Introductory R Workbook

Kyle Butts

2025-08-21

Table of contents

Preface **4**

I Introduction **5**

1 Installing R for the first time **6**
1.1 Installing packages for this class 7

2 Files and Folders on your Computer **8**

3 Quarto documents **10**
3.1 Rendering to PDF 10

4 R as a Calculator **12**
4.1 R as a functional programming language 13
4.2 Giving Things Names (i.e. Creating Variables) 14
4.3 Glueing together strings 15

5 Vectors **17**
5.1 Summarizing vectors 18
5.1.1 Help menu 20
5.1.2 `NA`s 21
5.1.3 Logical Vectors 22
5.1.4 Subsetting of vectors by logical 23
5.1.5 Vectorized operations 24
5.1.6 Sorting data 25

6 Dataframes (or, a group of vectors) **29**
6.1 Accessing vectors by name with `$` 31
6.2 General subsetting with `[,]` 32
6.3 Selecting rows based on criteria 35
6.4 Creating new variables 39
6.4.1 Exericse 40
6.5 Loading data into R 40
6.6 Sorting dataframes 42

7 Plotting **43**
7.1 Histograms . . . 43
7.2 Scatter Plots . . . 44
7.3 (optional) ggplot2 . . . 46

8 Running regressions in R **50**
8.1 First regression by hand . . . 52
8.1.1 Forecasting . . . 53
8.2 First regression by function . . . 55
8.2.1 Forecasting . . . 57
8.2.2 Regression with indicator varaibles . . . 58
8.2.3 Interactions . . . 60

9 Time to work with Times **62**
9.1 Years . . . 62
9.2 Working with dates . . . 64
9.3 Dates in R . . . 66
9.3.1 Back to football dataset . . . 68
9.3.2 More `lubridate` functions . . . 70
9.4 Unemployment data . . . 72
9.4.1 Calculating autocorrelation of unemployment rate . . . 72
9.4.2 Quarters . . . 74
9.5 Basic time-series regression . . . 74
9.6 Exercise . . . 77

II Intermediate **78**

10 Function Arguments **79**
10.1 Advanced arguments notes . . . 80

Preface

This "workbook" serves as an opinionated introduction to R for data science workflows. My goal is to introduce everything from first principles so that someone installing R for the first time can get up to speed. While not a complete tour of all that R is able to do, with the "Introduction" part of the book, you shuld be able to complete my class assignments.

This book is intended to be used interactively! There are many exercises scattered throughout. Some are focused to help with comprehension, while others are meant to build your confidence in exploring data to answer questions.

For those of you who would like to learn more, the Intermediate and Advanced guides will be contain extra information that continues to build up your programming skills. These will take longer to complete, but will grow over time.

With that, we can continue to our first topic: getting your computer ready for installing R and a code-editor to work in.

Part I

Introduction

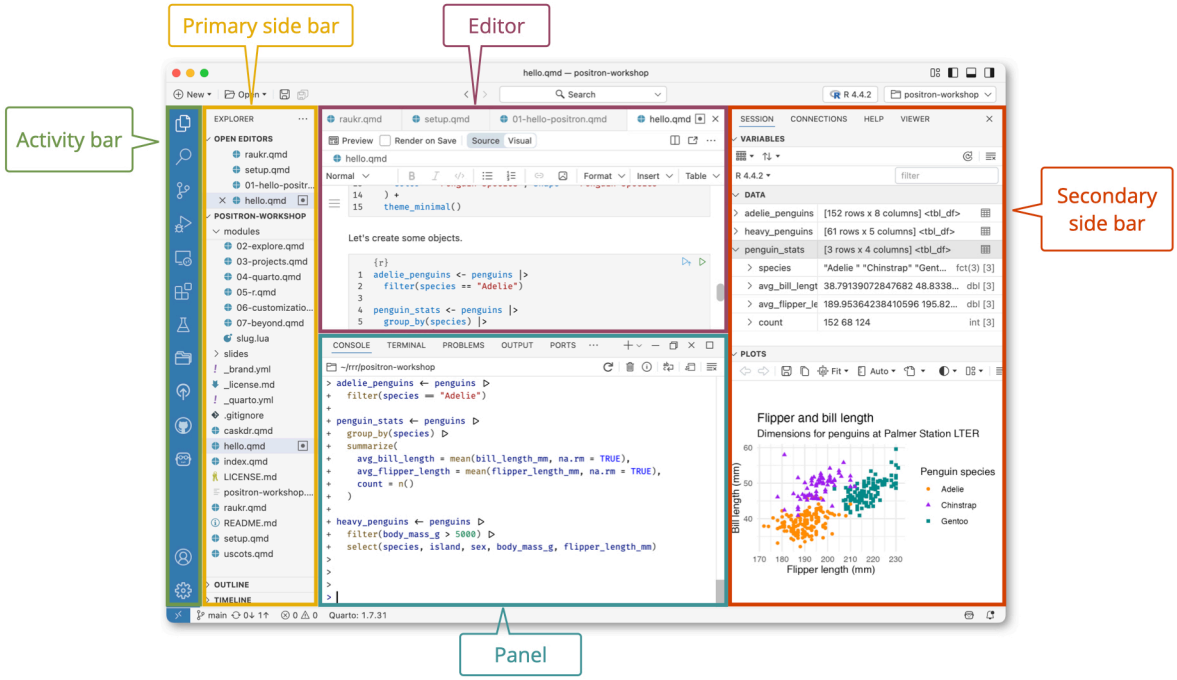
1 Installing R for the first time

The first part of the assignment is aimed at getting your computers set up for the rest of the course. You need to download two different softwares.

1. R is the statistical software that we will use for this course.
2. Positron is a visual code editor that makes editing and running code way easier.

You will know that you have installed R and Positron correctly if when you start Positron, you see a message in the Console tab that says 'R 4.5.1 started'.

There are various parts of the Poistron app and it can be overwhelming at first! Here's a breakdown from the Positron website:



- The **Activity Bar** provides quick access to switch between core views such as Explorer, Search, Source Control, and Extensions.

- The **Primary Side Bar** is on the left by default and shows you different views depending on what you have chosen in the Activity Bar. When you choose the Explorer icon, this pane provides the file explorer to navigate your project directory and the outline. When you choose the Assistant icon, this pane provides access to Positron Assistant.
- The **Editor** is in the center by default, and is where you write your code. For editor controls, refer to VS Code Editor Basics.
- The **Panel** is below the editor by default and contains the fully interactive, integrated Console as well as the Terminal You can also access logs from Output channels in the Panel.
- The **Secondary Side Bar** is on the right by default. You can switch between the Session pane (where you can explore the variables you have defined and the plots you have created), the Connections pane, the Help pane, and the Viewer pane.

The **Title Bar** at the very top of the window shows the active file and project, along with window controls. Below it, the **Top Bar** provides global project tools such as file search, the project switcher, and the interpreter selector with the ability to start, stop, and switch interpreters. The **Status Bar** at the bottom of the window displays details such as your git branch, language mode, Quarto version, and cursor position.

1.1 Installing packages for this class

Good job! Now, we have to install two packages. Copy the following lines into the "Console" and hit `Enter`. This will take a minute or two, but once you have done this you will be ready for the semester.

```
install.packages("tidyverse")
install.packages("fixest")
install.packages("fpp3")
install.packages("patchwork")

# This will install LaTeX on your system (for compiling qmd to pdf)
install.packages("tinytex")
tinytex::install_tinytex()
```

2 Files and Folders on your Computer

If you are not familiar with the file system, here is a brief introduction in this and the following paragraph.[1] It is important to understand how computers organize and store data. A computer's file system is essentially a large hierarchical structure that allows you to store and retrieve files and folders in an organized manner. At the top of this hierarchy is the root directory, which can be thought of as the main folder that contains everything else on your computer.

Within this root directory, there are numerous folders, each of which can contain other folders (known as subfolders) and individual files. Think of these folders as filing cabinets where you can store related documents together. For instance, when you download files from the internet, they typically go to the `Downloads` folder by default. However, it is crucial to not let them remain there indefinitely! If your `Downloads` folder becomes cluttered with hundreds of files, it becomes difficult to locate what you need. To maintain order, you should regularly organize your files into appropriate folders. For this class, it is recommended that you create a specific folder that contains *all* the relevant files, making it easier to find and manage your coursework.

Once you've created and organized your class folder, navigating to it within Positron is straightforward. In the left-hand side, go to the Explorer tab and click "Open Folder". Browse through your computer's directories to find your class folder and hit open. This action tells Positron to use this folder as the default location for loading and saving files. By setting your working directory correctly, you ensure that any data files stored in this folder can be easily accessed when you run your R scripts, streamlining your workflow and helping to avoid errors related to file paths.

When working in Positron, it's important to understand the concept of relative file paths, especially after setting your working directory. A relative file path is a way to specify the location of a file in relation to your current working directory, rather than using an absolute path that starts from the root directory. For example, if your working directory is set to your class folder, and you have a data file named `data.csv` stored inside a subfolder called `datasets`, you can load this file in R by using the relative path `datasets/data.csv` instead of the full path (e.g. `~/kylebutts/Documents/UARK\_4753/datasets/data.csv`). This is recommended for two reasons. First, when collaborating with people because they do not have the same root directory as you (only I have `~/kylebutts/`). Second, you might want to

[1] You can watch this three part tutorial if you're very new to computers: https://www.youtube.com/watch?v=k-EID5_2D9U.

move files around; with relative paths, as long as you keep everything in a project in a folder, you can move that folder as you wish.

3 Quarto documents

In this class, assignments will be completed in Quarto documents. Quarto documents are a method of creating *reproducible* reports in either html or pdf formats. The document contains written text intermixed with code that can create tables and/or figures. This can be incredibly useful in your future careers. For example, you might create a weekly sales report that you can update each week by changing the data input.

Quarto might feel confusing at first, so the first thing we will do is have you read through some resources to familiarize yourself with Markdown and quarto. To begin, read the following:

1. Read this guide on Using Quarto
2. Read this guide on Using Markdown

After reading these, download and put `Lab_0.qmd` in your class folder. Then, open positron, open your class folder, and finally open the quarto file.

When completing assignments, you can run individual blocks as you work to make sure the results come out correctly. You can do this in a few ways. First, you could copy the text from the quarto code chunk (not including the "‘ parts) and paste into the console. This is the least effective method but works. Alterantively, you could hit the "Run Cell" button. This will run the code and display the results below the code chunk.

The third option is for experts. Instead of using your mouse and clicking the button, you could instead use the keyboard shortcut `Cmd + Enter` on Mac or `Ctrl + Enter` on Windows. This will run whatever code cell the cursor currently is within.

3.1 Rendering to PDF

Above your quarto file, you can see a little newspaper icon and the word **Preview**. This is how you render your quarto file to an output report. After successfully rendering, the rendered output will show up in the **Viewer** tab.

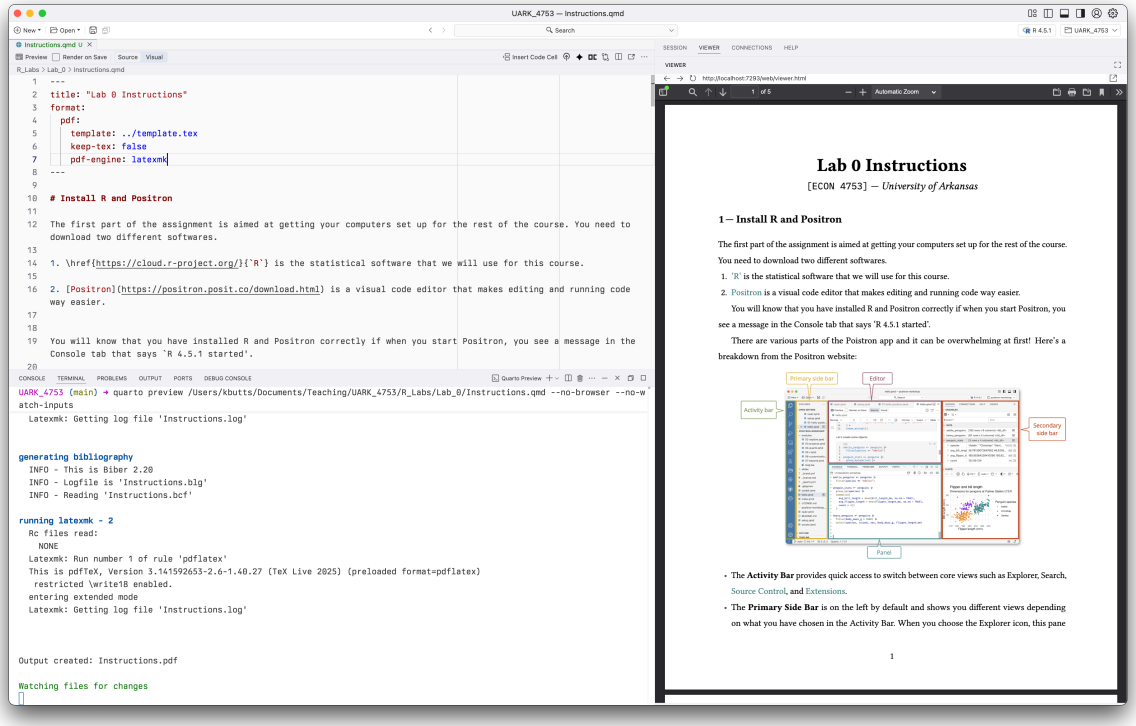


Figure 3.1: Example of Quarto Rendering

> **Note**
>
> You should be careful to include all the code in the quarto file. The reason is that when you go to **Render** the quarto file, it is creating a *completely new R instance* and running the code from the top of the quarto file to the bottom. So you should be careful to not use variables that are not defined *above* the cell.
> The most common problem students face is loading a csv in the console and then not including that in the quarto file.

4 R as a Calculator

The first thing we will learn is how to use R as a calculator. You can use any of the math operators you want:

- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `^` Exponentiation

Let's experiment with some arithmetic expressions:

```
1 + 1
```

```
[1] 2
```

```
2 / 4
```

```
[1] 0.5
```

```
3^2
```

```
[1] 9
```

Order of operations (via PEMDAS) apply here too:

```
5 + 2 * 3
```

```
[1] 11
```

This does 2 * 3 first and then adds 5 to get 11. If we want to do 5 + 2 first, then we can wrap it in parenthesis (the P in PEMDAS):

```
(5 + 2) * 3
```

```
[1] 21
```

There are even some operators that you might not know that have to do with remainders:

```
# Get the remainder
13 %% 2
```

```
[1] 1
```

```
# Divide and round down to the nearest decimal
13 %/% 2
```

```
[1] 6
```

4.0.0.1 Exercise

Compute the sample average of the following sample of baby weights (in lbs.):

(7.7, 8.2, 8.3, 7.6, 9.2, 7.4, 11.1)

4.1 R as a functional programming language

R is based around functions. A function takes an input (or multiple inputs) and produces an output. There are many many functions in R, but first lets learn some calculator type functions. For example, if I want to take the square root, I can use the function `sqrt`. Here are some example of math functions:

The form of a function call is `function_name(arguments)`

1. The function name, `sqrt`, `abs`, `factorial`
2. Opening parenthesis `(`
3. The argument (in the future arguments)
4. Closing parenthesis `)`

For example `sqrt(16)` says to take the argument `16` and apply the function `sqrt` of it.

4.1.0.1 Exercise

1. Calculate the square root of 147
2. Try finding the natural log of 10, using `log()`
3. Practical usage: say the $Var(x) = 12$ and we have a sample size of 55. What is the standard deviation of the sample distribution of the sample mean?

4.2 Giving Things Names (i.e. Creating Variables)

Variables are immensely helpful in R. It lets you store values by giving them a `name` and then lets you access the variables later by name. I can assign variables using either `<-` or `=`.

Create variable `x` with value 5 and a variable `y` with value 20.

```
x <- 5
y <- 20
```

What is the sum of `x` and `y`?

```
x + y
```

```
[1] 25
```

Note the form of creating the variable:

1. The variable name, `x` and `y`
2. Assignment operator `<-` or `=`
3. The value we want to store.

The reason behind the left arrow is that the arrow points to variable name where we want to put the value into.

