# Lecture 1: Introduction

*Kelly*

*September 25, 2019*

Why on Earth would you want to use **R**? The learning curve is steep, and Excel is often easier – at first – for some routine tasks. But a little effort now will pay off big-time: learning a bit of coding in **R** will save you many, many hours if you are doing any kind of analysis. For example, generating 100 plots from 100 datasets is trivially easy in **R**, but would take days in Excel. Making a histogram is an enormous pain in Excel, but is a single command in **R**. But most of all, learning some **R** opens doors you didn't even know existed: you can write your own code, document your work, make work easily repeatable, and even present your analysis in an interactive web application (if you so desire). All from within **R**. So… here we go.

OVERARCHING NOTE: The most important thing about analysis has nothing to do with the computer. You have to know what you want to know, and why you want to know it. The computer isn't magic. It will not tell you what you want to know unless you ask it precisely. So before even touching the keyboard, sit down with a pencil, draw your data, draw your question, and figure out what you want to know.

#A Few Ground Rules

- Not magic
- **R** is awesome, but has a steep learning curve. Maintain calm.
- Put in time now to save time later
- Try and avoid rabbit holes (for the subset of you who find this addictive)

#Jumping In

### What is R?

**R** is a computer language. And as a language, it has it own spelling and grammar rules. Learning both is what will take most of your time. It is also open software, so it is free and has millions of worldwide users that are creating their own *packages* and patronizing each other on the internet.

**R** is also the interface where you create your **R** commands or programs -called scripts. This is how the **R console** looks like on a Mac.

Gives very little away. The most important thing is the sign > at the bottom of the screen, which means **R** is waiting for you to do something.

### What is RStudio

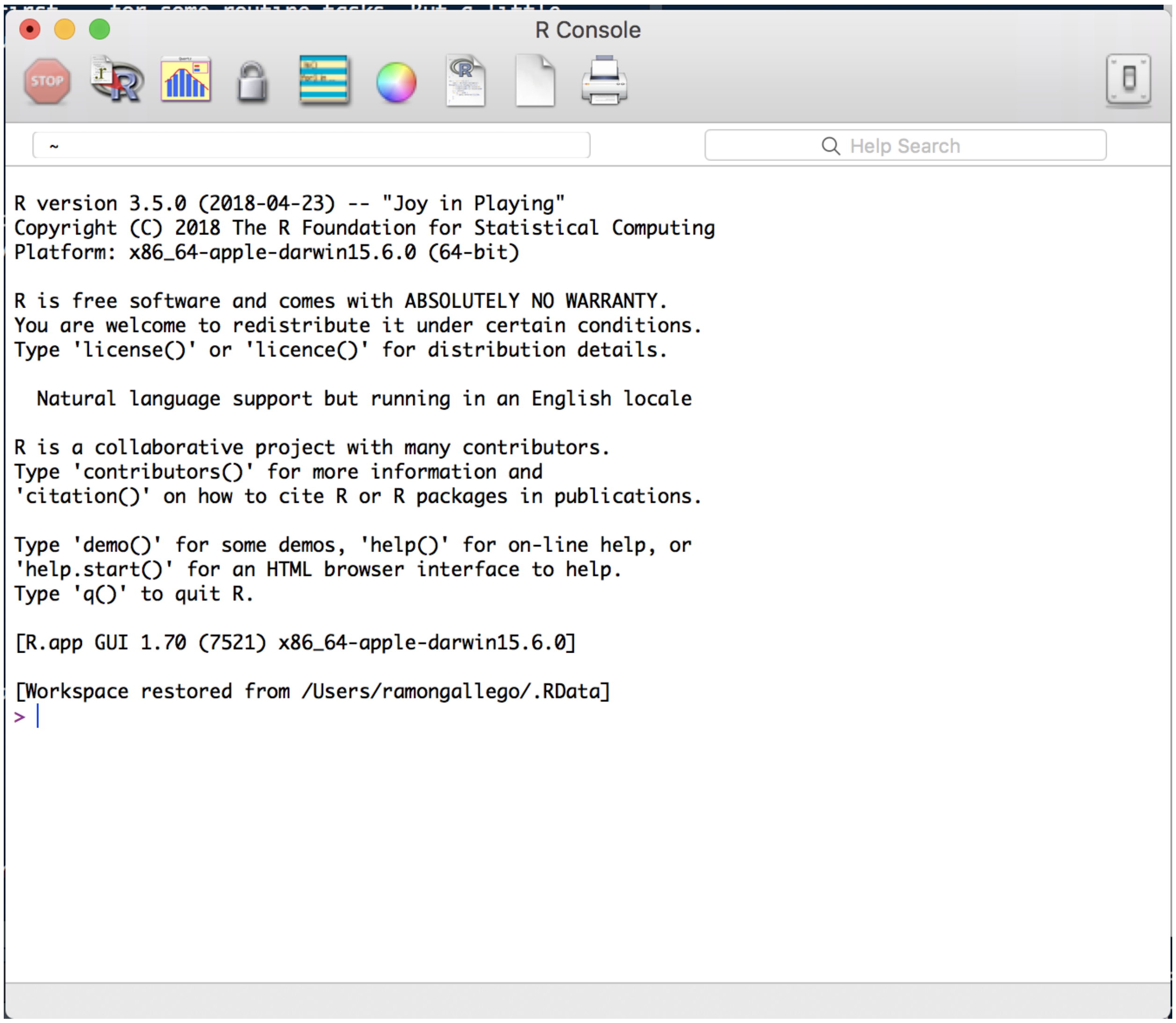
**RStudio** is a commercial software (although free for Academic / Research / leisure use) that allows you to have scripts, plots, the **R console** all at a glance. It is the friendliest way of using **R**, and also the coolest. This is how an **RStudio** Project looks like, again on a Mac.

