating a simple, single dimension vector.

```r
#ceating a numeric vector
Vector_One <- c(1,2,3,4,5,6)
```

Now, let's pull some stoof out. In this example, I am going to pull out several individual values from the vector we just created.

```r
Vector_One[1] #extracting the first value of the data set
```

```
[1] 1
```

```r
Vector_One[3] #extracting the third value of the data set
```

```
[1] 3
```

Now that we have extracted individual values, let's pull several out at once. Notice that the syntax has changed a bit. We now **must** tell R that we want to combine the three values into one output. This is done by adding, *c(1,2,3)*, within our brackets

```
Vector_One[c(1,2,3)] #extracting the first, second, and third value of the data set
```

[1] 1 2 3

The next step is to have R to pull values out based on a command. In the following example, we will use some of the logic commands we covered earlier.

Let's say I want to see all of the values in this data set that are above the number 3.

```
Vector_One[Vector_One > 3] #extracting values greater than 3
```

[1] 4 5 6

We can repeat this step with any logical operator we would like.

For example.

```
Vector_One[Vector_One >= 2] #extracting values greater than or equal to 2

Vector_One[Vector_One != 2] #extracting values that do not equal 2
```

Multiple dimension square bois: []

Now that we can see how to use the brackets when looking for single objects (like a simple vector), let's start to look at the use of brackets with an increase in dimensions. Multiple dimensions come into play when we are investigating a full data frame or matrix. In this section, we will be looking at the Sample_DF data frame we created above.

Within the bracket are assigned values. By this, I mean, depending on the *location* of the number within the bracket, the location that information is pulled from will change. The assigned locations are [**row, column**]

For example, if we were to run [**1,2**], our output would be the value in the *first* row and *second* column.

In this example, we will pull out the values from the first row, and second column.

```
Sample_DF[1,2] #extracting values from row 1 and column 2
```

[1] "wasp"

Next, let's investigate what happens when we leave one of the 'values' blank.

31

```r
Sample_DF[,2] #extracting values from all rows and the second column
```

```
[1] "wasp"    "beetle"  "ant"     "TrueBug"
```

What we see here is that R gave us the values from all rows, but just the second column.

We can use the same method if we want to view information from all one row, but *all* columns.

```r
Sample_DF[1,] #extracting values from row 1, and all columns
```

```
  score insect field Avg_Weight_mg adult
1     1   wasp  corn             2   yes
```

In the next example, we will investigate how to *exclude* information. Let's say we want to view the whole data frame except for the values of row 1. This is done by using, *-1*, in the row value of the brackets.

In this example, I am telling R to exclude all values of row 1 from the output.

```r
Sample_DF[-1,] #extracting all values, except for row 1 information
```

```
  score  insect field Avg_Weight_mg adult
2     2  beetle beans           7.0    no
3     3     ant  corn           0.5    no
4     4 TrueBug beans           3.0   yes
```

In our last example of the bracket, we will extract information from a specified column, but all rows. To do this, we will continue to leave the row value blank, but add in the exact name of the column we seek to view.

Let's take a look at the 'insect' column.

```r
Sample_DF[, "insect"] #extracting values from all rows, but just the insect column
```

```
[1] "wasp"    "beetle"  "ant"     "TrueBug"
```

Our output shows us all of the values within the *insect* column.

Last but not least, the pipe: %>%

SO. We have investigated, and worked through, the dollar sign operator and brackets for pulling out specific elements. These methods are certainly effective, but as we start to work through larger data sets of raw data, there may be many changes we need to apply.

To accomplish this, we could write out a new command line for each iteration, OR, we can 'pipe' several commands into one operation. This processing of piping *links* all of our changes to one command, allowing for efficiency and easy error-tracking. To reiterate, this task is the *chaining* of arguments into one command.

This operator, the pipe `%>%`, is arguably one of the most important operators in data wrangling and processing.

Rory Spanton, with *Toward Data Science*, explains this process well, "To visualize this process, imagine a factory with different machines placed along a conveyor belt. Each machine is a function that performs a stage of our analysis, like filtering or transforming data. The pipe therefore works like a conveyor belt, transporting the output of one machine to another for further processing."

Here I will write out two examples. Within these examples, I will be creating functions and then running them sequentially both with, and without the pipe operator. We **will** cover writing functions in the future.

```
# starting with creating three separate functions

# a function to add two values
add <- function(x,y) {
  return(x+y)
}

# a function to multiply two values
mul <- function(x,y) {
  return(x*y)
}

# a function to divide two values
div <- function(x,y) {
  return(x/y)
}
```

Now that we have our functions created, let's put them to work in the *long form*.

```r
# I am now calling each function sequentially

result_1 <- add(2,4) # applying my add function to two values (x,y)

result_2 <- mul(result_1, 5) # applying my mul function to the results from the add functi

result_3 <- div(result_2, 6) # applying my div function to the results from the mul functi

print(result_3)
```

```
[1] 5
```

As we can see, this method is effective. But, where it falters, is that we must save each iteration and then input that object name into the next function. While this example is simple, we can imagine how with an increase in the complexity of our functions and sequential manipulations, this can become an overwhelming method.