You can create a variable containing *text* by using `""`

```
instructor_name <- "Kyle Butts"
print(instructor_name)
```

```
[1] "Kyle Butts"
```

We can use the `cat` command to print out the text to the console:

```
cat(instructor_name)
```

```
Kyle Butts
```

4.2.0.1 Exercise

Use quotation marks to create a string and call it `my_name`.

```
my_name <- "Kyle Butts"
my_name
```

```
[1] "Kyle Butts"
```

4.3 Glueing together strings

Often time we might want to combine text from multiple sources and/or add data to our strings. We can use the `paste`/`paste0` functions to combine together strings. `paste0` will append the strings as written, while `paste` will automatically add a space between each thing it is concatenating. Dealer's choice for which you prefer

For example,

```
paste0("Kyle", "Butts")
```

```
[1] "KyleButts"
```

```
paste("Kyle", "Butts")
```

```
[1] "Kyle Butts"
```

What is cool about these functions is they take any number of arguments and append them together. For example, we can write a paste function that takes your height in inches and prints a human-readable string.

```r
height_in_inches <- 70
paste0("My height is ", height_in_inches %/% 12, "'", height_in_inches %% 12, "\"")
```

```
[1] "My height is 5'10\""
```

Note that numbers get automatically converted to a string. One subtle point you might have missed. If we start and end strings with double quotes, how can we include one in the text itself? Above, we did this with the escape key `\"`. You probably won't need to do this, but it's worth mentioning nevertheless.

4.3.0.1 Exercise

1. Construct a string that reports on the average baby weight in your sample. It would be nice to create a variable that stores `mean_baby_weight` to make the code nicer to read. These were the weights: `(7.7, 8.2, 8.3, 7.6, 9.2, 7.4, 11.1)`
2. Try printing out the following string: `\U1F919`. What is displayed?

5 Vectors

So far, we have dealth with scalar numbers and texts. But, when working with data, we will observe many units and need a way to store all their values together. This is where the bread and butter of data-science comes in: Vectors.

Vectors are a list of elements like integers, numbers, or strings. This is really useful for storing data! You use `c` to create a vector (pneumonically, `c` stands for combine).

```
## Rebounds from 2023 NBA Season
rebounds <- c(260, 114, 252, 310, 165, 236, 148, 336, 941, 127, 384, 278, 300, 6, 136, 145, 1
```

It is kind of a pain in the neck to write all these out; and worse, prone to errors! Later, we will learn how to load data from a file, making this much easier.

You can access elements of a vector by using `[#]`, where `#` is the i-th element you want

```
rebounds[1]
```

```
[1] 260
```

```
rebounds[2]
```

```
[1] 114
```

If you want to access more than one element, we can subset using *a vector!* (how meta):

```
rebounds[c(1, 2)]
```

```
[1] 260 114
```

One special syntax is ther `a:b` which generates $a, a+1, \ldots, b$. This makes it easy to grab the first 5 values:

```
rebounds[1:5]
```

```
[1] 260 114 252 310 165
```

Standard math operators work on vectors element by element:

```
rebounds[1:5] + 1
```

```
[1] 261 115 253 311 166
```

```
rebounds[1:5] / 12 # dozens of rebounds
```

```
[1] 21.66667  9.50000 21.00000 25.83333 13.75000
```

If we want to know how many elements are in a vector, we can use the `length` function:

```
length(rebounds)
```

```
[1] 386
```

5.1 Summarizing vectors

The natural next step is to start trying to summarize the data. There are a set of built-in functions that provide statistical summaries of the data.

```
## Mean, standard deviation, and variance
mean(rebounds)
```

```
[1] 250.4663
```

```
sd(rebounds)
```

```
[1] 232.6362
```

```r
var(rebounds)
```

```
[1] 54119.62
```

```r
## Extremes
max(rebounds)
```

```
[1] 1530
```

```r
min(rebounds)
```

```
[1] 0
```

```r
## chaining functions
sqrt(var(rebounds))
```

```
[1] 232.6362
```

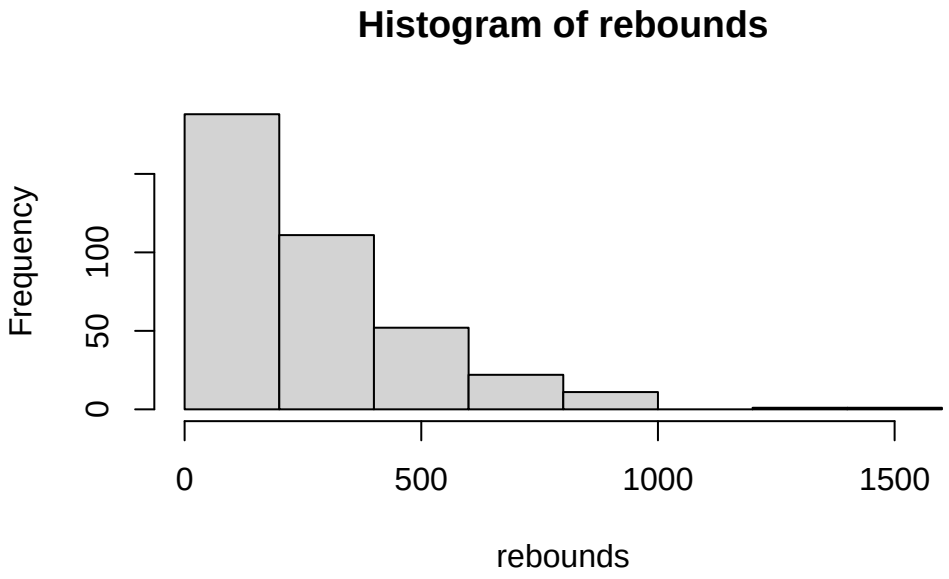
Some functions will take extra **arguments** that give more instructions. For example, the `quantile` function returns information about percentiles of the distribution. You can add extra arguments, separated by commas. For example, `quantile`'s first argument is a vector and the second argument is what percentiles you want to find.

```r
## Percentiles of the data
quantile(rebounds, c(0.1, 0.5, 0.9))
```

```
  10%   50%   90%
 19.0 205.5 575.5
```

We will talk in more details about plotting below, but for now we can use the `hist()` function to give us a view of the distribution of the vector

```r
hist(rebounds)
```



5.1.1 Help menu

How did I know the order of arguments for `quantile`? Are there ways to customize the histogram created by `hist`? In general, if you have a question about a function, then you need to address that function's **documentation**. To do so, click in to your console and type `?func_name` where `func_name` is the name of the function, e.g. `quantile`. At first, the information may be very overwhelming. The documentation in base R functions are very detailed, but are not great at introducing the functions. You should focus on the **Arguments** section and perhaps the **Examples** section, in my opinion. In particular, the order of the arguments is probably helpful in order of importance

5.1.1.1 Exercise

Similar to `c`, the `seq` function creates a vector: a **seq**uence of numbers.

1. Create a sequence of all multiples of four from 4 to 100. Look at '`?seq` for help. Hint: The arguments you need here are `from`, `to`, and `by`. Store your vector in a varaible
2. Find the 19th element of this sequence
3. What is the sum of the 10th and 11th element of this sequence.

5.1.2 NAs

In the real world, some times we will not have a value for a variable for an individual (e.g. people don't fill answer a survey question). In R, this is represented as an NA.

```
reviews <- c(5, NA, 4, 4, 3, 5,  NA, 4, 5, 2)
```

What is the average (mean and median) review?

```
mean(reviews)
```

```
[1] NA
```

```
median(reviews)
```

```
[1] NA
```

By default, the statistical summary functions will all produce NA when they are present in the data. R wants you to *opt-in* to ignoring the missings. To do this, functions will take an extra **argument** called na.rm:

```
mean(reviews, na.rm = FALSE)
```

```
[1] NA
```

```
max(reviews, na.rm = FALSE)
```

```
[1] NA
```

A lot of this information will be presented to you by the summary function. For numberic vectors, summary produces the five-number summary, the mean, and the number of NAs (if any)

```
summary(reviews)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   2.00    3.75    4.00    4.00    5.00    5.00       2
```

5.1.3 Logical Vectors

So far, we have seen two kinds of vectors: numeric and character vectors. A third, common, vector is a **logical** vector:

```
ordered_takeout <- c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, FALSE)
```

Logical vectors can take only two values: `TRUE` and `FALSE`. They can be treated as numbers by using `as.numeric()` with `TRUE` becoming `1` and `FALSE` becoming `0`.

```
as.numeric(ordered_takeout)
```

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

One common trick is to use `sum()` on a logical vector and it will return the number of `TRUE`s:

```
sum(ordered_takeout)
```

```
[1] 4
```

Logical vectors are often returned by other operations. For example, we can check whether elements of a vector equal some value with `==`. Other operators that produce a logical vector include `>`, `<`, `>=`, and `<=`.

```
## Find five-star reviews
reviews == 5
```

```
 [1]  TRUE    NA FALSE FALSE FALSE  TRUE    NA FALSE  TRUE FALSE
```

```
## Find three-star or lower
reviews <= 3
```

```
 [1] FALSE    NA FALSE FALSE  TRUE FALSE    NA FALSE FALSE  TRUE
```

Logical vectors can be **negated** using the `!` operator:

```
## Find non five-star reviews
!(reviews == 5)
```

```
 [1] FALSE    NA  TRUE  TRUE  TRUE FALSE    NA  TRUE FALSE  TRUE
```

You can also use the following operators to supply multiple criteria:

- `&` And operator. Both vector 1 **and** vector 2 must be true for the observation
- `|` Or operator. **Either** vector 1 **or** vector 2 must be true for the observation

```
(reviews == 5) & (reviews == 4)
```

```
 [1] FALSE    NA FALSE FALSE FALSE FALSE    NA FALSE FALSE FALSE
```

```
(reviews == 5) | (reviews == 4)
```

```
 [1]  TRUE    NA  TRUE  TRUE FALSE  TRUE    NA  TRUE  TRUE FALSE
```

```
## Equivalent to
reviews >= 4
```

```
 [1]  TRUE    NA  TRUE  TRUE FALSE  TRUE    NA  TRUE  TRUE FALSE
```

5.1.4 Subsetting of vectors by logical

We have already discussed one way of subsetting vectors via a vector of integers. This is called "subsetting by index".

The other common way is by using a logical vector the same length of the vector you wish to subset. For example, let's look at the reviews for takeout orders and dine-in orders separately

```
## Takeout orders
reviews[ordered_takeout]
```

```
[1]  5 NA  4  5  5  2
```

```
## Dine-in orders
reviews[!ordered_takeout]
```

```
[1]  4  3 NA  4
```

5.1.4.1 Exercise

1. What is the average review for take-out orders?
2. The `is.na()` function takes a vector as an argument and returns a logical vector that equals `TRUE` if the element is `NA`. Using this and the `!`, subset the reviews to only non-`NA` values. What is the average review? Comapre this to `mean(reviews, na.rm = TRUE)`.

```
mean(reviews[!is.na(reviews)])
```

```
[1] 4
```

```
mean(reviews, na.rm = TRUE)
```

```
[1] 4
```

5.1.5 Vectorized operations

Often times we want to use multiple vectors for some calculation. Like single numbers, arithmetic can be done element-by-element with vectors. This means +, -, *, / and ^ all work on a vector.

```
x <- c(1, 2, 3)
y <- c(5, 5, 5)

x + y
```

```
[1] 6 7 8
```

```
x^2
```

```
[1] 1 4 9
```

There are two main features to remember: 1. The vectors you use should be of the same length (if they are not, some weird *recycling* rules occur that we will not discuss in this introduction). 2. Scalars are treated as a vector of the same length with that single number repeated for each element.

```
## Equivalent:
x + 2
```

```
[1] 3 4 5
```

```
x + rep(2, 3)
```

```
[1] 3 4 5
```

More, many functions are designed to be used on vectors element by element. All of the functions we used in the "Calculator" section do this:

```
exp(x)
```

```
[1]  2.718282  7.389056 20.085537
```

```
log(x)
```

```
[1] 0.0000000 0.6931472 1.0986123
```

```
sqrt(x^2)
```

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

5.1.5.1 Exercise

1. Try to guess the output of the following expression `2*x + y + 1`.

5.1.6 Sorting data

The final vector operation we will discuss is how to sort data; either ascending or descending in value. There are two ways to do this.

First, we can use the `sort()` function. The function takes a function and an optional `decreasing` argument. `decreasing` takes a logical `TRUE/FALSE` option. By default, it increases in values

```r
sort(rebounds, decreasing = TRUE)
```

```
  [1] 1530 1258  941  934  921  909  899  870  862  845  830  829  807  770  762
 [16]  760  744  740  739  728  705  704  700  691  672  670  665  660  639  634
 [31]  633  631  630  618  607  593  580  580  578  573  564  564  556  551  549
 [46]  546  536  513  512  511  500  497  494  491  485  485  483  476  473  472
 [61]  469  468  460  454  453  451  450  449  448  447  439  435  434  432  429
 [76]  426  423  420  417  416  415  410  407  405  402  402  401  394  394  393
 [91]  391  384  381  372  350  346  344  342  336  336  330  324  319  318  317
[106]  317  314  312  312  310  310  310  308  307  305  305  301  301  300  298
[121]  296  296  296  295  295  292  292  289  286  286  282  281  278  275  272
[136]  271  270  269  269  268  265  265  261  260  260  260  259  258  257  257
[151]  257  257  256  255  253  253  252  252  249  247  247  247  246  245  244
[166]  243  240  239  238  236  236  235  233  233  233  231  227  227  227  223
[181]  223  222  220  220  219  213  211  210  209  208  206  206  206  205  204
[196]  202  202  201  191  190  189  188  188  187  185  184  184  184  183  182
[211]  182  179  178  177  173  171  170  168  168  165  162  162  161  161  152
[226]  150  149  148  147  145  145  145  145  144  142  136  132  129  127  124
[241]  118  118  118  116  114  112  112  111  110  106  105  102  101  101   99
[256]   99   98   98   97   96   96   96   95   95   95   94   92   90   88   85
[271]   85   84   82   82   81   81   81   80   79   78   78   78   77   76   75
[286]   74   71   70   69   69   69   66   66   63   63   63   61   58   56   55
[301]   54   54   54   52   51   51   49   47   47   46   45   43   42   42   41
[316]   41   39   39   37   36   35   35   34   34   34   33   33   32   30   30
[331]   30   30   28   28   27   26   26   26   26   25   24   24   24   22   22
[346]   21   19   19   19   16   16   15   15   14   12   11   11   11   11   10
[361]   10    9    8    8    7    6    5    5    5    5    3    2    2    2    1
[376]    1    1    1    1    1    0    0    0    0    0    0
```

Alternatively, we can use the `order()` function to get the *row indices* of the ordering:

```r
order(rebounds, decreasing = TRUE)
```

```
  [1] 298 379   9 313 113 215 177 244 347 252  25 295 139 310 141  85 216 111
 [19]  98  58 258 371 250 209  88 245 181 130 255 317 183  82 272  71 373 158
 [37]  65 127  41 219 217 271 263 265  64 197 273  30 356 180  96  97 238 101
 [55]  23  39 213  49 235  79 349 227 228 372 234 195 240 332 249 150 108 330
 [73]  33 138 194 266  60  19 327 294 315  87 163 377 129 189  67 204 285 282
 [91] 320  11 344 364 292  62  44 144   8 291 229  63 287 233 179 283 184 174
[109] 262   4 148 311 205 106 169 385 286 319  13 103  32 190 303 261 369 175
[127] 176  93  61 123 376  43  12 159  91 152 333  36 381 230 334 359 193   1
```

```
[145] 131 384  83 268 140 210 290 363 323 275 109 348   3 232 208  92 342 378
[163] 353  40  56 221 383  34 361   6 366  37  17 214 270 134  90 107 203  55
[181] 362  38 120 212 297 116 105 115 117 218 162 198 370 188 316 104 318  53
[199] 331 124  72  99 325  70 146  74 136 165 339 126 178 248 100 321 277 299
[217] 149 112 307   5  31  42  22 202  66 118 284   7 289  16 119 243 374 122
[235]  18  15 225 375  10 300 132 206 309 324   2  27 171 246 242 280 128 157
[253]  76 358  46 155 151 167 257  95 259 306  69 145 276 367 207 352 355 156
[271] 211  52  47 237  26 322 368 173  48  80 182 354 147 326 386 251 281  89
[289]  51  73 137 196 312  29  94 201 114  68 125 350 241 260 301  54 164 279
[307] 187 302 338 191 267 170 143 314  81 154  45 264 351  77  20 199  57  75
[325] 226  86 236 185 172 224 253 304 329 360 254 121 192 239 305 220 110 142
[343] 160  35 341 365 288 293 345  28 222 256 380 308  84  21 135 186 278 161
[361] 168 382 153 269  78  14  59 200 274 335  50 223 296 346  24 102 231 247
[379] 328 336 133 166 337 340 343 357
```