A lot more to see here, right? The screenshot is that of me writing this lecture, in case you are into circular references.

You will be using RStudio a lot, and you will be familiar with all these components.

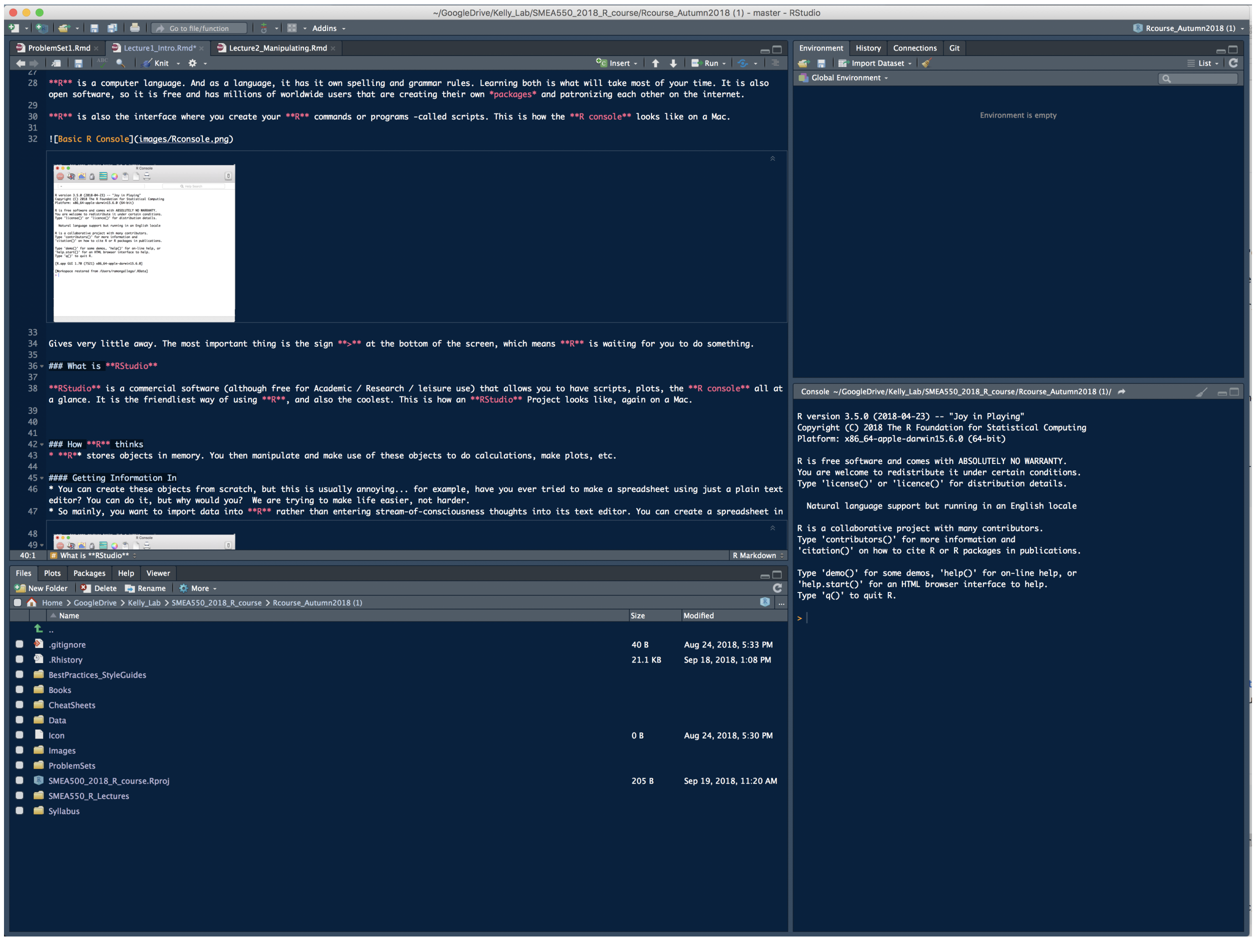
### Setting up your work environment

Open **RStudio**, and create a new Project. A Project is a way of keeping together all your R things that relate to a particular project. So click on the top right corner of your **RStudio** window and select *Create*



```
R version 3.5.0 (2018-04-23) -- "Joy in Playing"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

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

  Natural language support but running in an English locale

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

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

[R.app GUI 1.70 (7521) x86_64-apple-darwin15.6.0]

[Workspace restored from /Users/ramongallego/.RData]
> 
```