Let's now look at the same sequence of functions, but this time using the pipe operator.

First, we will need to load in the **dplyr** package to use the pipe operator.

```r
library(dplyr) #loading the dplyr package
```

```r
# piping my three functions together
results <- add(2,4) %>% # adding 2 and 4 with the add function
  mul(5) %>% # chaining the results from add into the mul function
  div(6) #chaining the results from the mul function into the div function

print(results) #printing the results
```

```
[1] 5
```

We got the same output! As we can see, this method is both cleaner (regarding your environment and saving objects over and over) and safer (regarding to errors) than the sequential example.

The results, explained

Continuing to follow Rory's brilliant synthesis of this operator, I will use their example here. Let's think of **%>%** as the word '*then*'.

Let's now write out the same piping example.

- The results from this chain will be named "*results*",

  – I will be **adding** the numbers 2 and 4 together, *THEN*

  – I will **multiply** the results from the addition by 5, *THEN*

  – I will **divide** the results from the multiplication by 6

As we can see, this operator acts as a *link in the chain* which holds the whole argument together, allowing it to act as one command. The pipe operator is an excellent addition your coding repertoire when you would like to eliminate the saving of multiple objects with each iterative change, lower the risk of an error occurring within the multiple changes, and allow for a cleaner, more palatable, R script.

6 Getting fancy widdit

6.1 Functions, packages, and all that jazz

What is a function?

A **function** in R is an object containing multiple interrelated statements that are run together in a predefined order every time the function is called. What this means, is that within every function, there are set of instructions to be followed in their respective order to complete a desired task.

For example, let's say we want to find the mean value of a desired set of numbers.

```
(1+2+3+4+5+6) / 6
```

```
[1] 3.5
```

This is an effective method for acquiring the average of a small set of numbers that are not saved in R. But, what if they were saved?

In this example, we create a vector named 'Mean_Example' with the previous six numbers. Rather than adding them manually, we use the **sum** function to automatically add the values. We then set this sum to be divided by six, which is the total number of values.

```
Mean_Example <- c(1,2,3,4,5,6)

sum(Mean_Example)/6
```

```
[1] 3.5
```

Now, let's crank this up a notch. What do we do if we have a large data set and want to calculate the mean of a column? First, I created a data set named *'bugs'* with three columns: *spiders, beetles, and wasps.* Then, using the **rnorm** function, I set the number of values per column with a **default mean of 0** and standard deviation of 1. The purpose of this fake data set is just so we have something to work with with an *expected* mean.

Once this data set is created, we can test out the **mean** function on one of the columns. Within the mean function, I tell R to take the mean of the spider column FROM the 'bugs' data set we created. The '$' symbol tells R where to look within an existing data set.

```r
bugs <- data.frame(
  spiders = rnorm(200),
  beetles = rnorm(250),
  wasps = rnorm(1000)
)

mean(bugs$spiders)
```

```
[1] 0.005515399
```

```r
#Remember, our default mean was 0
```

What is a package?

While R has many built in functions (e.g., **mean**), some of the most useful functions do not come pre-installed. When this is the case, they are provided to us in well made, neatly packed downloadable objects called *packages*. In essence, the creator of the package has nestled a bunch of things to make your programming life easier into a little folder you can download, and use, at your leisure. An R package can bundle together useful function, help files, and data sets. Typically, a package will have a list of functions all related to the same task or set of tasks.

Let's take a look at the **ggplot2** package. The *purpose* of this package is on the **grammar of graphics**; the idea that you can build every graph from the same components: a **data** set, a **coordinate system**, and **geoms**-visual marks that represent data points. Functions, such as **ggplot**, that reside within this package are all designed for the ease of figure development.

Let's take a look at how to download a package, starting with **ggplot2**.

To download a package, we must use the **install.packages** function, and place the desired package name in *"quotation marks"* within the function parenthesis. Once downloaded, we must then 'call' the function into our system. Using the **library** function, we tell R to load this package into our current project. We only need to install the package once, but we must 'call' it in every time we restart Rstudio. (To run this code, you must remove the # symbol from **install.packages**.

```
#install.packages("ggplot2")

library(ggplot2)
```

Packages with data

Now that we can investigated installing and downloading a package for the use of functions, we will now explore available data sets on R. There are many available data sets within R that we can download and practice programming, but for this tutorial we will work with **Palmer Penguins**. It is wise to download this now, because we will revisit this data set in future sections.

Take notice of how I use quotation marks around the package name when using the function **install.packages**, but not when I load the package into my session with the **library** function.

```
#install.packages("palmerpenguins")

library(palmerpenguins)
```

7 Coding etiquette

7.1 How to write code that is clean, clear, and reproducible

Naming new objects in R

First, and foremost, we should investigate how to name new objects within R. There are two methods to complete this task. These include the equals sign, `=`, and the carrot jawn, `<-`.