The first element of the resulting vector is the index of the maximum number (1530)

```
order(rebounds, decreasing = TRUE)[1]
```

```
[1] 298
```

```
which.max(rebounds)
```

```
[1] 298
```

What this means is that we can use the result of `order` to subset the vector to get the sorted vector:

```
rebounds[order(rebounds, decreasing = TRUE)]
```

```
  [1] 1530 1258  941  934  921  909  899  870  862  845  830  829  807  770  762
 [16]  760  744  740  739  728  705  704  700  691  672  670  665  660  639  634
 [31]  633  631  630  618  607  593  580  580  578  573  564  564  556  551  549
 [46]  546  536  513  512  511  500  497  494  491  485  485  483  476  473  472
 [61]  469  468  460  454  453  451  450  449  448  447  439  435  434  432  429
 [76]  426  423  420  417  416  415  410  407  405  402  402  401  394  394  393
 [91]  391  384  381  372  350  346  344  342  336  336  330  324  319  318  317
[106]  317  314  312  312  310  310  310  308  307  305  305  301  301  300  298
[121]  296  296  296  295  295  292  292  289  286  286  282  281  278  275  272
[136]  271  270  269  269  268  265  265  261  260  260  260  259  258  257  257
```

```
[151] 257 257 256 255 253 253 252 252 249 247 247 247 246 245 244
[166] 243 240 239 238 236 236 235 233 233 233 231 227 227 227 223
[181] 223 222 220 220 219 213 211 210 209 208 206 206 206 205 204
[196] 202 202 201 191 190 189 188 188 187 185 184 184 184 183 182
[211] 182 179 178 177 173 171 170 168 168 165 162 162 161 161 152
[226] 150 149 148 147 145 145 145 145 144 142 136 132 129 127 124
[241] 118 118 118 116 114 112 112 111 110 106 105 102 101 101  99
[256]  99  98  98  97  96  96  96  95  95  95  94  92  90  88  85
[271]  85  84  82  82  81  81  81  80  79  78  78  78  77  76  75
[286]  74  71  70  69  69  69  66  66  63  63  63  61  58  56  55
[301]  54  54  54  52  51  51  49  47  47  46  45  43  42  42  41
[316]  41  39  39  37  36  35  35  34  34  34  33  33  32  30  30
[331]  30  30  28  28  27  26  26  26  26  25  24  24  24  22  22
[346]  21  19  19  19  16  16  15  15  14  12  11  11  11  11  10
[361]  10   9   8   8   7   6   5   5   5   5   3   2   2   2   1
[376]   1   1   1   1   1   0   0   0   0   0   0
```

This might seem like a bit silly, but it will prove useful when we want to sort multiple vectors at the *same time* based on one of the vectors.

6 Dataframes (or, a group of vectors)

Dataframes are a special object in R. A dataframe is simply a collection of **vectors** and looks like a typical excel spreadsheet. The columns of a dataframe are each **vectors** that contain variables and a row contains an **observation**. This is the coding equivalent of an excel spreadsheet. If you are using Positron, clicking the dataframe in the `Variables` tab or typing `View(df_name)` into the console will let you interactively scroll though the data.

First, we will load some data.frames that come with a **package** in R. We can do that using the `data` function. Let's load the `penguins` data set which contain a census conducted on multiple

species of penguins on a set of islands:

```
data(penguins, package = "palmerpenguins")
```

We can use the `head()` function to view the first few rows. It prints out the first 6 rows of the dataset so you can see the variables. BTW, this function works on vectors too!

```
head(penguins)
```

```
# A tibble: 6 x 8
  species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
  <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
1 Adelie  Torgersen           39.1          18.7               181        3750
2 Adelie  Torgersen           39.5          17.4               186        3800
3 Adelie  Torgersen           40.3          18                 195        3250
4 Adelie  Torgersen           NA            NA                  NA          NA
5 Adelie  Torgersen           36.7          19.3               193        3450
6 Adelie  Torgersen           39.3          20.6               190        3650
# i 2 more variables: sex <fct>, year <int>
```

Another helpful function is `str()` which prints a similar format, but is a little easier to read, especially when there are a lot of variables in the dataset.

```
str(penguins)
```

```
tibble [344 x 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 output of string shows that a data.frame is just a list of vectors that have names (`species`, `island`, …).

Finally, you can access the number of rows and number of columns with `nrow`/`ncol`

```
print(paste0(nrow(penguins), " obs. of ", ncol(penguins), " variables"))
```

```
[1] "344 obs. of 8 variables"
```

6.0.0.1 Exercise

1. What constitutes a row in the penguins dataframe? What constitutes a column?

6.1 Accessing vectors by name with $

The first thing we might want to do is access some of the vectors from the `penguins` dataframe. If you type `species` into the console and hit enter, you will see this error: `Error: object 'species' not found`. Get used to recognizing this error becuase you will probably accidentally make this mistake a lot at first.

This errors occurs because R does not know that you mean "look for `species` in the `penguins` dataset". To access an individual vector, we must specify *both* the dataframe and the vector's name. To do this, we use the `$` operator like this: `penguins$species`. This looks into the dataframe stored in the `penguins` variable and looks for the vector named `species`:

```
penguins$species
```

```
  [1] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
  [8] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [15] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [22] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [29] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [36] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [43] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [50] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [57] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [64] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [71] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [78] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [85] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [92] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [99] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[106] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[113] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[120] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[127] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[134] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[141] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[148] Adelie    Adelie    Adelie    Adelie    Adelie    Gentoo    Gentoo
[155] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[162] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[169] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[176] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[183] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[190] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
```

```
[197] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[204] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[211] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[218] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[225] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[232] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[239] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[246] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[253] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[260] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[267] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[274] Gentoo    Gentoo    Gentoo    Chinstrap Chinstrap Chinstrap Chinstrap
[281] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[288] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[295] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[302] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[309] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[316] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[323] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[330] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[337] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[344] Chinstrap
Levels: Adelie Chinstrap Gentoo
```

When using Positron, after you type the `$` a little pop-up menu wil show up that shows you all the available variable names. You can use the arrow keys to find the variable and hit enter (or tab) to have it type the variable name for you.

Try that below to find the avariable measuring the depth of a penguin's bill:

6.2 General subsetting with `[,]`

Now, let's think about subsetting the dataframe to a specific set of rows. Given that we used `[idx]` for vectors (either a logical vector or integer index), you might think we can do the same with data.frames. This is good intuition, but remember a data frame has both rows and columns! We will still use `[,]` to subset, but we need a `,` to specify rows and columns.

The syntax therefore is `df[ROWS, COLUMNS]`. Before comma = rows and After comma = columns. If either `ROWS` or `COLUMNS` is left blank, then it will assume you want all rows or all colums respectively.

Let me show you some examples:

```r
## First 5 rows, all columns
penguins[1:5,]
```

```
# A tibble: 5 x 8
  species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
  <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
1 Adelie  Torgersen           39.1          18.7               181        3750
2 Adelie  Torgersen           39.5          17.4               186        3800
3 Adelie  Torgersen           40.3          18                 195        3250
4 Adelie  Torgersen           NA            NA                  NA          NA
5 Adelie  Torgersen           36.7          19.3               193        3450
# i 2 more variables: sex <fct>, year <int>
```

```r
## All rows, the species column
penguins[, "species"]
```

```
# A tibble: 344 x 1
   species
   <fct>
 1 Adelie
 2 Adelie
 3 Adelie
 4 Adelie
 5 Adelie
 6 Adelie
 7 Adelie
 8 Adelie
 9 Adelie
10 Adelie
# i 334 more rows
```

```r
## First 5 rows, the species column
penguins[1:5, "species"]
```

```
# A tibble: 5 x 1
  species
  <fct>
1 Adelie
2 Adelie
3 Adelie
```

```
4 Adelie
5 Adelie
```

```
## First 10 rows, the species and island columns
penguins[1:10, c("species", "island")]
```

```
# A tibble: 10 x 2
   species island
   <fct>   <fct>
 1 Adelie  Torgersen
 2 Adelie  Torgersen
 3 Adelie  Torgersen
 4 Adelie  Torgersen
 5 Adelie  Torgersen
 6 Adelie  Torgersen
 7 Adelie  Torgersen
 8 Adelie  Torgersen
 9 Adelie  Torgersen
10 Adelie  Torgersen
```

You can "chain" together calls and between `[,]` and `$` syntaxes. For example let's say I want the variable `island` for the first 6 observations:

```
penguins[1:6, "island"]
```

```
# A tibble: 6 x 1
  island
  <fct>
1 Torgersen
2 Torgersen
3 Torgersen
4 Torgersen
5 Torgersen
6 Torgersen
```

```
penguins[1:6, ]$island
```

```
[1] Torgersen Torgersen Torgersen Torgersen Torgersen Torgersen
Levels: Biscoe Dream Torgersen
```

But, if I do `penguins$island`, this results in a vector. So, to get the first 6 elements, I use only `[]` without a comma:

```
penguins$island[1:6]
```

```
[1] Torgersen Torgersen Torgersen Torgersen Torgersen Torgersen
Levels: Biscoe Dream Torgersen
```

6.2.0.1 Exercise

1. Use the `unique()` function to find the unique values of the variable `species` in the penguins dataset.
2. Use the `table()` function to find how many penguins there are of each species.
3. Imagine we use `penguins[1:5]` to try and grab the first 5 rows. What happens?

6.3 Selecting rows based on criteria

Lets see which penguins live on Torgersen.

We can use the `==` operator to compare a vector to a value (or set of values). For example, here we see if each penguin's island is `"Torgersen"`. It produces a `boolean` vector of TRUEs and FALSEs

```
on_torgersen <- penguins$island == "Torgersen"
on_torgersen
```

```
  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
 [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
 [73]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [85] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [97] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[109] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
[121]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
[145] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[157] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[169] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[181] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[193] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[205] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[217] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[229] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[241] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[253] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[265] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[277] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[289] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[301] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[313] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[325] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[337] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

We can subset the data using a boolen vector. We pass this boolean vector to the `ROWS` section of `[,]`

```
penguins[penguins$island == "Torgersen", ]
```

```
# A tibble: 52 x 8
   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
 1 Adelie  Torgersen           39.1          18.7               181        3750
 2 Adelie  Torgersen           39.5          17.4               186        3800
 3 Adelie  Torgersen           40.3          18                 195        3250
 4 Adelie  Torgersen           NA            NA                  NA          NA
 5 Adelie  Torgersen           36.7          19.3               193        3450
 6 Adelie  Torgersen           39.3          20.6               190        3650
 7 Adelie  Torgersen           38.9          17.8               181        3625
 8 Adelie  Torgersen           39.2          19.6               195        4675
 9 Adelie  Torgersen           34.1          18.1               193        3475
10 Adelie  Torgersen           42            20.2               190        4250
# i 42 more rows
# i 2 more variables: sex <fct>, year <int>
```

```
penguins[penguins$island == "Torgersen", ]$body_mass_g
```

```
 [1] 3750 3800 3250   NA 3450 3650 3625 4675 3475 4250 3300 3700 3200 3800 4400
[16] 3700 3450 4500 3325 4200 3050 4450 3600 3900 3550 4150 3700 4250 3700 3900
[31] 3550 4000 3200 4700 3800 4200 2900 3775 3350 3325 3150 3500 3450 3875 3050
[46] 4000 3275 4300 3050 4000 3325 3500
```

You can also use the following operators to supply multiple criteria: - `&` And operator. Both vector 1 **and** vector 2 must be true for the observation - `|` Or operator. **Either** vector 1 **or** vector 2 must be true for the observation

```
penguins$island == "Torgersen" & penguins$sex == "male"
```

```
  [1]  TRUE FALSE FALSE    NA FALSE  TRUE FALSE  TRUE    NA    NA    NA    NA
 [13] FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE
 [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE
 [73] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
 [85] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [97] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[109] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE
[121] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[145] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[157] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[169] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[181] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[193] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[205] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[217] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[229] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[241] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[253] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[265] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[277] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[289] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[301] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[313] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[325] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[337] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
penguins$island == "Torgersen" | penguins$sex == "male"
```

```
  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
 [25]  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
 [37]  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE    NA
 [49] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
 [61] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
 [73]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [85] FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE
 [97] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[109] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
[121]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[133] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[145] FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[157]  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[169] FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE    NA  TRUE
[181] FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[193] FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[205] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[217] FALSE  TRUE    NA  TRUE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE
[229] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[241] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE
[253] FALSE  TRUE FALSE  TRUE    NA  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[265] FALSE  TRUE FALSE  TRUE    NA  TRUE FALSE    NA FALSE  TRUE FALSE  TRUE
[277] FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[289] FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[301] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE FALSE
[313] FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE
[325]  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE FALSE
[337]  TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
```

6.3.0.1 Exercise

1. Subset the `penguins` dataframe to male penguins that weigh over 3500 grams
2. Are all three species present on the Torgersen island? The `table` or `unique` functions might be helpful here.
3. Which species has the heaviest penguin?
4.

6.3.0.2 Exercise

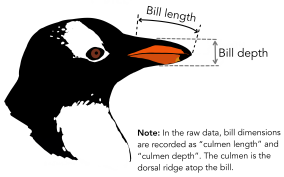


Figure 6.1: Penguin Beak Measurements

Lets compare Bill Length between the three species of penguins in our dataset: the Adelie, Gentoo, and Chinstrap penguins.

Using `[]` and `<-`, create 3 dataframes called `penguins_ad`, `penguins_gent`, and `penguins_chin` that subset based on the correct species of penguins.

"Adelie", "Gentoo", "Chinstrap"

What is the mean of each species bill length in millimeters? (remember there are NAs so we will need to use `na.rm = TRUE`).