Figure 1: Basic R Console



~/GoogleDrive/Kelly_Lab/SMEA550_2018_R_course/Rcourse_Autumn2018 (1) - master - RStudio

ProblemSet1.Rmd | Lecture1_Intro.Rmd* | Lecture2_Manipulating.Rmd

```
**R** is a computer language. And as a language, it has it own spelling and grammar rules. Learning both is what will take most of your time. It is also
open software, so it is free and has millions of worldwide users that are creating their own *packages* and patronizing each other on the internet.

**R** is also the interface where you create your **R** commands or programs -called scripts. This is how the **R console** looks like on a Mac.

![Basic R Console](images/Rconsole.png)


Gives very little away. The most important thing is the sign **>** at the bottom of the screen, which means **R** is waiting for you to do something.

### What is **RStudio**

**RStudio** is a commercial software (although free for Academic / Research / leisure use) that allows you to have scripts, plots, the **R console** all at
a glance. It is the friendliest way of using **R**, and also the coolest. This is how an **RStudio** Project looks like, again on a Mac.



### How **R** thinks
* **R** stores objects in memory. You then manipulate and make use of these objects to do calculations, make plots, etc.

#### Getting Information In
* You can create these objects from scratch, but this is usually annoying... for example, have you ever tried to make a spreadsheet using just a plain text
editor? You can do it, but why would you?  We are trying to make life easier, not harder.
* So mainly, you want to import data into **R** rather than entering stream-of-consciousness thoughts into its text editor. You can create a spreadsheet in
```

40:1 | What is **RStudio** | R Markdown

Environment is empty

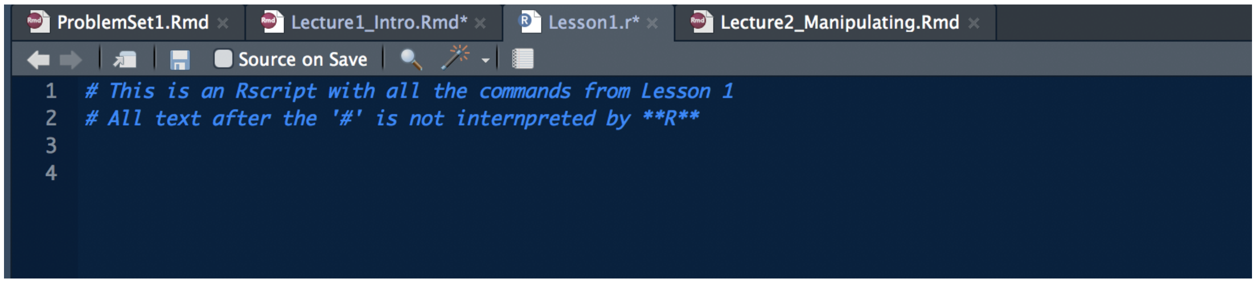
Home > GoogleDrive > Kelly_Lab > SMEA550_2018_R_course > Rcourse_Autumn2018 (1)

| Name | Size | Modified |
|---|---|---|
| .. | | |
| .gitignore | 40 B | Aug 24, 2018, 5:33 PM |
| .Rhistory | 21.1 KB | Sep 18, 2018, 1:08 PM |
| BestPractices_StyleGuides | | |
| Books | | |
| CheatSheets | | |
| Data | | |
| Icon | 0 B | Aug 24, 2018, 5:30 PM |
| Images | | |
| ProblemSets | | |
| SMEA500_2018_R_course.Rproj | 205 B | Sep 19, 2018, 11:20 AM |
| SMEA550_R_Lectures | | |
| Syllabus | | |

Console ~/GoogleDrive/Kelly_Lab/SMEA550_2018_R_course/Rcourse_Autumn2018 (1)/

```
R version 3.5.0 (2018-04-23) -- "Joy in Playing"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

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

  Natural language support but running in an English locale

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

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

>
```

Figure 2: RStudio



```
# This is an Rscript with all the commands from Lesson 1
# All text after the '#' is not internpreted by **R**
```

Figure 3: RScript

*New Project.* Create a new directory (SMEA_R_course, try to avoid spaces if you can) for this class in your hardrive.

Now you can follow up the lesson by creating a new script (so you can repeat everything later, "Lesson_1.r")

A script is just a text file with **R** commads in it. You can add your own comments by adding the '#' sign and **R** will ignore the text after it.

Now we can start

### How R thinks

- **R** stores objects in memory. You then manipulate and make use of these objects to do calculations, make plots, etc.

### Getting Information In

- You can create these objects from scratch, but this is usually annoying… for example, have you ever tried to make a spreadsheet using just a plain text editor? You can do it, but why would you? We are trying to make life easier, not harder.
- So mainly, you want to import data into **R** rather than entering stream-of-consciousness thoughts into its text editor. You can create a spreadsheet in Excel, for example, and then import it into R for analysis. As we will see.