For example, the output of these lines will be identical.

```r
data = c(1,2,3) #new object with equals sign

data <- c(1,2,3) #new object with carrot jawn
```

While these *are* the same, what we recommend is using the carrot method (`<-`). This is because many functions require the use of `=` within code (this will come up in the future). For this reason, we find it best to utilize the carrot *whenever* you change the name of an object, or add a new object.

It is important to remember that you can change the time of an object *whenever* you want. This is especially helpful when making changes to our data set. With each iterative change, we can change the name of the object. This will allow us to backtrack to the pevious change in the event we make a mistake.

Style

Coding, like any other writing type, is dependent upon clear and consistent style. As the tidyverse style guide so eloquently put it, "Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread." Here, we can clearly see how a simple phrase becomes exponentially more challenging to read and understand. The same goes for coding.

It is important to remember that R **cannot** handle spaces between words. Because of this, we must be creative in how we name things. It is a good idea to follow the **BigCamelCase** naming method. Let's start by naming a vector.

```
#GOOD
MyNewVector <- c(1,2,3,4,5,6)

#BAD
mynewvector <- c(7,8,9,10,11,12)
```

While in this example it is fairly easy to read both, we see how the name following the **BigCamelCase** format is easier to follow.

Let's look at another example of naming, this time getting more specific with our vector name.

```
#GOOD
SlugDensityData_Spring2023 <- c(1,3,5,7)

#BAD
slugdensitydataspring2023 <- c(2,4,6,8)
```

I incorporated an underscore in the first name to make it even more distinct. We can clearly see now with increasing complexity of our names, the first is much easier to read.

Annotations

When taking notes in a lecture, do you think it wise to take poorly written and hard to understand notes? Or, would we rather take clear, concise, and methodical notes to ensure we can return to them and understand exactly what the lecture was about? If you choose the former, then please, continue reading.

Annotation, like note taking, is very important within our code. We must be able to return to each line and know exactly what we did and why we did it. Along with this, if we wish to share this code with anyone, they too must be able to understand the methodology without needing you by their side. The habit of good code annotation is one that should be adopted immediately and practiced throughout the duration of your programming days.

Let's look at some examples of both good, and bad annotations.

```
#GOOD
##
#I am creating a vector to practice running different functions

PracticeVector <- c(11,3,4,5,6,7)
```

```
#Trying out the mean function here
mean(PracticeVector)
#This works. I will leave this code here to reference in the future
##
```

```
#BAD
practicevector <- c(11,2,3,4,5,11,2)
mean(practicevector)
```

While these examples are very simple in their nature, we can imagine how scrolling through 500+ lines of un-annotated code can be a nightmare. Along with this, to reiterate my naming point, we can see how with poor naming practices and a lack of annotation, the bad example is doubly hard to follow.

Along with annotating what you are doing, it can also be helpful to write out your thought process for an action. Let's say you are writing code for a project on a Friday, and since you are great at managing your workload, you plan to not work this weekend. When Monday rolls around, you open your R script up and have completely forgotten why you were running a specific test or structuring your code a specific way.

With proper annotation, this hiccup can be avoided.

Let's take a look at some examples.

```
#GOOD
##
#I am trying to create a fake data set to practice some functions on
#Not to be used for analyses, simply for me

bugs <- data.frame( #naming this 'bugs' and using the data.frame function to build this
  spiders = rnorm(200), #naming this column 'spiders' and using the rnorm function. This f
  beetles = rnorm(250), #Same as above, but with 250 values
  wasps = rnorm(1000) #Same as above, but with 1000 values
)
##
```

In this example, I clearly noted what I was doing and why I was doing it. For my sake, I can return to this easily. If someone else was to come upon this, they too would be able to understand what my process was.

```
#BAD
notbugs <- data.frame(
  clover = rnorm(200),
```

```
    shrubberies = rnorm(200),
    elderberry = rnorm(2000)
)
```

In this example, it is unclear what the purpose of this data set is. Along with that, if someone is not familiar with this script, they may find it very challenging to follow.

Part III

Buckle up

8 Data wrangling

8.1 Cleaning our data in R

R > excel

Before we begin wrangling our data set, let's quickly discuss why cleaning data in R is important. First and foremost, my Excel file never needs to change. Along with this, I only need to save it once. No saving a new copy every time I make a new column or want to write a formula.

In this section, we will go over some important packages and key functions that will aid in the transition from Excel-based data wrangling to exclusively R-based data wrangling. SO, without further ado, lettuce begin.

What we use to wrangle

When we wrangle our data sets, there is one 'umbrella' package I find to be the most versatile. This is **tidyverse**. **Tidyverse** houses many useful packages for data manipulation, but in this section I will only be referring to one. This package is **dplyr**. While I rely on this package a lot, I also use functions pre-installed in Base R while wrangling.

Please see the wrangling cheat sheets here.

8.2 dplyr Examples

Let's start by installing the needed packages. Remember to remove the # symbol to install these packages. We will also call in our Palmer Penguins data set that you downloaded earlier.