6.4 Creating new variables

We can create *new* variables in our dataset by using the `$` or `[]` operators with `<-`.

```
penguins$body_mass_g
```

```
  [1] 3750 3800 3250   NA 3450 3650 3625 4675 3475 4250 3300 3700 3200 3800 4400
 [16] 3700 3450 4500 3325 4200 3400 3600 3800 3950 3800 3800 3550 3200 3150 3950
 [31] 3250 3900 3300 3900 3325 4150 3950 3550 3300 4650 3150 3900 3100 4400 3000
 [46] 4600 3425 2975 3450 4150 3500 4300 3450 4050 2900 3700 3550 3800 2850 3750
 [61] 3150 4400 3600 4050 2850 3950 3350 4100 3050 4450 3600 3900 3550 4150 3700
 [76] 4250 3700 3900 3550 4000 3200 4700 3800 4200 3350 3550 3800 3500 3950 3600
 [91] 3550 4300 3400 4450 3300 4300 3700 4350 2900 4100 3725 4725 3075 4250 2925
[106] 3550 3750 3900 3175 4775 3825 4600 3200 4275 3900 4075 2900 3775 3350 3325
[121] 3150 3500 3450 3875 3050 4000 3275 4300 3050 4000 3325 3500 3500 4475 3425
[136] 3900 3175 3975 3400 4250 3400 3475 3050 3725 3000 3650 4250 3475 3450 3750
[151] 3700 4000 4500 5700 4450 5700 5400 4550 4800 5200 4400 5150 4650 5550 4650
[166] 5850 4200 5850 4150 6300 4800 5350 5700 5000 4400 5050 5000 5100 4100 5650
[181] 4600 5550 5250 4700 5050 6050 5150 5400 4950 5250 4350 5350 3950 5700 4300
[196] 4750 5550 4900 4200 5400 5100 5300 4850 5300 4400 5000 4900 5050 4300 5000
```

```
[211] 4450 5550 4200 5300 4400 5650 4700 5700 4650 5800 4700 5550 4750 5000 5100
[226] 5200 4700 5800 4600 6000 4750 5950 4625 5450 4725 5350 4750 5600 4600 5300
[241] 4875 5550 4950 5400 4750 5650 4850 5200 4925 4875 4625 5250 4850 5600 4975
[256] 5500 4725 5500 4700 5500 4575 5500 5000 5950 4650 5500 4375 5850 4875 6000
[271] 4925   NA 4850 5750 5200 5400 3500 3900 3650 3525 3725 3950 3250 3750 4150
[286] 3700 3800 3775 3700 4050 3575 4050 3300 3700 3450 4400 3600 3400 2900 3800
[301] 3300 4150 3400 3800 3700 4550 3200 4300 3350 4100 3600 3900 3850 4800 2700
[316] 4500 3950 3650 3550 3500 3675 4450 3400 4300 3250 3675 3325 3950 3600 4050
[331] 3350 3450 3250 4050 3800 3525 3950 3650 3650 4000 3400 3775 4100 3775
```

```
# 0.0022 lbs = 1 g
penguins$body_mass_lb <- 0.00220462 * penguins$body_mass_g
```

6.4.1 Exericse

Your R exercise asks you to calculate the standard deviation without using the `var()` or `sd()` command. We want to remove `NA`s manually to fix this. A helpful command is `is.na()`. Let's practice computing the variance of body_mass_lb by hand. You will want to use `nrow()`

6.5 Loading data into R

In R, you can either load data from a website or from a computer. Usually data is found in a .csv file, but sometimes it will be in different forms that R can read.

```
# From a website
fandango <- read.csv("https://raw.githubusercontent.com/kylebutts/UARK_4753/refs/heads/main/
head(fandango)
```

```
                          FILM RottenTomatoes RottenTomatoes_User Metacritic
1 Avengers: Age of Ultron (2015)            74                  86         66
2              Cinderella (2015)            85                  80         67
3                 Ant-Man (2015)            80                  90         64
4         Do You Believe? (2015)            18                  84         22
5  Hot Tub Time Machine 2 (2015)            14                  28         29
6       The Water Diviner (2015)            63                  62         50
  Metacritic_User IMDB Fandango_Stars Fandango_Ratingvalue RT_norm RT_user_norm
1             7.1  7.8           5.0                  4.5    3.70          4.3
2             7.5  7.1           5.0                  4.5    4.25          4.0
3             8.1  7.8           5.0                  4.5    4.00          4.5
```

```
4             4.7  5.4            5.0                 4.5    0.90          4.2
5             3.4  5.1            3.5                 3.0    0.70          1.4
6             6.8  7.2            4.5                 4.0    3.15          3.1
  Metacritic_norm Metacritic_user_nom IMDB_norm RT_norm_round
1            3.30                3.55      3.90           3.5
2            3.35                3.75      3.55           4.5
3            3.20                4.05      3.90           4.0
4            1.10                2.35      2.70           1.0
5            1.45                1.70      2.55           0.5
6            2.50                3.40      3.60           3.0
  RT_user_norm_round Metacritic_norm_round Metacritic_user_norm_round
1                4.5                   3.5                        3.5
2                4.0                   3.5                        4.0
3                4.5                   3.0                        4.0
4                4.0                   1.0                        2.5
5                1.5                   1.5                        1.5
6                3.0                   2.5                        3.5
  IMDB_norm_round Metacritic_user_vote_count IMDB_user_vote_count
1             4.0                      1330               271107
2             3.5                       249                65709
3             4.0                       627               103660
4             2.5                        31                 3136
5             2.5                        88                19560
6             3.5                        34                39373
  Fandango_votes Fandango_Difference
1          14846                0.5
2          12640                0.5
3          12055                0.5
4           1793                0.5
5           1021                0.5
6            397                0.5
```

However, most common is to download the data and put it in the folder where your `.Rmd` file is. To load data you will need to find the file location. By default, the working directory is whereever the `.qmd` file is located. So if your dataset is in the same folder as `.qmd`, you can load it by name:

```
penguins <- read.csv("penguins.csv")
```

Or, say you have your data in a subfolder called `data`. Then, you would use `penguins <- read.csv("data/penguins.csv")`.

If you have a dataframe that you want to export to a csv function, you will use the function `write.csv(df, "path/to/file.csv")`. This uses the same working directory as `read.csv` and relative paths to the file work the same way.

6.6 Sorting dataframes

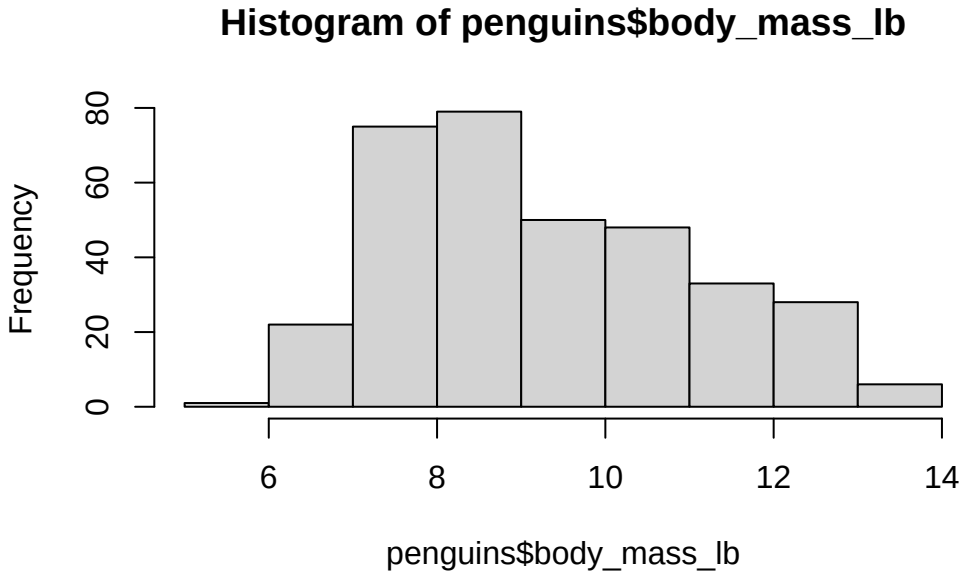
Two ways: - `sort_by` - `df[order(), ]`

7 Plotting

7.1 Histograms

Common functions for graphing are `hist()` for plotting one variable and `plot()` for plotting two variables.

```
# Count
hist(penguins$body_mass_lb)
```



You can add additional commands for better plots. Use `?hist` to see the list of options.

7.1.0.1 Exercise

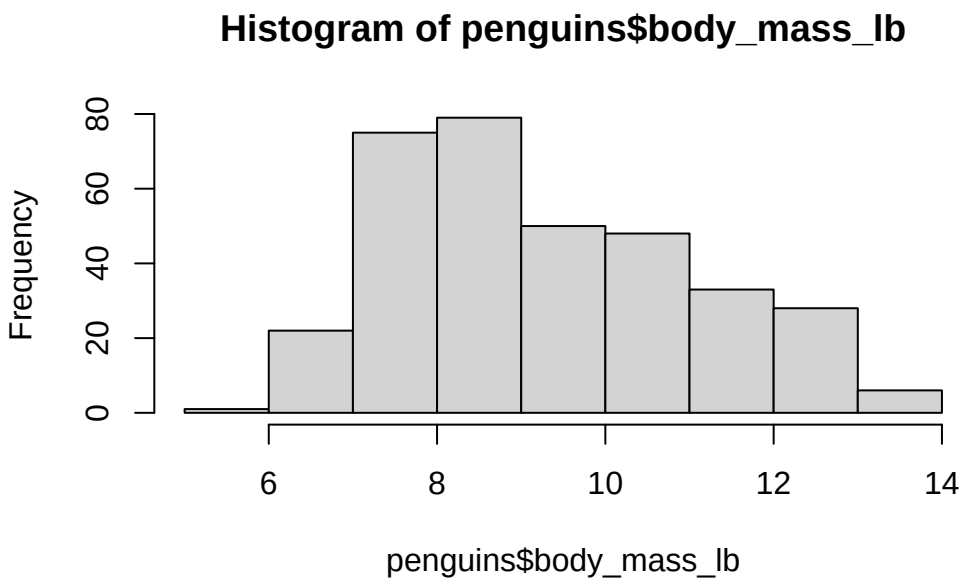
1. Type `?hist` into the console and find the following options:

- Find the option that will give you probabilities (instead of counts)
- the bar color
- the x-axis label

- the main title

Change these to make a more professional histogram:

```
## Modify this
hist(penguins$body_mass_lb)
```

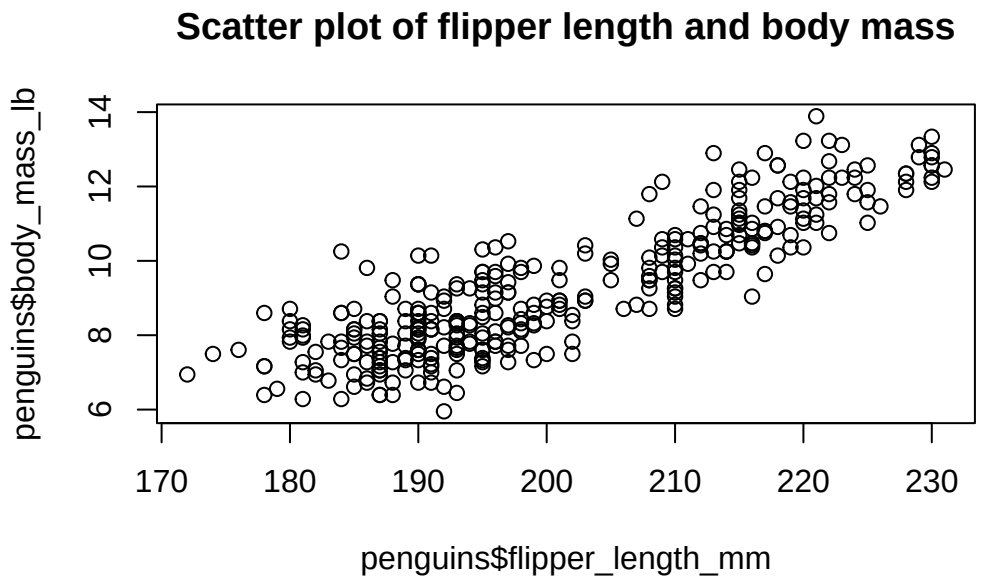


2. Compare this distributions of `bill_length_mm` for Adelie and Chinstrap penguins.

7.2 Scatter Plots

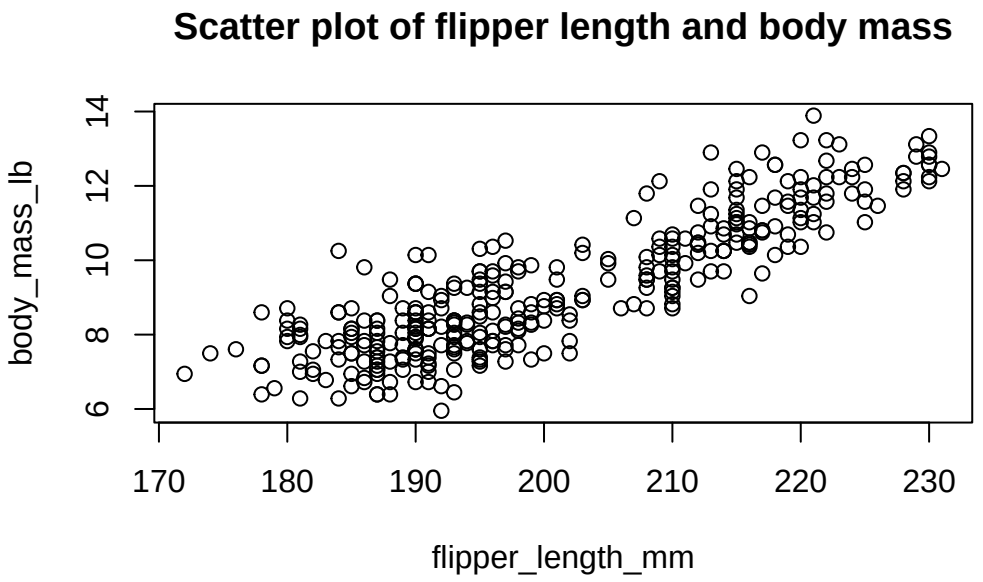
While we haven't talked about this yet, it is typically of interest to compare multiple variables together. To plot two variables, we will use the `plot()` function to make scatter plots.

```
plot(penguins$flipper_length_mm, penguins$body_mass_lb, main = "Scatter plot of flipper leng
```



It can be kind of annoying to type `penguins$` twice in plot. There is a second way to call `plot` which uses **formula** syntax. A formula has the following syntax `yvar ~ xvar` where `yvar` and `xvar` are variable names. So we can use `body_mass_lb ~ flipper_length_mm` to specify variables and then tell `plot` to use the `penguins` data.frame using the `data` argument.

```
plot(
  body_mass_lb ~ flipper_length_mm,
  data = penguins,
  main = "Scatter plot of flipper length and body mass"
)
```



If the `data` argument is not provided, it will look for `body_mass_lb` and `flipper_length_mm` as variables defined in the **global environment**. If these variables are within a data.frame and not as their own variables, then you will get the "object not found" error again.

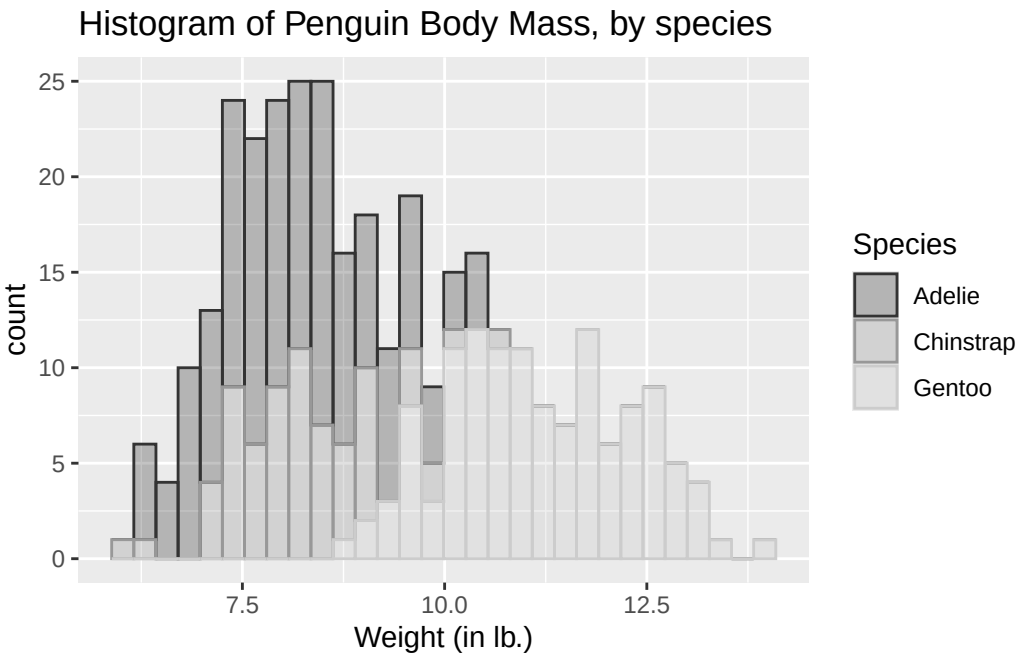
The reason I am showing you this is that this syntax will show up again when we run linear regressions in this class. And, in fact, this form of function calling where we use a `data` argument and then reference variables *within* that data.frame by name is actually quite common (see `dplyr` section in advanced materials).