### Categories of Information

- When coding, it helps to think in categories. Excel doesn't make you do this: you can enter anything into a cell, and you don't necessarily care if it's a number, a word, etc. But the computer cares. You can't add up words the way you can add up numbers.
- Consequently, **R** thinks about things in categories. If a value is a letter or a word, **R** calls it a *character*. If a value is a number, **R** calls it *numeric* (usually). If a value is TRUE or FALSE, **R** calls it *logical*. There are a few more of these, but you get the point: **R** wants to know the KIND of thing that is stored in memory. These are called *modes* (You don't really need to know this term, but you should know the idea).

- Relatedly, **R** wants to know what kind of structure values are stored in. Objects stored in memory can only be one of a few kinds of things. The ones you're see most often are *scalars*, *vectors*, *data frames*, *matrices*, and *lists*. (**R** calls these *classes*).

### Acting on Information

- **R** carries out operations using *functions*. You can think of functions as verbs, and objects as nouns.

- Functions do something to an object to create some kind of output. (Functions require *arguments* to tell them what to do… often, these arguments will include an object on which to act, as well as some options). The syntax is as follows: `function_name(arguments)` , which tells **R** use the function `function_name` and apply it to these arguments. (Or, if you prefer, you can think of this as `verb(noun)`)
- Side note: unlike some languages, **R** ignores spaces, such that `2+3` works just as well as `2 + 3`. New lines and tabs are also generally ignored, but there are exceptions to this rule. **R** is always case-sensitive, so `MyData` is a totally different object than `Mydata`.

###HELP!

No one remembers every function or every detail of coding in any language. Use HELP early and often. When you have a question about a particular function, type `?function` (replacing the word "function" with the name of the function you want to know about). When you have a question about how to get something done in R, **Google it**. Seriously. Someone has already done whatever you want to do, and has written about it on the web.

###Examples:

####Scalars

A scalar is just a single value, such as the number 1972. We can create a new object to store this value, which we'll call $x$, which will then be a numeric scalar with the value 1972.

```
  x <- 1972
```

If we wanted to store that as a character (i.e., word) rather than as a number, we could tell **R** that by placing 1972 in quotes. Anything you see in quotes in **R** is a character. We'll call this character scalar $y$.

```
 y <- "1972"
```

We can see what values are stored by which object (i.e., variable) names by merely typing in the names and pressing return.

```
x
```

```
## [1] 1972
```

```
y
```

```
## [1] "1972"
```

If we want to do some math, we can now do that using our object name. For example, we can add 10 to 1972 as follows:

```
x + 10
```

```
## [1] 1982
```

```
#which is the same as

1972+10
```

```
## [1] 1982
```

But note that $x$ hasn't changed… it is still 1972, not 1982. So if we want to change the value of $x$, we have to do that explicitly using the assignment symbol (<- or, less good, =)

```
x
```

```
## [1] 1972
```

```
x <- x + 10

x
```

```
## [1] 1982
```

####Vectors

Vectors are just a bunch of scalars in a row. They can be numeric, character, logical, or other.

Let's make a vector of numbers, calling it *num.vec* so it's easy to remember. And we'll do the same thing with letters, calling it *char.vec*. And we'll make a mixed vector, just to show that you can.

```
num.vec <- c(1,2,3,4,5,6)

char.vec <- c("a", "b", "c", "d")

mixed.vec <- c(1, "a", 2, 7, TRUE)
```

Note a few things here.

- First, you make vectors in **R** by using the function `c()`, which is short for "combine" or "concatenate". Separate elements of a vector by using commas.
- Second, characters still need quotes around them to define them as characters
- Third, you save objects in exactly the same way we did for scalars… this is always true in **R**; you use "<-" to save any object.

We can then carry out operations on those vectors, for example:

```
num.vec + 5  #add 5 to each element of numeric vector
```

```
## [1]  6  7  8  9 10 11
```

```
toupper(char.vec)  #use function "toupper" to change the case of character vector from lower to upper
```

```
## [1] "A" "B" "C" "D"
```

```
paste(mixed.vec, collapse="_") #use function "paste" to collapse the elements of the mixed vector into
```

```
## [1] "1_a_2_7_TRUE"
```

There's a lot to see here, but for now, the most important is how to carry out operations on objects using *functions*. **R** has many built-in functions, and the ever-expanding universe of available packages has many more.

####Data Frames

Data frames are the most common object in **R**, probably, and they are basically what you think of as a table or a spreadsheet. We can make them by sticking vectors together (useful functions here are `rbind` and `cbind` to bind together vectors as rows or as columns, respectively) or by reading in some data. Here we'll switch over to the YaRrr! The Pirate's Guide to R, by Nathaniel Phillips, because it is awesome and useful.