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

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

```
library(tidyverse)

library(palmerpenguins)
data(package = 'palmerpenguins') #investigating the pieces of this data set. How many data
```

NAs in R

Before we begin, we must first check for missing values. R *does not* love when NAs get thrown into the mix, especially when running numerical commands, like the **mean** function.
To resolve any missing values, we must first determine if there are any NAs, and where they may be.

First, we will look to see IF and WHERE potential NAs are in our data set. We will do this by using the **which** function, followed by the **is.na** function within our penguins data set. What these two functions do together is which locates all of the columns where NAs are present, based on the **is.na** function. If we wanted to find all of the columns where there were no NAs, we could simply change **is.na** to **!is.na**.

In this example, my command is to determine where the NAs are. Using the *arr.ind = TRUE* command, I am telling R to show me *exactly* where they are. In this example we are using *penguins_raw*.

> **i** Note
>
> This is the **only** section that will use the raw data set in the example.

```
which(is.na(penguins_raw), arr.ind = TRUE) #checking for NA locations in penguins_raw
```

Now that we see there are NAs riddled throughout, we will name a new object (using the same name), but omitting all NAs.

In this example, I am using the **na.omit** function to remove all NAs from our penguin data set. I am also naming this new object (penguins_raw again for ease).

```
penguins_raw <-
  na.omit(penguins_raw) #I must add this because there are NAs within this data set and th
```

select()

Let's start by selecting for only the columns we are interested in. This can be useful when removing variables we are not currently interested in. **Remember**, at any manipulation, you

can save the changes as a new object which will maintain the integrity of the original if you must back track for whatever reason.

In this example, we use the **select** function to choose which columns we want to look at. **Notice**, we are back to using *penguins*!

```
# select()
penguins %>% #From the penguins data set
  select(species, bill_length_mm, year) # selecting columns species, bill_length, and
```

```
# A tibble: 344 x 3
   species bill_length_mm  year
   <fct>            <dbl> <int>
 1 Adelie            39.1  2007
 2 Adelie            39.5  2007
 3 Adelie            40.3  2007
 4 Adelie            NA    2007
 5 Adelie            36.7  2007
 6 Adelie            39.3  2007
 7 Adelie            38.9  2007
 8 Adelie            39.2  2007
 9 Adelie            34.1  2007
10 Adelie            42    2007
# ... with 334 more rows
```

Now that we have selected for certain columns, let's say we want to view everything except for one or several columns. Instead of typing out every column we want, we can simple type out the one(s) we don't.

In this example, I tell R to remove the sex column using the **select** function again.

```
# select()
penguins %>% #From the penguins data set
  select(-sex)# selecting all columns except for sex
```

```
# A tibble: 344 x 7
   species island    bill_length_mm bill_depth_mm flipper_length~1 body_~2  year
   <fct>   <fct>              <dbl>         <dbl>            <int>   <int> <int>
 1 Adelie  Torgersen           39.1          18.7              181    3750  2007
 2 Adelie  Torgersen           39.5          17.4              186    3800  2007
 3 Adelie  Torgersen           40.3          18                195    3250  2007
 4 Adelie  Torgersen           NA            NA                NA      NA   2007
```

```
 5 Adelie  Torgersen             36.7        19.3            193   3450  2007
 6 Adelie  Torgersen             39.3        20.6            190   3650  2007
 7 Adelie  Torgersen             38.9        17.8            181   3625  2007
 8 Adelie  Torgersen             39.2        19.6            195   4675  2007
 9 Adelie  Torgersen             34.1        18.1            193   3475  2007
10 Adelie  Torgersen             42          20.2            190   4250  2007
# ... with 334 more rows, and abbreviated variable names 1: flipper_length_mm,
#   2: body_mass_g
```

Within the **select** function, you can also look for items based on their spelling. This can be especially helpful if you suspect there to be a spelling error somewhere if your data set. In this example, we will search our data set for any variable name that starts with the letter 'b'.

```
# select()
penguins %>% #From the penguins data set
  select(starts_with('b')) # selecting columns that start with 'b' and using starts_with
```

```
# A tibble: 344 x 3
   bill_length_mm bill_depth_mm body_mass_g
            <dbl>         <dbl>       <int>
 1           39.1          18.7        3750
 2           39.5          17.4        3800
 3           40.3          18          3250
 4           NA            NA            NA
 5           36.7          19.3        3450
 6           39.3          20.6        3650
 7           38.9          17.8        3625
 8           39.2          19.6        4675
 9           34.1          18.1        3475
10           42            20.2        4250
# ... with 334 more rows
```

rename()

Now that we have viewed and selected for different columns and such, we manipulate our data set further. We will start by renaming some columns. Notice with rename, there are two methods you can use. One *without* quotation marks, and one *with.*

In this example, using the **rename** function, I am changing 'species' to 'Species and 'year' to 'Year'.

```
  # rename()
  penguins %>%
    select(species, bill_length_mm, year) %>% #selecting the columns I want to look at
    rename( #rename function. notice here the two methods of changing names
      Species = species, #changing species to Species without quotes
      "Year" = year #changing year to Year with quotes
    )
```

```
# A tibble: 344 x 3
   Species bill_length_mm  Year
   <fct>            <dbl> <int>
 1 Adelie            39.1  2007
 2 Adelie            39.5  2007
 3 Adelie            40.3  2007
 4 Adelie            NA    2007
 5 Adelie            36.7  2007
 6 Adelie            39.3  2007
 7 Adelie            38.9  2007
 8 Adelie            39.2  2007
 9 Adelie            34.1  2007
10 Adelie            42    2007
# ... with 334 more rows
```

arrange()

let's arrange some stuff this is equivalent to sort!