7.3 (optional) ggplot2

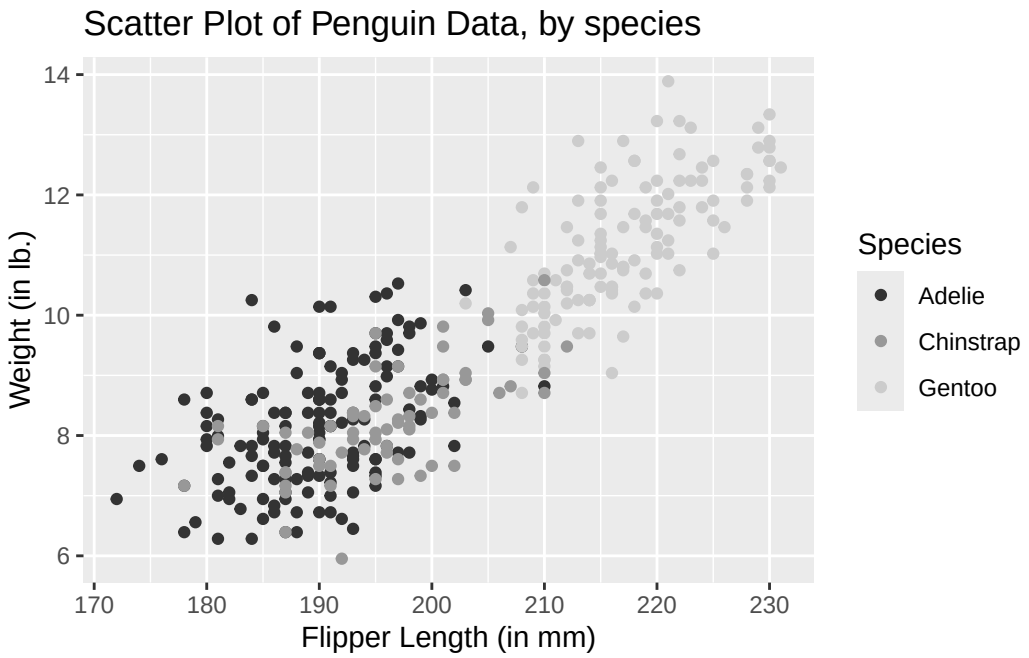
There is a package called `ggplot2` that improves base Rs graphing library. We will not cover the details here, but a curious student can find much more details here: https://ggplot2-book.org/

This is a particularly nice introduction: https://uopsych-r-bootcamp-2020.netlify.app/post/06-ggplot2/

```
library(ggplot2)
ggplot() +
  geom_histogram(data = penguins, aes(x = body_mass_lb, color = species, fill = species), al
  labs(
    title = "Histogram of Penguin Body Mass, by species",
    x = "Weight (in lb.)",
    color = "Species",
    fill = "Species"
  ) +
  scale_color_grey() +
  scale_fill_grey() +
  theme_gray()
```




```
ggplot() +
  geom_point(data = penguins, aes(x = flipper_length_mm, y = body_mass_lb, color = species))
  labs(
    title = "Scatter Plot of Penguin Data, by species",
    x = "Flipper Length (in mm)",
    y = "Weight (in lb.)",
    color = "Species",
    fill = "Species"
  ) +
  scale_color_grey() +
  theme_gray()
```



`ggplot2` makes it really easy to make beautiful and professional graphs and it would be a **really** useful skill to have in your career

```
ggplot(
  data = penguins,
  aes(
    x = bill_length_mm,
    y = bill_depth_mm,
    group = species
  )
) +
  geom_point(
    aes(color = species, shape = species),
    size = 3,
    alpha = 0.8
  ) +
  geom_smooth(method = "lm", se = FALSE, aes(color = species)) +
  theme_minimal() +
  scale_color_manual(values = c("darkorange", "purple", "cyan4")) +
  labs(
    title = "Penguin bill dimensions",
    subtitle = "Bill length and depth for Adelie, Chinstrap and Gentoo Penguins at Palmer Sta
    x = "Bill length (mm)",
    y = "Bill depth (mm)",
```

```
    color = "Penguin species",
    shape = "Penguin species"
  ) +
  theme(
    legend.position = "inside",
    legend.position.inside = c(0.85, 0.15),
    legend.background = element_rect(fill = "white", color = NA),
    plot.title.position = "plot",
    plot.caption = element_text(hjust = 0, face = "italic"),
    plot.caption.position = "plot"
  )
```

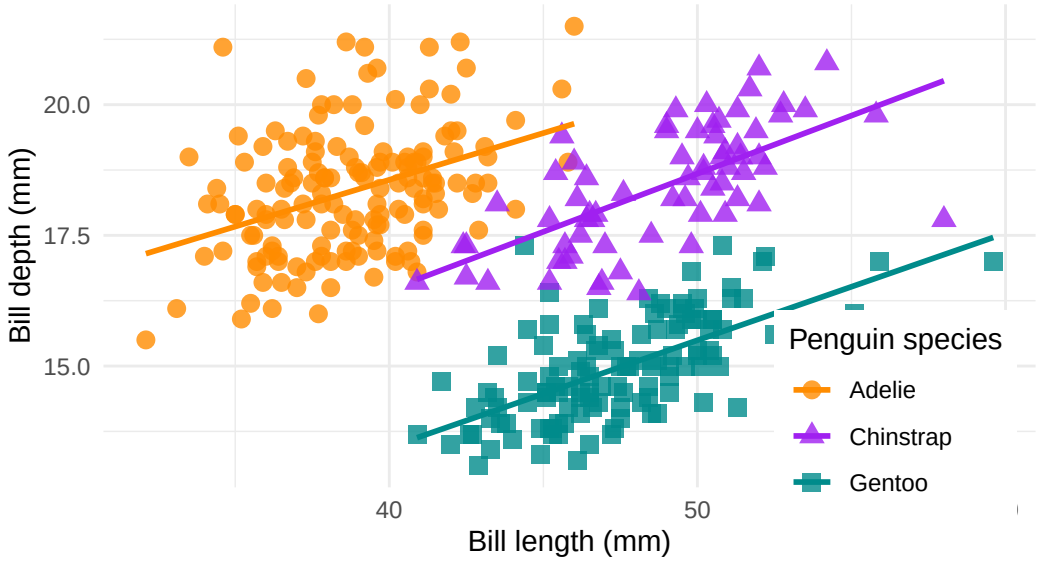
```
`geom_smooth()` using formula = 'y ~ x'
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_smooth()`).
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

Penguin bill dimensions

Bill length and depth for Adelie, Chinstrap and Gentoo Penguins at Palmer Statio



8 Running regressions in R

One of the most common and flexible ways to analyze data is using a linear regression model. While `R` comes with the `lm` function to run regressions, I recommend the `fixest` package. This uses the same syntax as `lm` but has a bunch of features that make many things simpler and nicer.

```
## You might need to install this. To do so, run this:
## install.packages("fixest")
library(fixest)
```

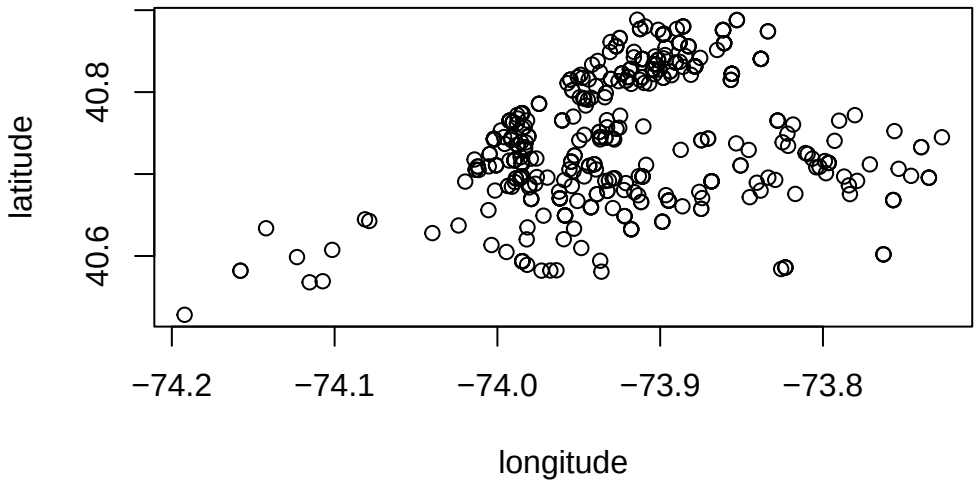
To begin, we will use a data set on the SAT scores of NYC public schools

```
nyc <- read.csv("data/nyc_sat.csv")
str(nyc)
```

```
'data.frame':   435 obs. of  22 variables:
 $ school_id                : chr  "02M260" "06M211" "01M539" "02M294" ...
 $ school_name              : chr  "Clinton School Writers and Artists" "Inwood Early College
 $ borough                  : chr  "Manhattan" "Manhattan" "Manhattan" "Manhattan" ...
 $ building_code            : chr  "M933" "M052" "M022" "M445" ...
 $ street_address           : chr  "425 West 33rd Street" "650 Academy Street" "111 Columbia
 $ city                     : chr  "Manhattan" "Manhattan" "Manhattan" "Manhattan" ...
 $ state                    : chr  "NY" "NY" "NY" "NY" ...
 $ zip_code                 : int  10001 10002 10002 10002 10002 10002 10002 10002 10002 100
 $ latitude                 : num  40.8 40.9 40.7 40.7 40.7 ...
 $ longitude                : num  -74 -73.9 -74 -74 -74 ...
 $ phone_number             : chr  "212-695-9114" "718-935-3660  " "212-677-5190" "212-475-47
 $ start_time               : chr  "" "8:30 AM" "8:15 AM" "8:00 AM" ...
 $ end_time                 : chr  "" "3:00 PM" "4:00 PM" "2:45 PM" ...
 $ student_enrollment       : int  NA 87 1735 358 383 416 255 545 329 363 ...
 $ percent_white            : chr  "" "3.4%" "28.6%" "11.7%" ...
 $ percent_black            : chr  "" "21.8%" "13.3%" "38.5%" ...
 $ percent_hispanic         : chr  "" "67.8%" "18.0%" "41.3%" ...
 $ percent_asian            : chr  "" "4.6%" "38.5%" "5.9%" ...
 $ average_score_sat_math   : int  NA NA 657 395 418 613 410 634 389 438 ...
```

```
 $ average_score_sat_reading: int  NA NA 601 411 428 453 406 641 395 413 ...
 $ average_score_sat_writing: int  NA NA 601 387 415 463 381 639 381 394 ...
 $ percent_tested           : chr  "" "" "91.0%" "78.9%" ...
```

```
## Looks like NYC :-)
plot(latitude ~ longitude, data = nyc)
```



8.0.0.1 Exercise

Whenever working with a new dataset, it is good to plot variables to get a sense of the data. In this class, a lot of data is clean already, but in the wild, you will see a lot of weirdness. For example, the Census Bureau often times codes `NA` in monthly income as `9999`. If you don't look at the data and instead just run regressions, your results will be really wonky from these "very high income earners".

To familiarize ourselves with this data, let's plot a few key variables.

1. First, create a histogram of `average_score_sat_math` and `average_score_sat_reading`
2. Print out `percent_white`; what's the problem here?
3. Create a scatter plot comparing SAT reading score with SAT math score. Please use quality x and y labels. Additionally, use a high-quality title. Try and write a descriptive title that gives your reader a "key takeaway".

8.1 First regression by hand

First, let's use some code to fix the percent variables. I simply delete the `%` from the string and then convert to a number:

```
nyc$percent_white <- as.numeric(gsub("%", "", nyc$percent_white))
nyc$percent_black <- as.numeric(gsub("%", "", nyc$percent_black))
nyc$percent_hispanic <- as.numeric(gsub("%", "", nyc$percent_hispanic))
nyc$percent_asian <- as.numeric(gsub("%", "", nyc$percent_asian))
nyc$percent_tested <- as.numeric(gsub("%", "", nyc$percent_tested))
```

Now, let's do our first regression by hand. We want to regress average SAT reading score on average SAT math score. Recall the formula for the OLS coefficients $\hat{\beta}_0$ and $\hat{\beta}_1$ are given by

$$\hat{\beta}_1 = \text{cov}(X, Y)/\text{var}(X) \text{ and } \hat{\beta}_0 = \bar{Y} - \bar{X} * \hat{\beta}_1$$

Let's grab the math and reading scores and put in variables `x` and `y` for ease of writing. But first, we need to drop NAs in either `x` or `y`. To do this, we will grab the two columns from `nyc` and then use the `na.omit()` function to drop any rows that contain an `NA` in any variable.

```
## need to drop NAs manually
vars <- nyc[, c("average_score_sat_math", "average_score_sat_reading")]
vars <- na.omit(vars)
x <- vars$average_score_sat_math
y <- vars$average_score_sat_reading

## No NAs
sum(is.na(x)) + sum(is.na(y))
```

```
[1] 0
```

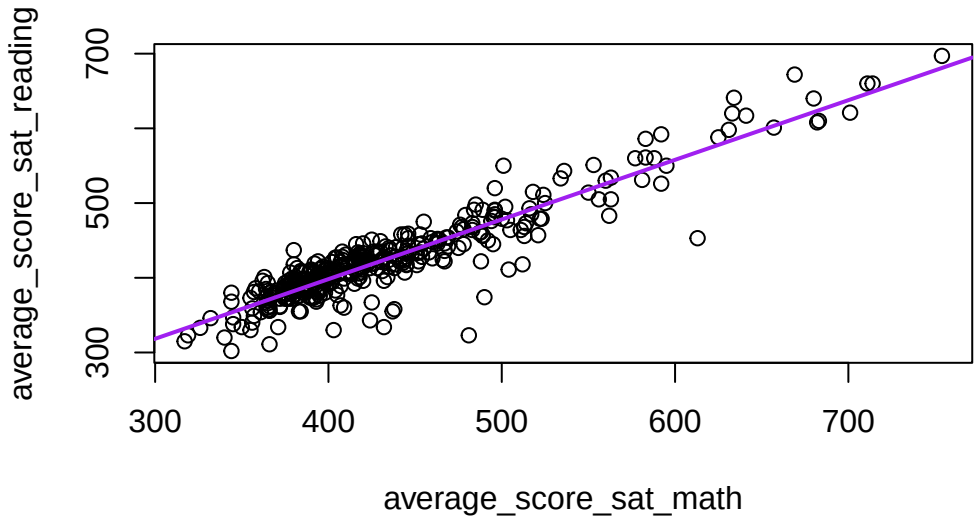
Now, that `x` and `y` contain no NAs

```
## Calcualate regression by hand
ybar <- mean(y)
xbar <- mean(x)
var_x <- var(x)
cov_x_y <- cov(x, y)

beta_1_hat <- cov_x_y / var_x
beta_0_hat <- ybar - xbar * beta_1_hat
```

Let's plot our data and our estimated regression line. We can plot the regression line using `abline()`. `abline` is similar to `points`/`lines` in that it needs to come after a call to `plot`. `abline` takes two required arguments `a` for the intercept and `b` for the slope.

```
plot(
  average_score_sat_reading ~ average_score_sat_math,
  data = nyc
)
abline(
  a = beta_0_hat, b = beta_1_hat,
  col = "purple", lwd = 2
)
```



8.1.1 Forecasting

We can use our **fitted model** to produce **fitted values**. For example, if we want to predict the average SAT reading score for a school with an average SAT math score of 450, we can do the following:

```
beta_0_hat + beta_1_hat * 450
```

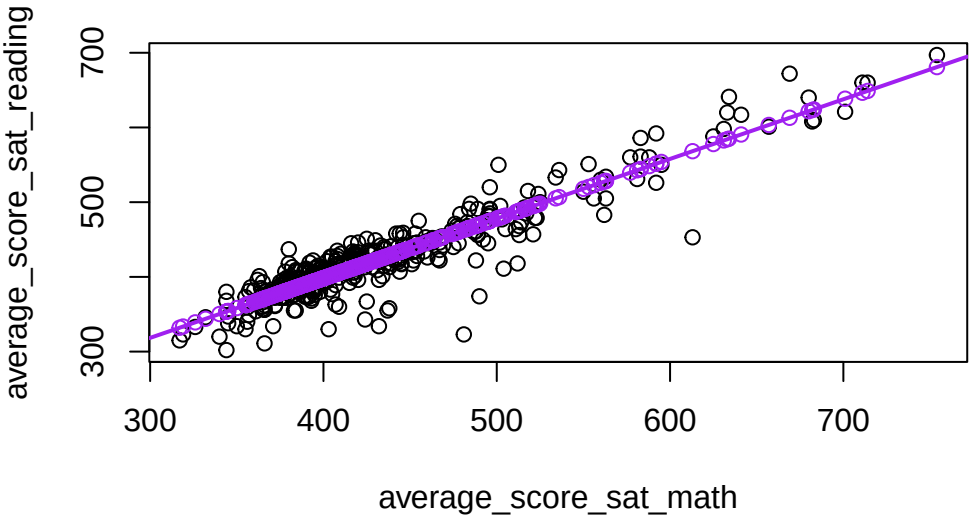
```
[1] 438.12
```

Or, we can do *in-sample* prediction by using our dataframe

```
nyc$average_score_sat_reading_hat <- (beta_0_hat + beta_1_hat * nyc$average_score_sat_math)
```

These, of course, will all fall on our line of best fit:

```
plot(
  average_score_sat_reading ~ average_score_sat_math,
  data = nyc
)
abline(
  a = beta_0_hat, b = beta_1_hat,
  col = "purple", lwd = 2
)
points(average_score_sat_reading_hat ~ average_score_sat_math, data = nyc, col = "purple")
```



Another interesting property of regression is that $(\bar{X}, \bar{Y})$ will fall on the regression line. That is, the fitted value for a school with SAT math score of $\bar{X}$ is $\bar{Y}$:

```
ybar == beta_0_hat + beta_1_hat * xbar
```

```
[1] TRUE
```

8.1.1.1 Exercise

1. Predict the SAT reading score for a school with an SAT math score of 600

8.2 First regression by function

Now, let's compare our estimates to the ones produced by R's built-in function `lm`. The `lm` function operates very similarly to `plot`: it takes two arguments: the formula `y ~ x` and the `data` argument. I will store the results of `lm` in a variable called `est_lm` so that I can use it in later functions.