To do this, we will load our first *package* (a set of extensions for **R**), the first of many you will use in your deep and passionate relationship-to-be with **R**.

```
# Download/Install the yarrr package
install.packages('yarrr')

# Load the package
library(yarrr)
```

The package comes with some sample data installed; we're going to look at the data frame called `pirates`. To find out what's in the dataset, we'll look at the help file that conveniently comes with it.

```
?pirates
```

```
## No documentation for 'pirates' in specified packages and libraries:
```

```
## you could try '??pirates'
```

```
#load the data into memory, using function read.csv to read in the table
pirates <- yarrr::pirates #read.csv("../Data/pirates.csv")

# Look at the first few rows of the data
head(pirates)
```

```
##   id    sex age height weight headband college tattoos tchests parrots
## 1  1   male  28 173.11   70.5      yes   JSSFP       9       0       0
## 2  2   male  31 209.25  105.6      yes   JSSFP       9      11       0
## 3  3   male  26 169.95   77.1      yes    CCCC      10      10       1
## 4  4 female  31 144.29   58.5       no   JSSFP       2       0       2
## 5  5 female  41 157.85   58.4      yes   JSSFP       9       6       4
## 6  6   male  26 190.20   85.4      yes    CCCC       7      19       0
##   favorite.pirate sword.type eyepatch sword.time beard.length
## 1    Jack Sparrow    cutlass        1       0.58           16
## 2    Jack Sparrow    cutlass        0       1.11           21
## 3    Jack Sparrow    cutlass        1       1.44           19
## 4    Jack Sparrow   scimitar        1      36.11            2
## 5            Hook    cutlass        1       0.11            0
## 6    Jack Sparrow    cutlass        1       0.59           17
##             fav.pixar grogg
## 1      Monsters, Inc.    11
## 2              WALL-E     9
## 3          Inside Out     7
## 4          Inside Out     9
## 5          Inside Out    14
## 6 Monsters University     7
```

You can look at the names of the columns in the dataset with the `names()` function

```
# What are the names of the columns?
names(pirates)
```

```
##  [1] "id"              "sex"             "age"
##  [4] "height"          "weight"          "headband"
##  [7] "college"         "tattoos"         "tchests"
## [10] "parrots"         "favorite.pirate" "sword.type"
## [13] "eyepatch"        "sword.time"      "beard.length"
## [16] "fav.pixar"       "grogg"
```

## Descriptive statistics

Now let's calculate some basic statistics on the entire dataset. We'll calculate the mean age, maximum height, and number of pirates of each sex:

```
# What is the mean age?
mean(pirates$age)
```

```
## [1] 27.36
```

```
# What was the tallest pirate?
max(pirates$height)
```

```
## [1] 209.25
```

```
# How many pirates are there, by age?
table(pirates$age)
```

```
##
## 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
##  2  3  4  5 12  9 15 15 26 30 31 48 42 58 59 73 82 70 61 61 58 51 43 37 31
## 36 37 38 39 40 41 43 44 45 46
## 23 10 10 15  5  5  2  1  2  1
```

Now, let's calculate statistics for different groups of pirates. For example, the following code will use the `aggregate()` function to calculate the mean age of pirates, separately for each sex.

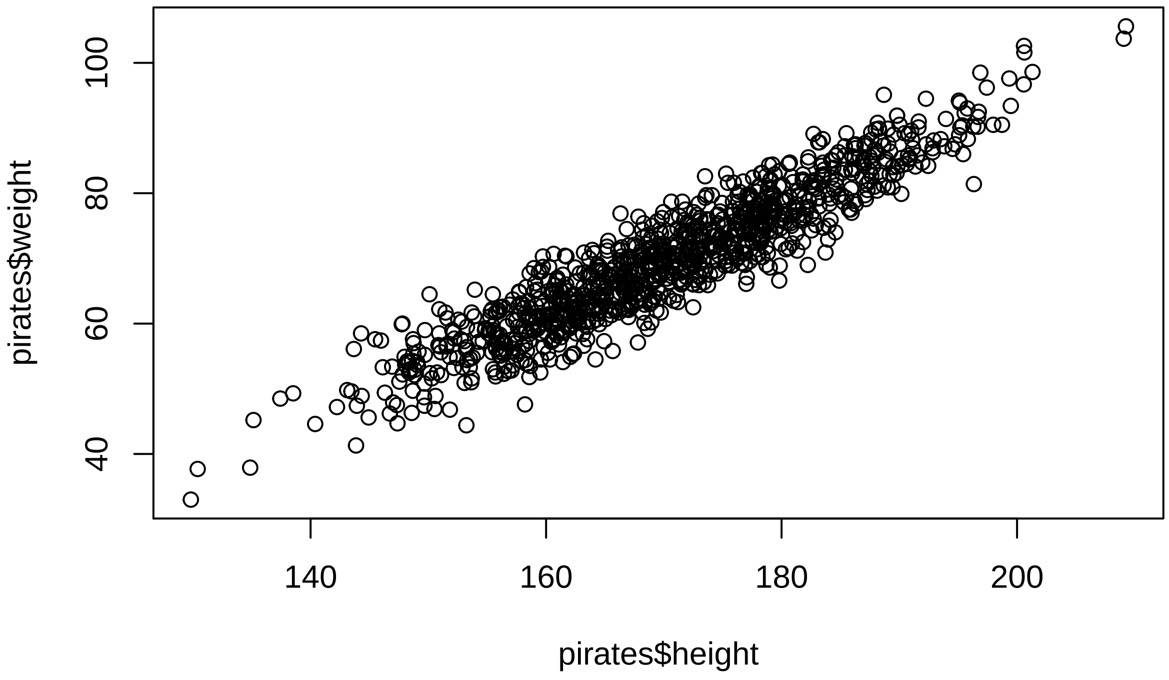
```
# Calculate the mean age, separately for each sex
aggregate(formula = age ~ sex,
          data = pirates,
          FUN = mean)
```

```
##      sex      age
## 1 female 29.92241
## 2   male 24.96735
## 3  other 27.00000
```