One of the first steps we take in *Excel* is the **sorting** of our data sets. Whether that be the sorting of plots, or dates, or anything; we start by sorting. The same is possible in R. We do this using the **arrange** function.

In this example, we will be sorting by bill length in an increasing order (smallest to largest). Notice here that R will *default* to the order of small-large with the **arrange** function.

```
  # select() and arrange()
  penguins %>%
    select(species, bill_length_mm, year) %>%
    arrange(bill_length_mm) #I want to look at bill length in an increasing order from
```

```
# A tibble: 344 x 3
   species bill_length_mm  year
```

```
   <fct>             <dbl> <int>
 1 Adelie             32.1  2009
 2 Adelie             33.1  2008
 3 Adelie             33.5  2008
 4 Adelie             34    2008
 5 Adelie             34.1  2007
 6 Adelie             34.4  2007
 7 Adelie             34.5  2008
 8 Adelie             34.6  2007
 9 Adelie             34.6  2008
10 Adelie             35    2008
# ... with 334 more rows
```

In this example, we will be sorting by bill length in a decreasing order (largest to smallest). Notice here, we need the **arrange** function to tell R we will be changing the order. Once that command is established, we can further command the order.

In this example, I use the **arrange** function, followed by the **desc** function (descending), commanding the order of bill length to go from big to small values.

```
# select(), arrange(), and desc()
penguins %>%
  select(species, bill_length_mm, year) %>%
  arrange(desc(bill_length_mm)) #using the desc() function to command the order from
```

```
# A tibble: 344 x 3
   species     bill_length_mm  year
   <fct>                <dbl> <int>
 1 Gentoo               59.6  2007
 2 Chinstrap            58    2007
 3 Gentoo               55.9  2009
 4 Chinstrap            55.8  2009
 5 Gentoo               55.1  2009
 6 Gentoo               54.3  2008
 7 Chinstrap            54.2  2008
 8 Chinstrap            53.5  2008
 9 Gentoo               53.4  2009
10 Chinstrap            52.8  2008
# ... with 334 more rows
```

Now, let's say we want to see bill length in the same descending order, but we want to order this by year. This is done with a very simple addition to our *arrange()* section. To accomplish

this, we add the year variable first (remembering the the default for **arrange** is small-large) followed by the bill length command (which is the same as the previous example.)

```
# select() and arrange()
penguins %>%
  select(species, bill_length_mm, year) %>%
  arrange(year, desc(bill_length_mm)) #year and bill separated by a comma
```

```
# A tibble: 344 x 3
   species   bill_length_mm  year
   <fct>              <dbl> <int>
 1 Gentoo              59.6  2007
 2 Chinstrap           58    2007
 3 Chinstrap           52.7  2007
 4 Chinstrap           52    2007
 5 Chinstrap           52    2007
 6 Chinstrap           51.7  2007
 7 Chinstrap           51.3  2007
 8 Chinstrap           51.3  2007
 9 Chinstrap           51.3  2007
10 Chinstrap           50.6  2007
# ... with 334 more rows
```

filter()

Within R, we also have the ability to subset out data sets and pull out rows with specific values. Let's say I *only* want to look at data from *2007.* To accomplish this, we will use the **filter** function.

In this example, we will be adding the **filter** function as well as recall our knowledge of operators within R.

```
# select() and filter()
penguins %>%
  select(species, bill_length_mm, bill_depth_mm, year) %>%
  filter(year == 2007) #using the '==' operator to show everything with the year 2007
```

```
# A tibble: 110 x 4
   species bill_length_mm bill_depth_mm  year
   <fct>            <dbl>         <dbl> <int>
 1 Adelie            39.1          18.7  2007
```

```
 2 Adelie             39.5           17.4  2007
 3 Adelie             40.3           18    2007
 4 Adelie             NA             NA    2007
 5 Adelie             36.7           19.3  2007
 6 Adelie             39.3           20.6  2007
 7 Adelie             38.9           17.8  2007
 8 Adelie             39.2           19.6  2007
 9 Adelie             34.1           18.1  2007
10 Adelie             42             20.2  2007
# ... with 100 more rows
```

What if I want to see which penguins have bill lengths that are higher than the average length, but *only* from 2007? This can be accomplished by, again, adding an operator, but also calling another function. We will command R further with **mean** function from base R. Notice I am separating each line in the **filter** function with a comma. This allows me to add multiple commands within the same function.

```
# select(), filter(), and mean()

penguins %>%
  select(species, bill_length_mm, year) %>%
  filter(
    year == 2007,#using the '==' operator to show everything with the year 2007
    bill_length_mm > mean(bill_length_mm), # using '>' to view every row where the bill le
  )
```

```
# A tibble: 0 x 3
# ... with 3 variables: species <fct>, bill_length_mm <dbl>, year <int>
```

Let's say we are interested in manipulating our data set by species.
I want to know how many species I have to further filter this set. To accomplish this, I will use the the **count** function to view how many species I have and their associated values within the data set.