```
## When lines get too long, use new lines to make reading the code easier
est_lm <- lm(
  average_score_sat_reading ~ average_score_sat_math,
  data = nyc
)
print(est_lm)
```

```

Call:
lm(formula = average_score_sat_reading ~ average_score_sat_math,
    data = nyc)

Coefficients:
           (Intercept)  average_score_sat_math
               78.8797                  0.7983
```

```
cat(paste0("\nbeta_0_hat = ", round(beta_0_hat, 4)))
```

```

beta_0_hat = 78.8797
```

```
cat(paste0("\nbeta_1_hat = ", round(beta_1_hat, 4)))
```

```

beta_1_hat = 0.7983
```

If we want more information, we pass `est_lm` to `summary`:

```
summary(est_lm)
```

```
Call:
lm(formula = average_score_sat_reading ~ average_score_sat_math,
    data = nyc)

Residuals:
     Min       1Q   Median       3Q      Max
-139.868   -9.486    2.426   13.195   71.166

Coefficients:
                       Estimate Std. Error t value Pr(>|t|)
(Intercept)            78.87972    7.26968   10.85   <2e-16 ***
average_score_sat_math  0.79831    0.01656   48.19   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 23.05 on 373 degrees of freedom
  (60 observations deleted due to missingness)
Multiple R-squared:  0.8616,    Adjusted R-squared:  0.8613
F-statistic:  2323 on 1 and 373 DF,  p-value: < 2.2e-16
```

As mentioned above, we will use the `fixest` package in this class. The function `feols` is *equivalent* to the `lm` function in base R and takes the same formula and data arguments. I will store the results of `feols` in a variable called `est` so that I can use it in later functions. To display the results, I will use the `print` function.

```
## Using fixest package
est <- feols(
  average_score_sat_reading ~ average_score_sat_math,
  data = nyc
)
```

```
NOTE: 60 observations removed because of NA values (LHS: 60, RHS: 60).
```

```
print(est)
```

```
OLS estimation, Dep. Var.: average_score_sat_reading
Observations: 375
Standard-errors: IID
                       Estimate Std. Error t value  Pr(>|t|)
(Intercept)            78.879717   7.269680 10.8505 < 2.2e-16 ***
```

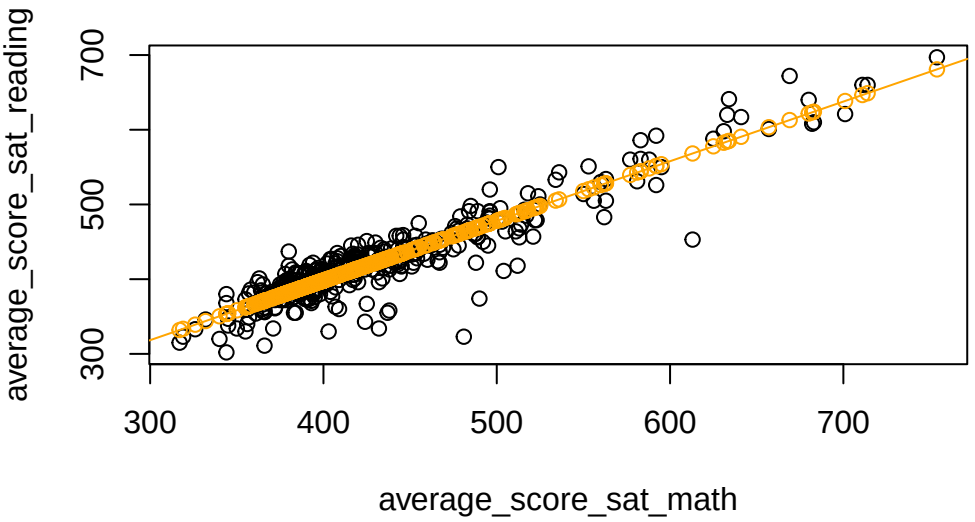
```
average_score_sat_math  0.798312   0.016565 48.1936 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
RMSE: 23.0   Adj. R2: 0.861257
```

8.2.1 Forecasting

To forecast fitted values, we will use the `predict` function. The `predict` function takes as a first argument the result of `lm` or `feols`. In addition, we will use the `newdata` argument to use. The `newdata` argument must contain the same X variable(s) that we use in our regression.

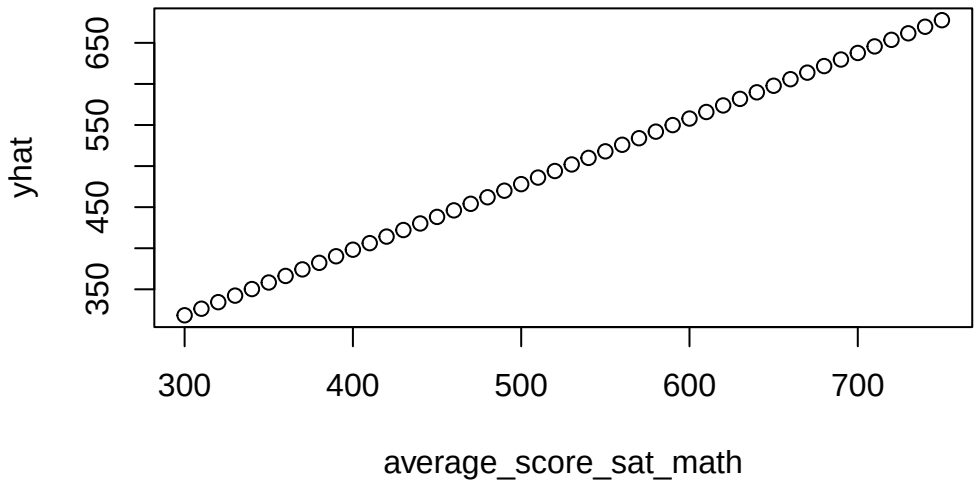
```
# Predict yhat in-sample
nyc$yhat <- predict(est, newdata = nyc)

plot(
  average_score_sat_reading ~ average_score_sat_math, data = nyc
)
abline(coef = coef(est), col = "orange")
points(
  yhat ~ average_score_sat_math, data = nyc,
  col = "orange"
)
```



We can also forecast for different values by creating a new data.frame containing hypothetical values:

```r
predict_df <- data.frame(average_score_sat_math = seq(300, 750, by = 10))
predict_df$yhat <- predict(est, newdata = predict_df)
plot(yhat ~ average_score_sat_math, data = predict_df)
```



8.2.1.1 Exercise

1. Run a regression of `average_score_sat_math` as the outcome variable and `percent_white` as the predictor variable using `feols`. Interpret the coefficients in worlds

Answer:

2. Create a plot of the raw data as well as the fitted regression line. Make sure each axis has high-quality labels.

8.2.2 Regression with indicator varaibles

```r
feols(average_score_sat_math ~ 0 + i(borough), data = nyc)
```

```
NOTE: 60 observations removed because of NA values (LHS: 60).

OLS estimation, Dep. Var.: average_score_sat_math
Observations: 375
Standard-errors: IID
                      Estimate Std. Error t value  Pr(>|t|)
```

```
borough::Bronx          404.357    6.82985 59.2044 < 2.2e-16 ***
borough::Brooklyn       416.404    6.47607 64.2989 < 2.2e-16 ***
borough::Manhattan      455.888    7.16687 63.6104 < 2.2e-16 ***
borough::Queens         462.362    8.13954 56.8045 < 2.2e-16 ***
borough::Staten Island  486.200   21.38083 22.7400 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
RMSE: 67.2   Adj. R2: 0.114643
```

```
feols(average_score_sat_math ~ i(borough), data = nyc)
```

```
NOTE: 60 observations removed because of NA values (LHS: 60).

OLS estimation, Dep. Var.: average_score_sat_math
Observations: 375
Standard-errors: IID
                      Estimate Std. Error  t value   Pr(>|t|)
(Intercept)           404.3571    6.82985 59.20435  < 2.2e-16 ***
borough::Brooklyn      12.0465    9.41203  1.27991 2.0138e-01
borough::Manhattan     51.5305    9.90005  5.20508 3.2223e-07 ***
borough::Queens        58.0052   10.62540  5.45911 8.7887e-08 ***
borough::Staten Island 81.8429   22.44519  3.64634 3.0411e-04 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
RMSE: 67.2   Adj. R2: 0.117004
```

```
feols(average_score_sat_math ~ i(borough, ref = "Manhattan"), data = nyc)
```

```
NOTE: 60 observations removed because of NA values (LHS: 60).

OLS estimation, Dep. Var.: average_score_sat_math
Observations: 375
Standard-errors: IID
                       Estimate Std. Error   t value   Pr(>|t|)
(Intercept)           455.88764    7.16687 63.610428  < 2.2e-16 ***
borough::Bronx        -51.53050    9.90005 -5.205076 3.2223e-07 ***
borough::Brooklyn     -39.48397    9.65937 -4.087634 5.3483e-05 ***
borough::Queens         6.47468   10.84510  0.597014 5.5086e-01
borough::Staten Island 30.31236   22.55003  1.344227 1.7970e-01
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
RMSE: 67.2   Adj. R2: 0.117004
```

8.2.3 Interactions

```
workers <- read.csv("data/cps.csv")
str(workers)
```

```
'data.frame':   15992 obs. of  9 variables:
 $ age      : int  45 21 38 48 18 22 48 18 48 45 ...
 $ education: int  11 14 12 6 8 11 10 11 9 12 ...
 $ black    : int  0 0 0 0 0 0 0 0 0 0 ...
 $ hispanic : int  0 0 0 0 0 0 0 0 0 0 ...
 $ married  : int  1 0 1 1 1 1 1 0 1 1 ...
 $ nodegree : int  1 0 0 1 1 1 1 1 1 0 ...
 $ re74     : num  21517 3176 23039 24994 1669 ...
 $ re75     : num  25244 5853 25131 25244 10728 ...
 $ re78     : num  25565 13496 25565 25565 9861 ...
```

```
workers$college <- 1 - workers$nodegree
```

For interactions between two discrete variables, we need a special syntax:

```
feols(re78 ~ i(black, i.college), workers)
```

```
OLS estimation, Dep. Var.: re78
Observations: 15,992
Standard-errors: IID
                    Estimate Std. Error   t value   Pr(>|t|)
(Intercept)         12817.225    146.402 87.548265  < 2.2e-16 ***
black::0:college::1  3153.177    173.126 18.213228  < 2.2e-16 ***
black::1:college::0 -2152.322    445.900 -4.826912 1.3995e-06 ***
black::1:college::1   216.947    396.580  0.547045 5.8436e-01
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
RMSE: 9,510.4   Adj. R2: 0.02795
```

For interactions between a discrete variable and a continuous variable,

```
feols(re78 ~ 0 + i(black) + i(black, age), workers)
```

```
OLS estimation, Dep. Var.: re78
Observations: 15,992
Standard-errors: IID
              Estimate Std. Error  t value  Pr(>|t|)
black::0     10610.336  247.64760 42.84449 < 2.2e-16 ***
black::1      7791.504  869.97852  8.95597 < 2.2e-16 ***
black::0:age   134.108    7.06422 18.98411 < 2.2e-16 ***
black::1:age   129.046   25.24742  5.11126 3.237e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
RMSE: 9,499.7   Adj. R2: 0.030079
```

9 Time to work with Times

In preparation for the remainder of the course, we will be thinking about working with data that is arranged in time. To do so, we are going to practice working with dates in R.

9.1 Years

The simplest time-series data to deal with is annual data. For example, take `uark_enrollment` below.

```
message("greetings!")
```

```
greetings!
```

```
warning("careful!")
```

```
Warning: careful!
```

```
error("uh oh!")
```

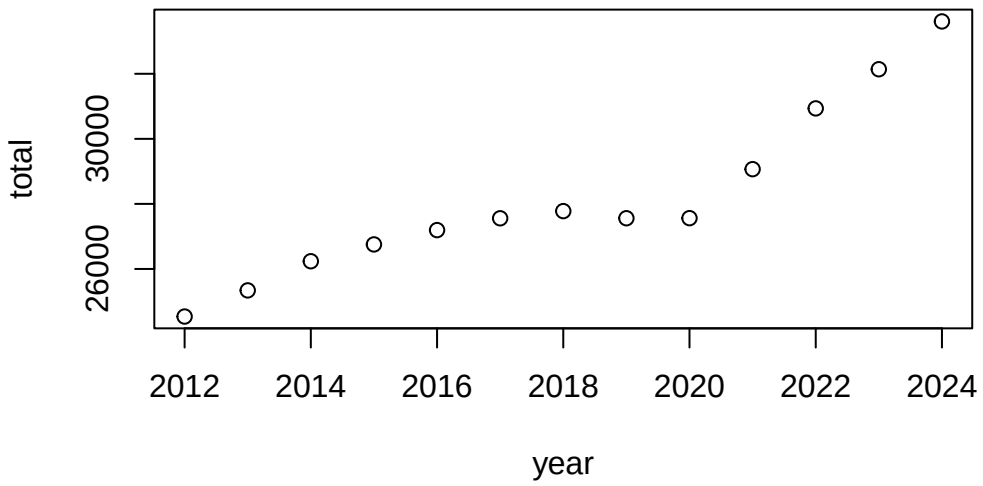
```
Error in error("uh oh!"): could not find function "error"
```

```
uark_enrollment <- data.frame(
  year = c(2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024),
  full_time = c(19508, 20379, 21047, 21415, 21668, 22144, 22602, 22193, 22070, 23282, 25214,
  part_time = c(5029, 4962, 5190, 5339, 5526, 5414, 5176, 5366, 5492, 5786, 5722, 3714, 3724
)
uark_enrollment$total <-
  uark_enrollment$full_time + uark_enrollment$part_time

# Make sure the data is sorted by year
uark_enrollment <- sort_by(uark_enrollment, uark_enrollment$year)
```

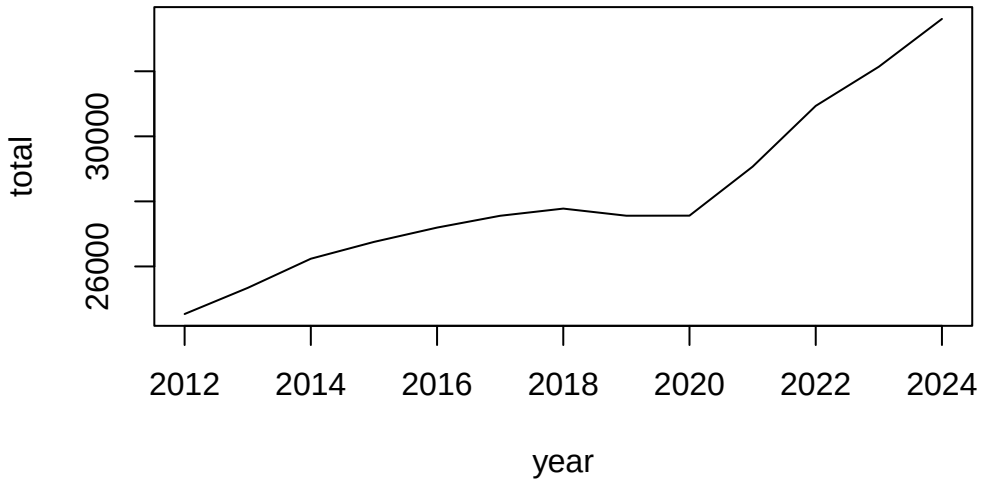
In this setting, year is just another regular *numeric* variable. Let's create a plot of enrollment over time. To do so, plot `year` on the x-axis and `total` on the y-axis.