## Plotting

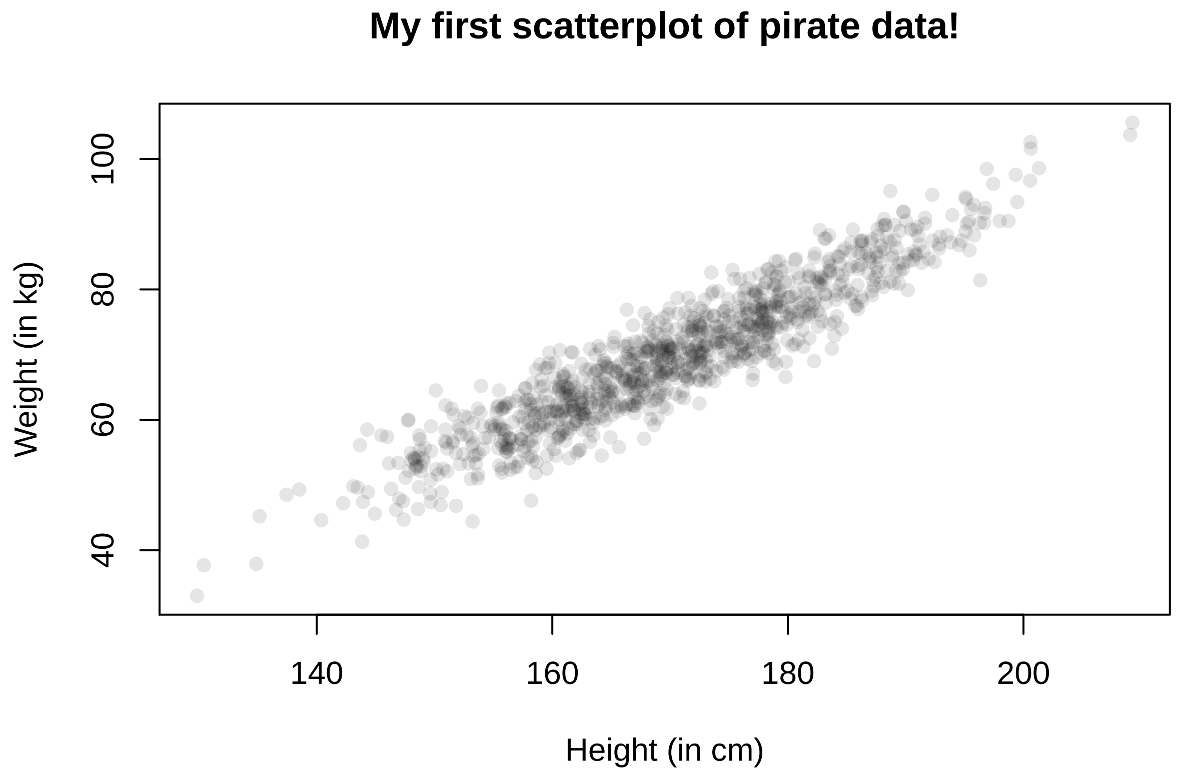
Cool stuff, now let's make a plot! We'll plot the relationship between pirate's height and weight using the `plot()` function

```
# Create scatterplot
plot(x = pirates$height,        # X coordinates
     y = pirates$weight)        # y-coordinates
```