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

```
# A tibble: 3 x 2
  species        n
  <fct>      <int>
```

```
1 Adelie       152
2 Chinstrap     68
3 Gentoo       124
```

It appears there are three species within my data set. For one reason or another, I want to filter out *Adelie* from further interpretations. To do this, I will add another line below the bill length filter.

This new line says *when species equals Chinstrap OR Gentoo*, keep them in the data set.

```
# select() and filter()
penguins %>%
  select(species, bill_length_mm, year) %>%
  filter(
    year == 2007,#using the '==' operator to show everything with the year 2007
    bill_length_mm > mean(bill_length_mm),# using '>' to view every row where the bill len
    species == "Chinstrap" | species == "Gentoo" # look in species and pull out chinstrap
  )
```

```
# A tibble: 0 x 3
# ... with 3 variables: species <fct>, bill_length_mm <dbl>, year <int>
```

Another way to accomplish the same task is to tell R which values to *exclude*, rather than *include*. This is done by using the 'does not equal' operator to command R to return every species value that is not Adelie.

```
# select () and filter()
penguins %>%
  select(species, bill_length_mm, year) %>%
  filter(
    year == 2007,#using the '==' operator to show everything with the year 2007
    bill_length_mm > mean(bill_length_mm),# using '>' to view every row where the bill len
    species != "Adelie" #does not equal operator
  )
```

```
# A tibble: 0 x 3
# ... with 3 variables: species <fct>, bill_length_mm <dbl>, year <int>
```

```
#the output is the same!
```

Now that we only the data we want to see, let's create some new columns and row values. Let's say we want to add a new column combining *species and year* and a new column with the *rounded values of bill length.* We will be using the **mutate** function here. Along with this, we then want to *rearrange* our data set for viewing purposes of our new variables. This will be done with the **select** function.

In this example, I have created the column 'sp_year' which will contain both species and year, but keep their respective values separated by a dash. I then created a new column of the rounded bill length values using the **round** function. *Notice* with these new columns, the first step is to name the new column and then command R what to put in. Lastly, using the **select** function, I command R to order this data set as follows.

```
# select(), mutate(), select()
penguins %>%
  select(species, bill_length_mm, year) %>%
  mutate( #mutate()
    sp_year = paste(species, "-", year), #adding a new column named 'sp_year' and pasting
    rn_bill_length_mm = round(bill_length_mm) #creating a column of rounded bill lengths
  ) %>%
  select(species, year, sp_year, bill_length_mm, rn_bill_length_mm) #placing these new col
```

```
# A tibble: 344 x 5
   species  year sp_year        bill_length_mm rn_bill_length_mm
   <fct>   <int> <chr>                   <dbl>             <dbl>
 1 Adelie   2007 Adelie - 2007            39.1                39
 2 Adelie   2007 Adelie - 2007            39.5                40
 3 Adelie   2007 Adelie - 2007            40.3                40
 4 Adelie   2007 Adelie - 2007            NA                  NA
 5 Adelie   2007 Adelie - 2007            36.7                37
 6 Adelie   2007 Adelie - 2007            39.3                39
 7 Adelie   2007 Adelie - 2007            38.9                39
 8 Adelie   2007 Adelie - 2007            39.2                39
 9 Adelie   2007 Adelie - 2007            34.1                34
10 Adelie   2007 Adelie - 2007            42                  42
# ... with 334 more rows
```

summarize()

Now that we are confident in our wrangling, we can investigate some summary statistics.

First, let's look at the means and standard deviations of both bill length and depth. This will be done by name new columns and then using either the **mean** function or **sd** function to

produce a desired output.

```
# summarize ()
penguins %>%
  summarize( #summarize to run summary stats
  bill_length_mean = mean(bill_length_mm), #new column with mean value of bill length
  bill_length_sd = sd(bill_length_mm), #new column with standard deviation value of bill l
  bill_depth_mean = mean(bill_depth_mm), #new column with mean value of bill depth
  bill_depth_sd = sd(bill_depth_mm) #new column with standard deviation value of bill dept
  )
```

```
# A tibble: 1 x 4
  bill_length_mean bill_length_sd bill_depth_mean bill_depth_sd
            <dbl>          <dbl>          <dbl>          <dbl>
1               NA             NA              NA             NA
```

The last output was informative, but lets look a little deeper. I now want to group these new values by species. Using the **group_by** function, we can tell R to group our data set by one, or more variables.