```
plot(total ~ year, data = uark_enrollment)
```

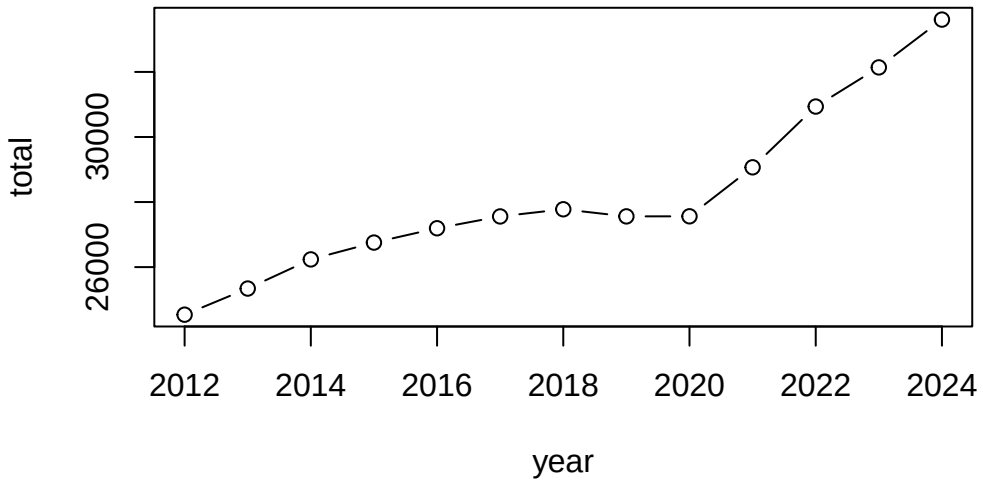


If I want to connect these points, I can add the `type = "l"` argument to plot. Or, if I want both lines and points, I can use `type = "b"` (for "both").

```
plot(total ~ year, data = uark_enrollment, type = "l")
```




```
plot(total ~ year, data = uark_enrollment, type = "b")
```



9.2 Working with dates

However, when we get to dates (day month year), this gets more difficult. Here we have box scores from Arkansas football's 2023 season, but note the days are written as strings

```
# Arkansas' 2023 football games
football <- data.frame(
  date = c(
    "11-11-2023", "11-04-2023", "09-23-2023", "09-02-2023", "10-07-2023",
    "09-16-2023", "09-09-2023", "10-21-2023", "11-24-2023", "10-14-2023",
    "11-18-2023", "09-30-2023"
  ),
  month = c(11, 11, 9, 9, 10, 9, 9, 10, 11, 10, 11, 9),
  day = c(11, 4, 23, 2, 7, 16, 9, 21, 24, 14, 18, 30),
  year = rep(2023, 12L),
  school = rep("Arkansas", 12L),
  opponent = c(
    "Auburn", "Florida", "(12) LSU", "Western Carolina", "(16) Ole Miss", "BYU",
    "Kent State", "Mississippi State", "(10) Missouri", "(11) Alabama",
    "Florida International", "Texas A&M"
  ),
  result = c("L", "W", "L", "W", "L", "L", "W", "L", "L", "L", "W", "L"),
  pts = c(10, 39, 31, 56, 20, 31, 28, 3, 14, 21, 44, 22),
  pts_opponent = c(48, 36, 34, 13, 27, 38, 6, 7, 48, 24, 20, 34)
)
```

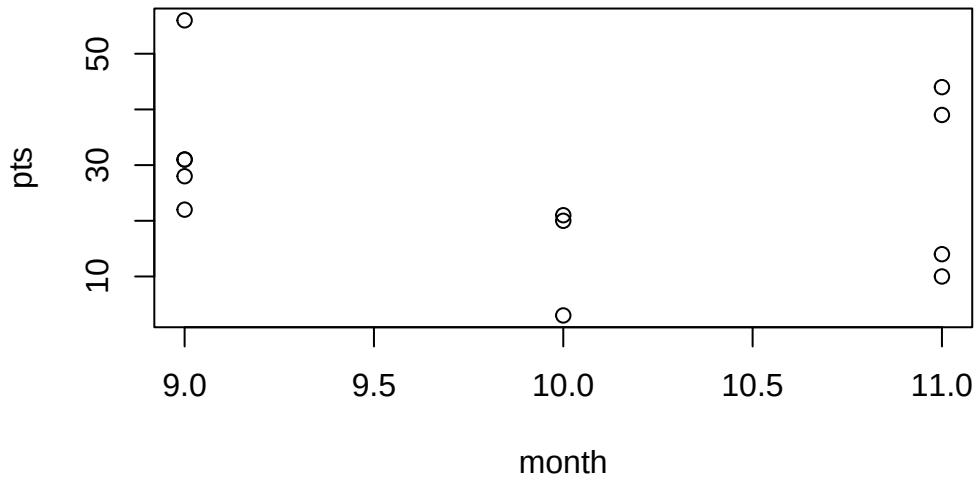
For example, say I wanted to plot the points scored by Arkansas over the season. Trying to use `date` will create a problem since it's a character

```
plot(pts ~ date, data = football)
```

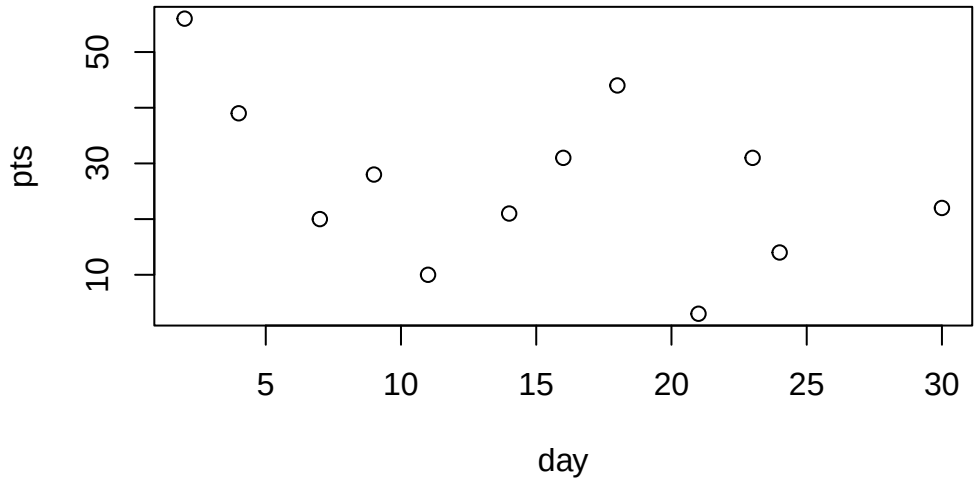
```
Error in plot.window(...): need finite 'xlim' values
```

I can try with `month` or `day`, but both are wrong. For example, if I use `day` on the x-axis, these are not in the correct order.

```
plot(pts ~ month, data = football)
```

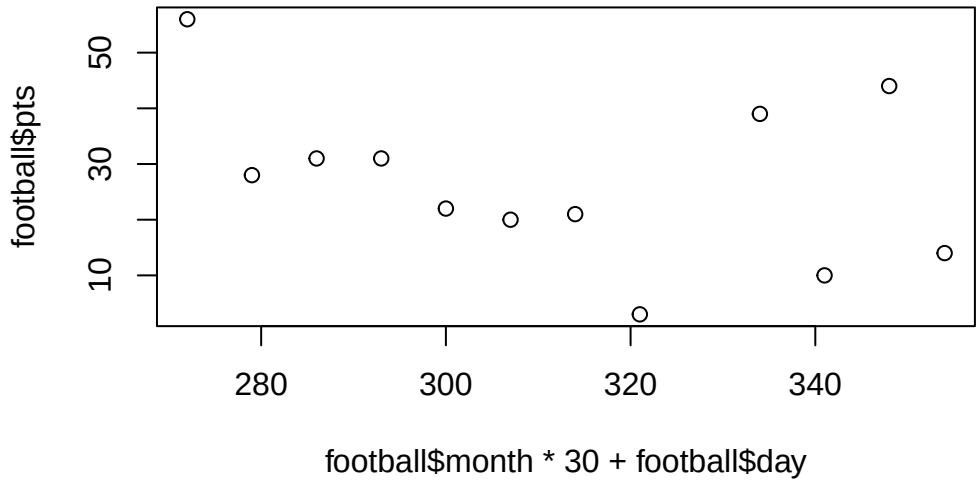



```
plot(pts ~ day, data = football)
```



The best I could think to do is to kind of fake it by doing

```
## approximately converts to days since January 1st
plot(football$month * 30 + football$day, football$pts)
```



9.3 Dates in R

It turns out R has a bunch of functionality to work with dates built in. But, I think the easiest way to work with dates is to use the `lubridate` package, so let's load that **library**.

```
## You might need to install this.
## To do so, run this:
## install.packages("lubridate")
library(lubridate)
```

```
Attaching package: 'lubridate'
```

```
The following objects are masked from 'package:base':

    date, intersect, setdiff, union
```

`lubridate` has a bunch of functions to help work with dates. First, we have `date()` which creates a Date object in R

```r
today <- today()
class("2025-08-22")
```

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

```r
class(today)
```

```
[1] "Date"
```

Note the order I am writing this: `year-month-day`. This is called the ISO Date format. ISO is the "International Organization for Standardization" and is a group that sets standards for all kinds of measurements. I LOVE this format. One reason is that if you have strings containing the dates and sort those strings, they will sort chronologically as well! `Month/day/year` does not have this feature (it would group same days on different years).

Internally, R represents dates as a number! But a very strange number:

```r
as.numeric(today)
```

```
[1] 20324
```

Because dates are represented a number, we need a day "0". If we used the first day BC as the 0, then most modern days would be really big numbers. When computers were much smaller, this could create problems, so they went with January 1st, 1970 as day 0 (or "1970-01-01").

```r
today - date("1970-01-01")
```

```
Time difference of 20324 days
```

You can add and subtract days from `Date` objects. `1` is a single *day*.

```r
tomorrow <- today + 1
two_days_ago <- today - 2
cat(paste0("Today is ", today, ". Tomorrow is ", tomorrow, "."))
```

```
Today is 2025-08-24. Tomorrow is 2025-08-25.
```

Dates and the `date` function work as vectors too:

```
last_4_classes <- date(c("2025-10-22", "2025-10-20", "2025-10-15", "2025-10-13"))
print(last_4_classes)
```

```
[1] "2025-10-22" "2025-10-20" "2025-10-15" "2025-10-13"
```

9.3.1 Back to football dataset

So returning to our previous problem, we can convert our string of dates to actual dates. But, if we try with `date`, we will get an error:

```
date(football$date)
```

This is because the date is in an ambiguous format. It does not know if "11-04-2023" is November 4th or April 11th.

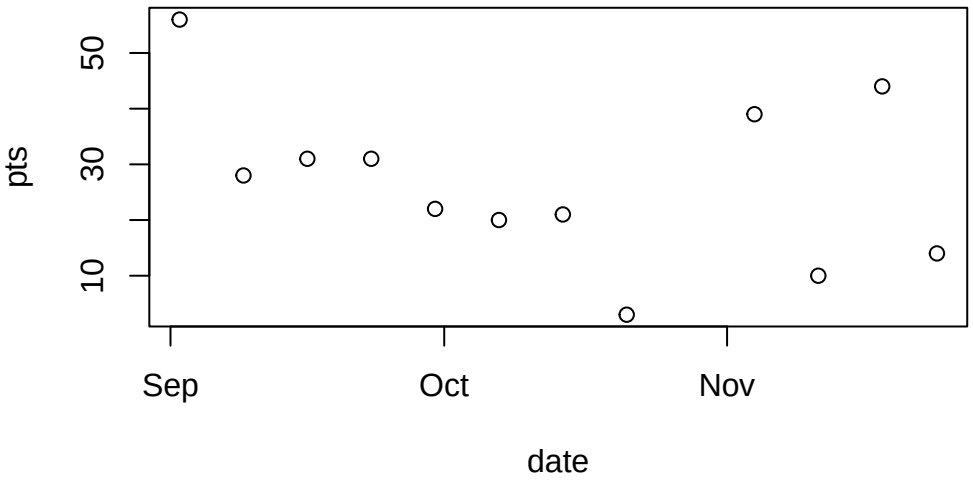
Instead, lurbidate has a set of functions `mdy`, `myd`, `dmy`, `dym`, `ymd`, `ydm` that allow you to tell R what order the year, month, and day are in. There are 6 possible combinations and 6 functions.

```
## Convert to date
football$date <- mdy(football$date)
football$date
```

```
 [1] "2023-11-11" "2023-11-04" "2023-09-23" "2023-09-02" "2023-10-07"
 [6] "2023-09-16" "2023-09-09" "2023-10-21" "2023-11-24" "2023-10-14"
[11] "2023-11-18" "2023-09-30"
```

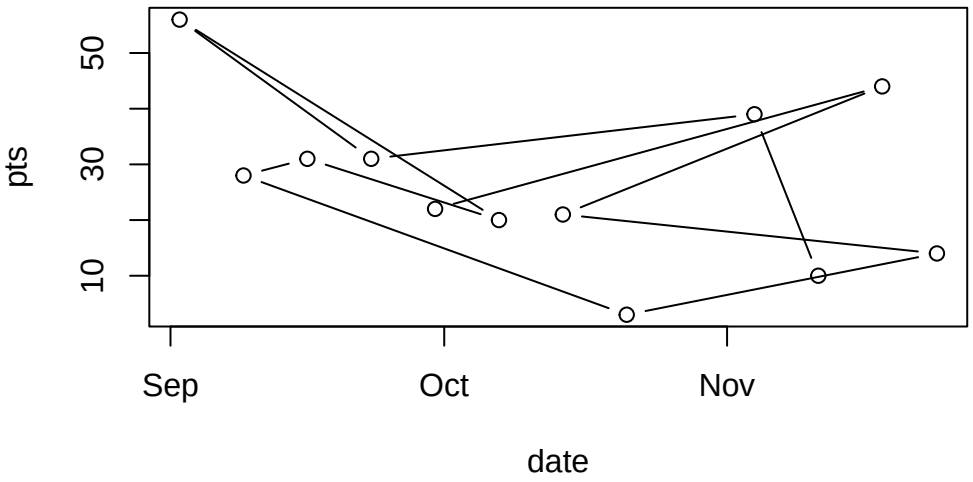
Now we can plot our scores over time. and look, R will print out pretty labels!!

```
plot(pts ~ date, data = football)
```



But, you might notice something weird if you use `type = "l"` or `type = "b"`

```
plot(pts ~ date, data = football, type = "b")
```

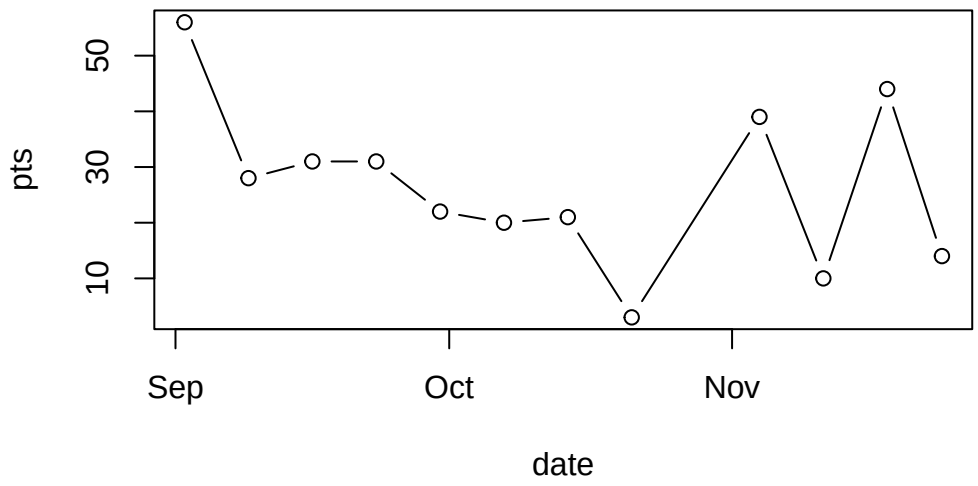


The problem occurs because our data is not sorted. When a line is being plotted, it will connect points in the order they appear in the data set. It is **very important** to sort you data when working with time-series data!