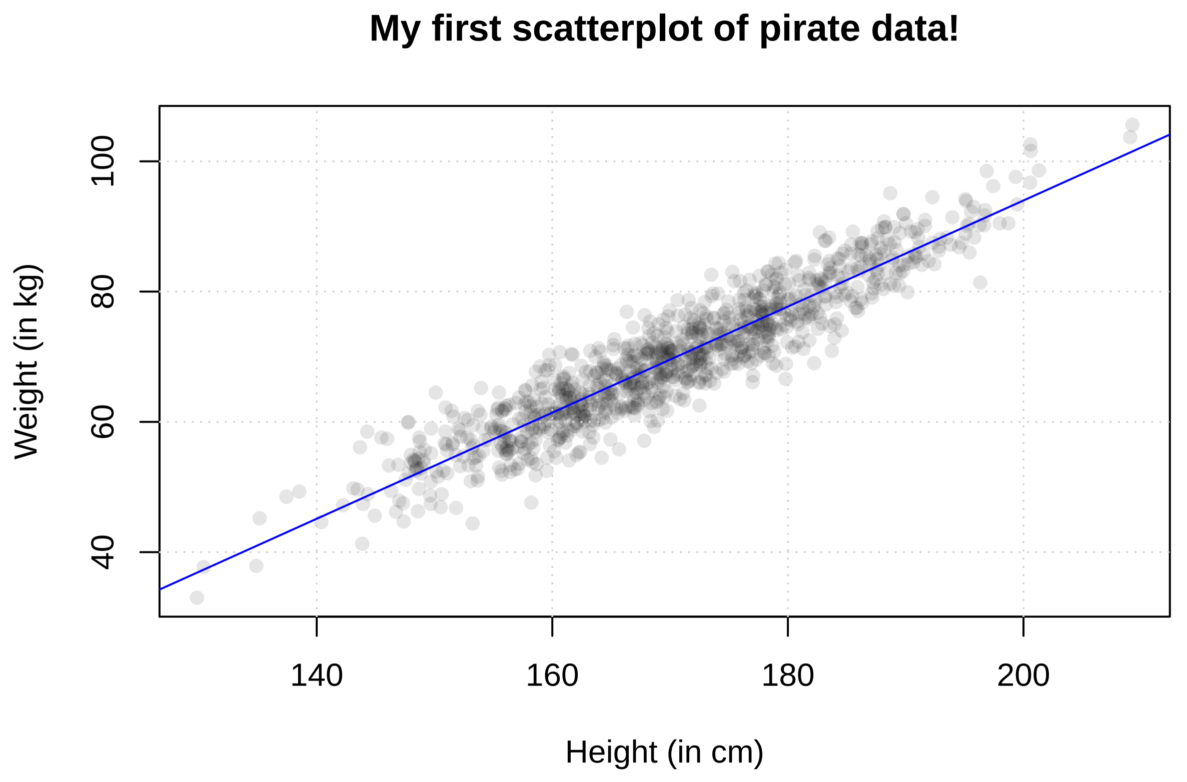
Now let's make a fancier version of the same plot by adding some customization

```r
# Create scatterplot
plot(x = pirates$height,          # X coordinates
     y = pirates$weight,          # y-coordinates
     main = 'My first scatterplot of pirate data!',
     xlab = 'Height (in cm)',     # x-axis label
     ylab = 'Weight (in kg)',     # y-axis label
     pch = 16,                    # Filled circles
     col = gray(.0, .1))          # Transparent gray
```



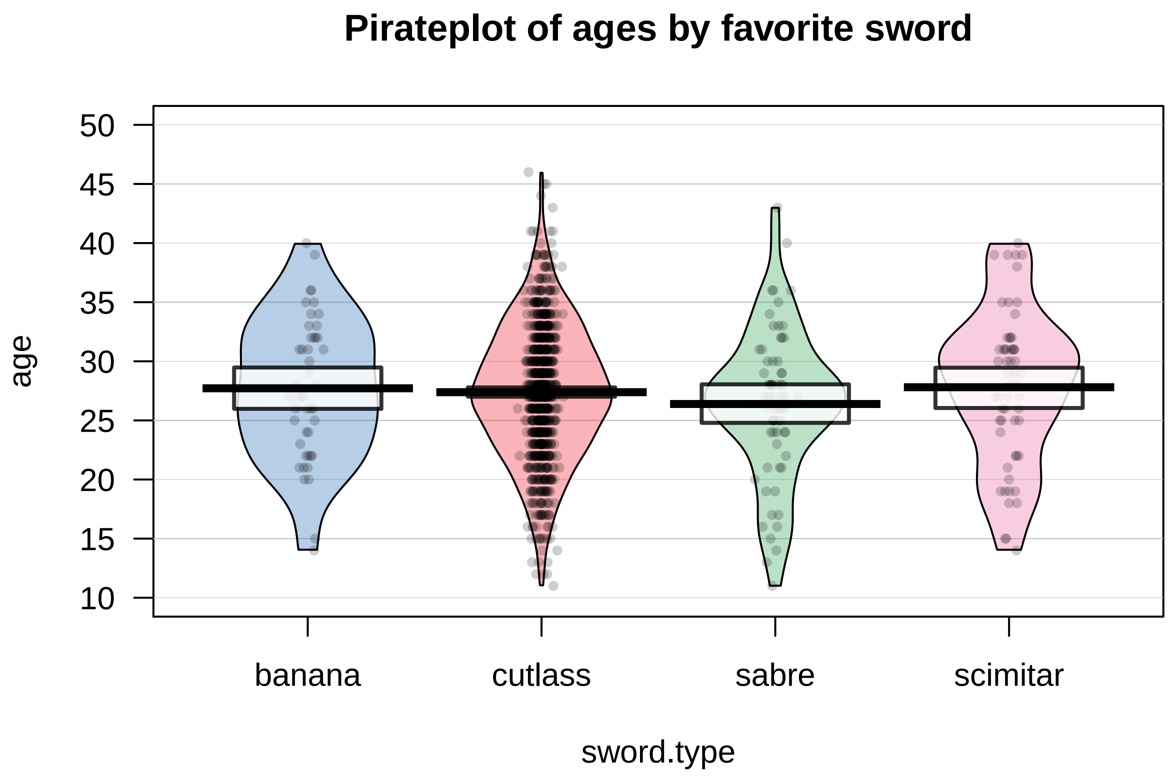
Now let's make it even better by adding gridlines and a blue regression line to measure the strength of the relationship.

```r
# Create scatterplot
plot(x = pirates$height,          # X coordinates
     y = pirates$weight,          # y-coordinates
     main = 'My first scatterplot of pirate data!',
     xlab = 'Height (in cm)',     # x-axis label
     ylab = 'Weight (in kg)',     # y-axis label
     pch = 16,                    # Filled circles
     col = gray(.0, .1))          # Transparent gray

grid()        # Add gridlines

# Create a linear regression model
model <- lm(formula = weight ~ height,
            data = pirates)

abline(model, col = 'blue')      # Add regression to plot
```