In this example, I am telling R to **group_by** species, and then provide me with the means and standard deviations of bill length and depth.

```
# group_by() and summarize()
penguins %>%
  group_by(species) %>% #grouping by one column, species
  summarize( #summarize to run summary stats
    bill_length_mean = mean(bill_length_mm), #new column with mean value of bill length
    bill_length_sd = sd(bill_length_mm), #new column with standard deviation value of bill
    bill_depth_mean = mean(bill_depth_mm), #new column with mean value of bill depth
    bill_depth_sd = sd(bill_depth_mm) #new column with standard deviation value of bill de
  )
```

```
# A tibble: 3 x 5
  species   bill_length_mean bill_length_sd bill_depth_mean bill_depth_sd
  <fct>                <dbl>          <dbl>          <dbl>          <dbl>
1 Adelie                  NA             NA              NA             NA
2 Chinstrap             48.8           3.34            18.4           1.14
3 Gentoo                  NA             NA              NA             NA
```

Following the trend of the last example, let's further group our data set. I want to now see these same values but by species AND year. Using the **group_by** function again, we can accomplish this.

In this example, the only change is I added ', year' into my **group_by** function.

```
# group_by() and summarize()
penguins %>%
  group_by(species, year) %>% #grouping by two columns, species and year
  summarize( #summarize to run summary stats
    bill_length_mean = mean(bill_length_mm), #new column with mean value of bill length
    bill_length_sd = sd(bill_length_mm), #new column with standard deviation value of bill
    bill_depth_mean = mean(bill_depth_mm), #new column with mean value of bill depth
    bill_depth_sd = sd(bill_depth_mm) #new column with standard deviation value of bill de
  )
```

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

```
# A tibble: 9 x 6
# Groups:   species [3]
  species    year bill_length_mean bill_length_sd bill_depth_mean bill_depth_sd
  <fct>     <int>            <dbl>          <dbl>           <dbl>         <dbl>
1 Adelie     2007               NA             NA              NA            NA
2 Adelie     2008             38.6           2.98            18.2          1.09
3 Adelie     2009             39.0           2.56            18.1          1.24
4 Chinstrap  2007             48.7           3.47            18.5          1.00
5 Chinstrap  2008             48.7           3.62            18.4          1.40
6 Chinstrap  2009             49.1           3.10            18.3          1.10
7 Gentoo     2007             47.0           3.27            14.7         0.908
8 Gentoo     2008             46.9           2.64            14.9         0.985
9 Gentoo     2009               NA             NA              NA            NA
```

8.3 Practice on your own

First, take the penguins_raw data set and get it as close to the cleaned data set, penguins, as you can. Use the methods we went over in this section, and apply changes where needed. **Hint**, the is.na section is already $_{done}$. Once complete, continue to the next practice assignment.

Now that we have worked through some examples with Palmer Penguins, let's try and work through a data set of our own.

Attached here is a Slug data set.

Remember, you will need to import this file into R in the correct format!

Your task is to [1] input it into R, [2] investigate the variables and classes of these variables, [3] produce an output using *each* of the functions we just covered, [4] and at least *one* example where you use **select, rename, arrange, filter, mutate, and group_by** in the same command line. In part 3, for each change to the data set, save the changed data set as a new object. For part 4, save this object as, 'Final_Changes'. If you conduct more than one iteration of part 4, add the associated number at the end of each name. For example, Final_Changes_1, Final_Changes_2, etc.

Part IV

Plotting along: IN BETA

9 Testing your skills: ggplot

Recreate these plots to the best of your ability

In this section, we will be using the **mtcars** data set. Let's start by reading in **ggplot** and the **mtcars** data set.

```
library(ggplot2)
data("mtcars")
```

For more information about this data set, look in the *help* section.

```
?mtcars()
```

Now, let's take a look a little closer at the data set. Which 'class' do these variables belong to? Are they numeric? Characters? Integers?

This knowledge is very impoartant when building your plots.

```
View(mtcars)
str(mtcars)
```