```
football <- sort_by(football, ~date)

## equivalent to
## football <- sort_by(football, football$date)
## football <- sort_by(football, football$year, football$month, football$day)
## football <- football[order(football$date), ]
```

```
plot(pts ~ date, data = football, type = "b")
```



9.3.2 More `lubridate` functions

Okay, say we have a vector of `Date`s. I can use `lubridate`'s `year()`/`month()`/`day()` functions to extract the components.

Try the month function out on `football$date`. What happens if you add the argument `label = TRUE` option to `month`?

```
year(football$date)
```

```
 [1] 2023 2023 2023 2023 2023 2023 2023 2023 2023 2023 2023 2023
```

```
month(football$date)
```

```
 [1]  9  9  9  9  9 10 10 10 11 11 11 11
```

```
month(football$date, label = TRUE)
```

```
 [1] Sep Sep Sep Sep Sep Oct Oct Oct Nov Nov Nov Nov
12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
```

```r
## day of month =
day(football$date)
```

```
 [1]  2  9 16 23 30  7 14 21  4 11 18 24
```

```r
mday(football$date)
```

```
 [1]  2  9 16 23 30  7 14 21  4 11 18 24
```

```r
## day of year = days since january 1
yday(football$date)
```

```
 [1] 245 252 259 266 273 280 287 294 308 315 322 328
```

```r
## day of the week
wday(football$date)
```

```
 [1] 7 7 7 7 7 7 7 7 7 7 7 6
```

```r
wday(football$date, label = TRUE)
```

```
 [1] Sat Sat Sat Sat Sat Sat Sat Sat Sat Sat Sat Fri
Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

```r
wday(football$date, week_start = "Monday")
```

```
 [1] 6 6 6 6 6 6 6 6 6 6 6 5
```

```r
## Quarters Q1, Q2, Q3, Q4
quarter(football$date)
```

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

```r
## Year + Quarter
quarter(football$date, type = "year.quarter")
```

```
 [1] 2023.3 2023.3 2023.3 2023.3 2023.3 2023.4 2023.4 2023.4 2023.4 2023.4
[11] 2023.4 2023.4
```

9.3.2.1 Exercise

What is the most common month in the football dataset? Hint: use the `table` function to help.

9.4 Unemployment data

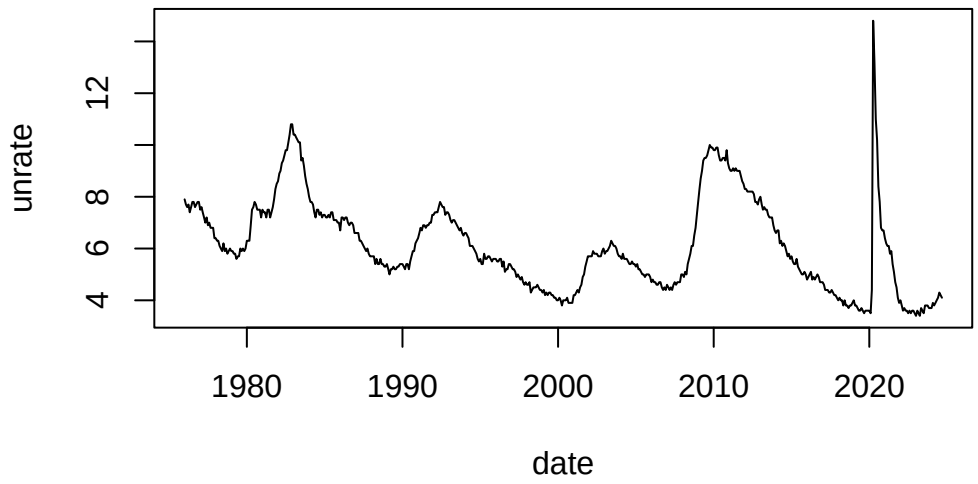
Let's introduce a new dataset on the rate of unemployment in the US.

```
unemployment <- read.csv("data/unemployment.csv")

# Convert `date` string into a `Date`:
unemployment$date <- ymd(unemployment$date)
unemployment <- sort_by(unemployment, ~date)
```

Now, let's make a time-series plot of the unemployment rate over time

```
plot(unrate ~ date, data = unemployment, type = "l")
```



9.4.1 Calculating autocorrelation of unemployment rate

To calculate the autocorrelation between y_t and y_{t-1}, we need to "shift" y back by one. Of course, the first period does not have a lag! So we will append an `NA` at the start like this: `c(NA, ...)`.

Do this to create the variable `unemployment$unrate_lag1`

```r
## Number of time-periods
T <- nrow(unemployment)

## Get y_{t-1}
unemployment$unrate_lag1 <-
  c(NA, unemployment$unrate[1:(T - 1)])

## Get y_{t-2}
unemployment$unrate_lag2 <-
  c(NA, NA, unemployment$unrate[1:(T - 2)])

## Get y_{t-3}
unemployment$unrate_lag3 <-
  c(NA, NA, NA, unemployment$unrate[1:(T - 3)])

## Grab last-years values, y_{t-12}
unemployment$unrate_lag12 <-
  c(
    rep(NA, 12),
    unemployment$unrate[1:(T - 12)]
  )
```

Then, calculate the autocovariance or autocorrelation between `unrate` and `unrate_lag1` using `cov` or `cor`

```r
cor(
  x = unemployment$unrate,
  y = unemployment$unrate_lag2,
)
```

```
[1] NA
```

Similar to before, if there are NAs present, then `NA` is returned. Instead of `na.rm = TRUE`, we need to use the argument `use = "complete.obs"`.

```r
cor(
  x = unemployment$unrate,
  y = unemployment$unrate_lag1,
  use = "complete.obs"
)
```

```
[1] 0.9616146
```

```
## Alternatively, we could grab the correct rows
cor(
  x = unemployment$unrate[1:(T - 1)],
  y = unemployment$unrate[2:T]
)
```

```
[1] 0.9616146
```

```
cor(
  x = unemployment$unrate,
  y = unemployment$unrate_lag12,
  use = "complete.obs"
)
```

```
[1] 0.6591303
```

9.4.2 Quarters

One important variable we might want is the quarter that a date falls within (Q1, Q2, Q3, and Q4). Let's try to make this using the `quarter` function from lubridate.

```
## make new variable in unemployment called `quarter`
unemployment$quarter <- quarter(unemployment$date)

## Keep yourself from accidentally using quarter as a numeric
unemployment$quarter <-
  paste("Q", unemployment$quarter)
```

9.5 Basic time-series regression

As a preview of what is to come, let's see which quarter of the year has the lowest unemployment rate:

```r
library(fixest)

## Do not do this
unemployment$q1 <- (quarter(unemployment$date) == 1)
unemployment$q2 <- (quarter(unemployment$date) == 2)
unemployment$q3 <- (quarter(unemployment$date) == 3)
unemployment$q4 <- (quarter(unemployment$date) == 4)
est_bad_version <- feols(
  unrate ~ 0 + q1 + q2 + q3 + q4,
  data = unemployment, vcov = "hc1"
)
```

```
The variable 'q4TRUE' has been removed because of collinearity (see
$collin.var).
```

```r
## Use `i`, it prints more nicely and is more simple!
est <- feols(
  unrate ~ 0 + i(quarter(date)),
  data = unemployment, vcov = "hc1"
)
etable(est)
```

```
                               est
Dependent Var.:              unrate

quarter(date) = 1 6.063*** (0.1375)
quarter(date) = 2 6.230*** (0.1614)
quarter(date) = 3 6.114*** (0.1409)
quarter(date) = 4 6.072*** (0.1411)
----------------- -----------------
S.E. type         Heteroskeda.-rob.
Observations                    585
R2                        -0.00027
Adj. R2                   -0.00544
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Just like with cross-sectional regression, a regression of an outcome on a set of indicator variables (without an intercept) produces a set of averages. If we were to add an intercept, then we would estimate difference in means between groups:

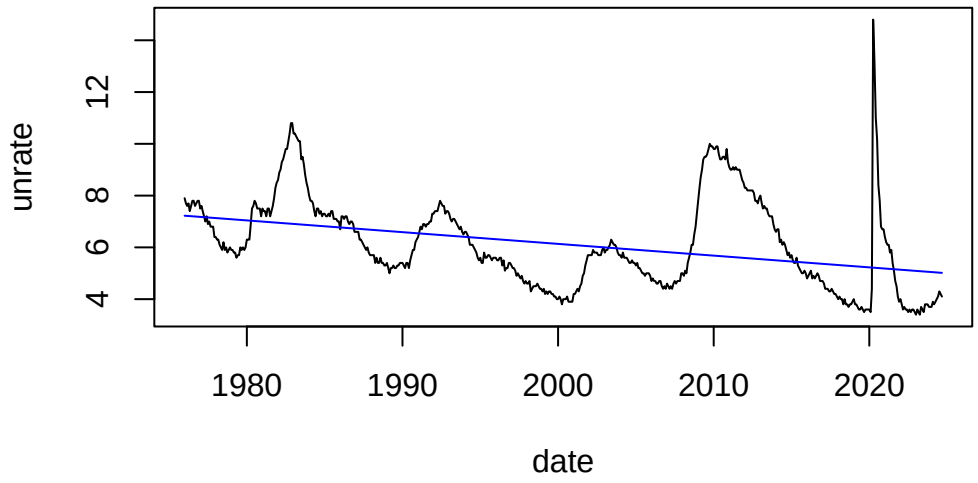
```r
est_w_intercept <- feols(
  unrate ~ 1 + i(quarter(date)),
  data = unemployment, vcov = "hc1"
)
etable(est, est_w_intercept)
```

```
                               est   est_w_intercept
Dependent Var.:             unrate            unrate

quarter(date) = 1 6.063*** (0.1375)
quarter(date) = 2 6.230*** (0.1614)   0.1673 (0.2120)
quarter(date) = 3 6.114*** (0.1409)   0.0510 (0.1968)
quarter(date) = 4 6.072*** (0.1411)   0.0096 (0.1970)
Constant                            6.063*** (0.1375)
---------------- ----------------- -----------------
S.E. type         Heteroskeda.-rob. Heteroskeda.-rob.
Observations                   585               585
R2                         -0.00027           0.00144
Adj. R2                    -0.00544          -0.00372
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Let's fit a simple time-trend to the data. We can use `predict` to get fitted values and then add them to the plot using the `lines` function.

```r
model_linear_trend <- feols(
  unrate ~ date,
  data = unemployment
)
unemployment$unrate_linear_trend <- predict(model_linear_trend)
plot(unrate ~ date, data = unemployment, type = "l")
lines(unrate_linear_trend ~ date, data = unemployment, col = "blue")
```



9.6 Exercise

1. What were the average numbers of points scored by Arkansas in the 2023 for each month? Use a regression to answer this question

Part II

Intermediate

10 Function Arguments

The way R accepts arguments to functions is a bit complicated; but once you understand the rules, it makes writing code less annoying. In particular, when a person creates a function, they pick an ordering of the arguments.

As an example, let's look at the `mean` function. The help menu shoes the usage like this:

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

This function has three kinds of arguments: 1. The `x` is a **mandatory** argument that must be passed; otherwise the function will produce an error: `argument "x" is missing, with no default` 2. The `trim` and `na.rm` arguments are **optional**. This is because they have default values `trim = 0` and `na.rm = FALSE`. With optional arguments, you can set any number of them and leave the rest to their default; For example, `mean(x, na.rm = TRUE)` works just fine and uses `trim = 0`. 3. The `...` is called `dots` and will be discussed later in the section.

When *calling* the function, we can pass arguments to the arguments in two ways: by *position* or (2) by *name*. When calling `mean(rebounds, na.rm = TRUE)` we are passing rebounds to `x` by *position* and `na.rm = TRUE` by name.

We could call `mean(rebounds, 0, TRUE)`, but if we tried `mean(rebounds, TRUE)` it would error (or in some cases produce results that you did not expect). This is because, when passing without names, the arguments get applied to the other arguments *in order*. Since `trim` comes before `na.rm`, R is assigning `TRUE` to `trim`.

Takeaway:

Just because you know in the moment the order of arguments does not mean you will remember later when you're reading your code. I recommend writing the argument names for *all* arguments, except maybe not the first. The only reason why I do not think it is necessary for the first argument is that it will *usually* be the vector, data.frame, or other key object acted on.

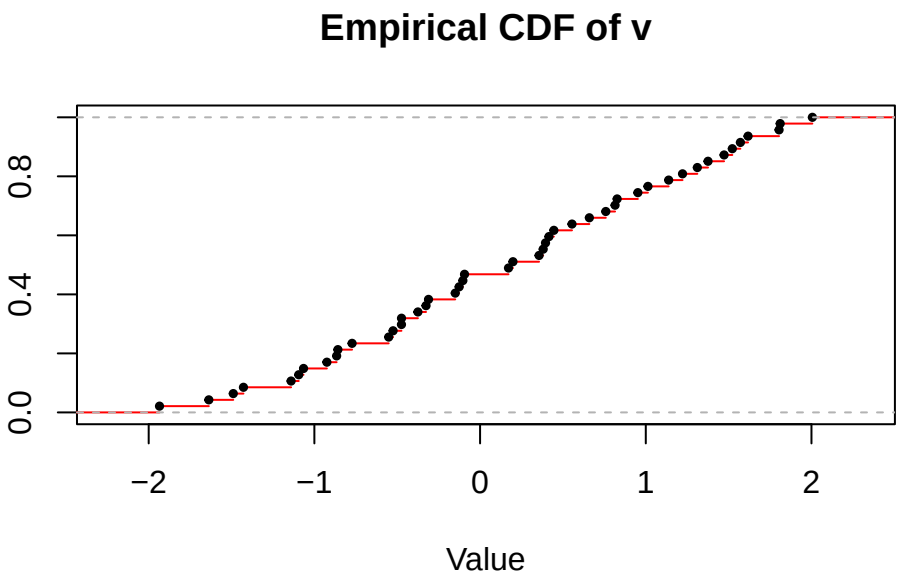
10.1 Advanced arguments notes

Given the above takeaway, this section won't be necessary if you name your arguments! The way R does argument matching is as follows: 1. First, all *named* arguments are matched. 2. What remains are the arguments that are not passed by name. These are assigned by position to the remaining arguments the function accepts. The first unnamed argument goes to the first remaining argument in the function, and so on.

The `...` is a special R syntax that `gobbles up` all other arguments passed to a function. So, for example, look at the `?plot` menu. There are only two arguments `x` and `y` that are named and then `....`

But, the following does not create any problems:

```
v <- ecdf(rnorm(47))
plot(v, main = "Empirical CDF of v", col.hor = "red", cex = 0.5, xlab = "Value", ylab = NA)
```



What happens here is that the `x` argument gets assigned the value `v`, `y` uses the default value because there are no unnamed arguments, and `...` collects the other arguments. Why not just name the arguments? Two reasons: 1. Sometimes `...` get "passed along" to another function. When this is the case, then the documentation will point to that documentation for the arguments. 2. With `S3` methods, the `...` allows better customization. This won't make sense yet; but will hopefully if you read the section below on `S3` methods.

10.1.0.1 Exercise

1. What values of arguments are being assigned here? Given these three examples, which do you think is the easiest to read?

    a. `mean(na.rm = TRUE, rebounds, 0.1)`
    b. `mean(0.1, x = rebounds, TRUE)`
    c. `mean(rebounds, na.rm = TRUE, trim = 0.1)`