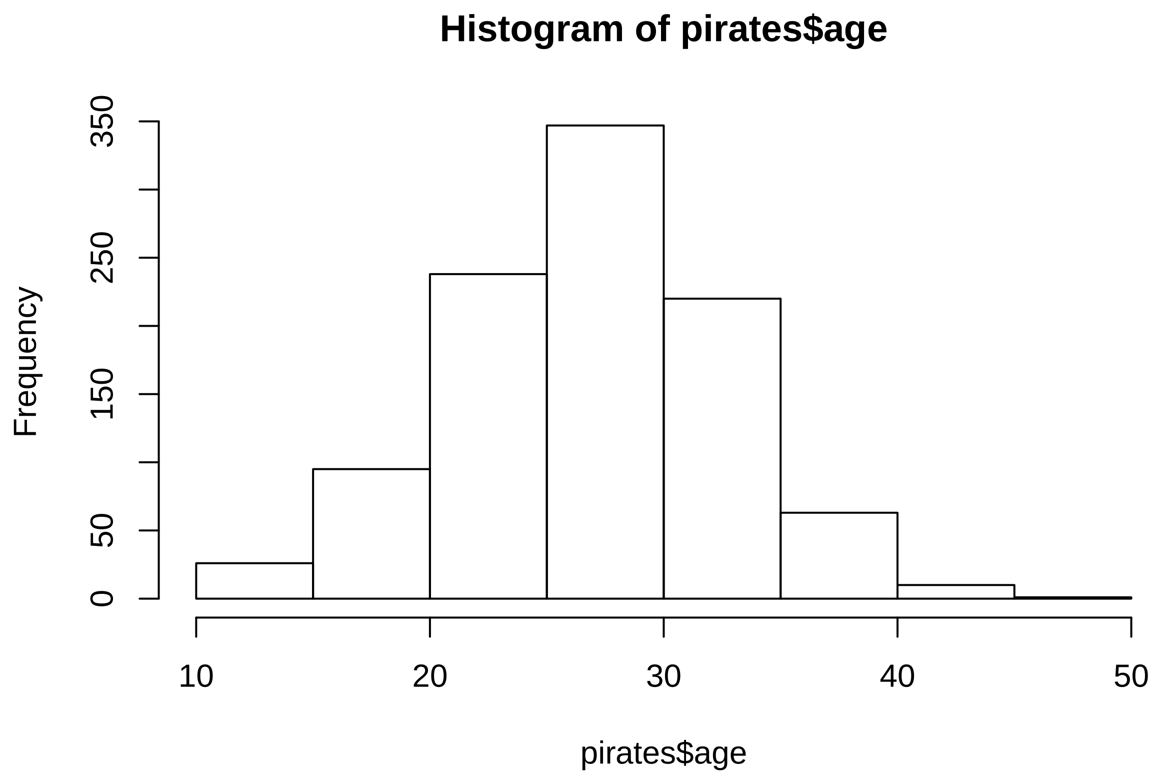
Scatterplots are great for showing the relationship between two continuous variables, but what if your independent variable is not continuous? In this case, pirateplots are a good option. Let's create a pirateplot using the `pirateplot()` function to show the distribution of pirate's age based on their favorite sword:

```
yarrr::pirateplot(formula = age ~ sword.type,
           data = pirates,
           main = "Pirateplot of ages by favorite sword")
```



Histograms are extremely useful in life. Excel sucks at doing them. By contrast, it's super easy in **R**. Let's create a histogram of pirates by age (a graphic version of what we did in the table above).

```
hist(pirates$age)
```


Histogram of pirates$age
Frequency
pirates$age