Hints

Here is an example of how to change the variable type inside of the ggplot lines **and** outside. Notice here, we are calling the *am* column with the **$** operator.
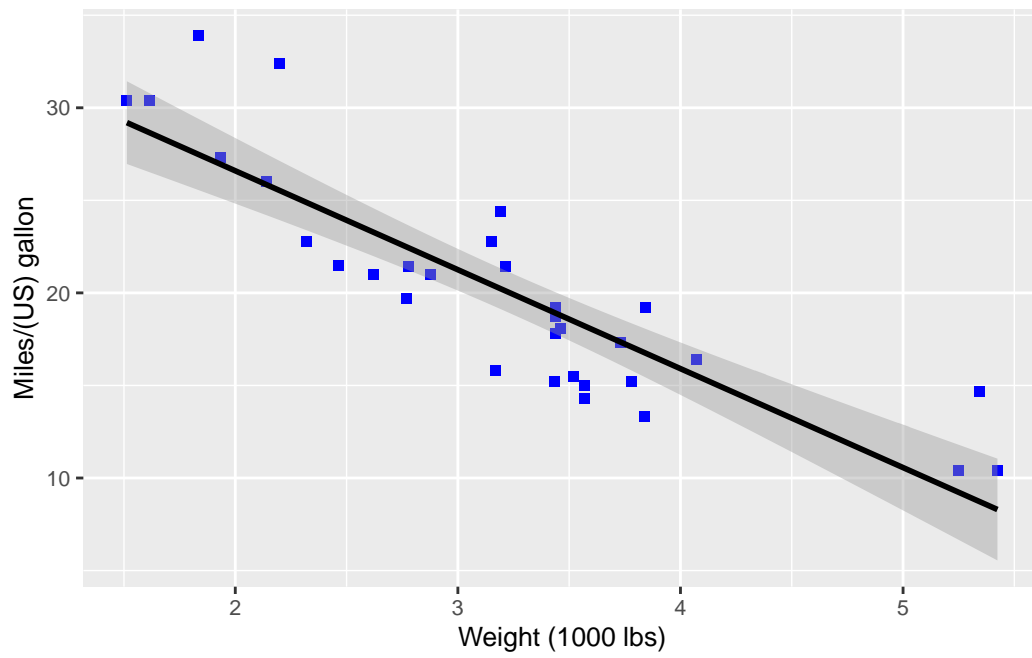
```
ggplot(mtcars, aes(x = as.factor(am)))
```

```
mtcars$am <- as.factor(mtcars$am)
```

You may also need to search for ggplot format help on these. Please see the extra resources at the bottom of this page.
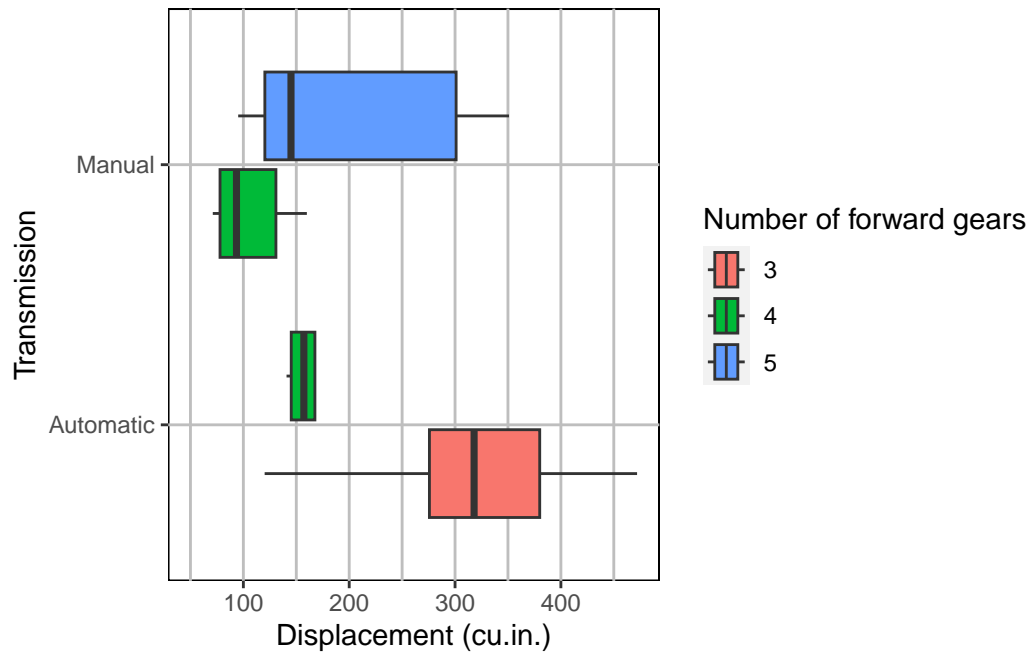
Plot time

Graph 1: geom_point

Graph 2: geom_boxplot

```
[1] "0" "1"
```

More to come!! 6/9/2023

Additional resources

[Lecture notes](#)

[Basic R graphics guide](#)

[ggplot online textbook](#)

Part V

Last part

10 Resources and cheat sheets

Cheat sheet links

Data Table

dplyr and tidyr

ggplot

11 Terms of endearment

To whom it may concern

Shout out to Daniel Bliss for listening to me think loud (and complain) during this process and also for being the first person to proof read.

Chief editor: Daniel D'Bliss

Editor: Shea Tillotson

GitHub troubelshooter: Callum Arnold

References

What is R?

1. nd. "What is R?" *The R Foundation.* (link)

RStudio vs r

2. nd. "R and RStudio" *BYU: Department of Statistics.* (link)

Excel vs r

3. Abrahams, Amieroh. February 23,2023. "Why should I use R: The Excel R Data Wrangling comparison" *jumping rivers.* (link)

Style

4. nd. "The tidyverse style guide" *tidyverse.* (link)

5. nd. "Google's R Style Guide" *styleguide.* (link)

6. https://towardsdatascience.com/an-introduction-to-the-pipe-in-r-823090760d64 pipe