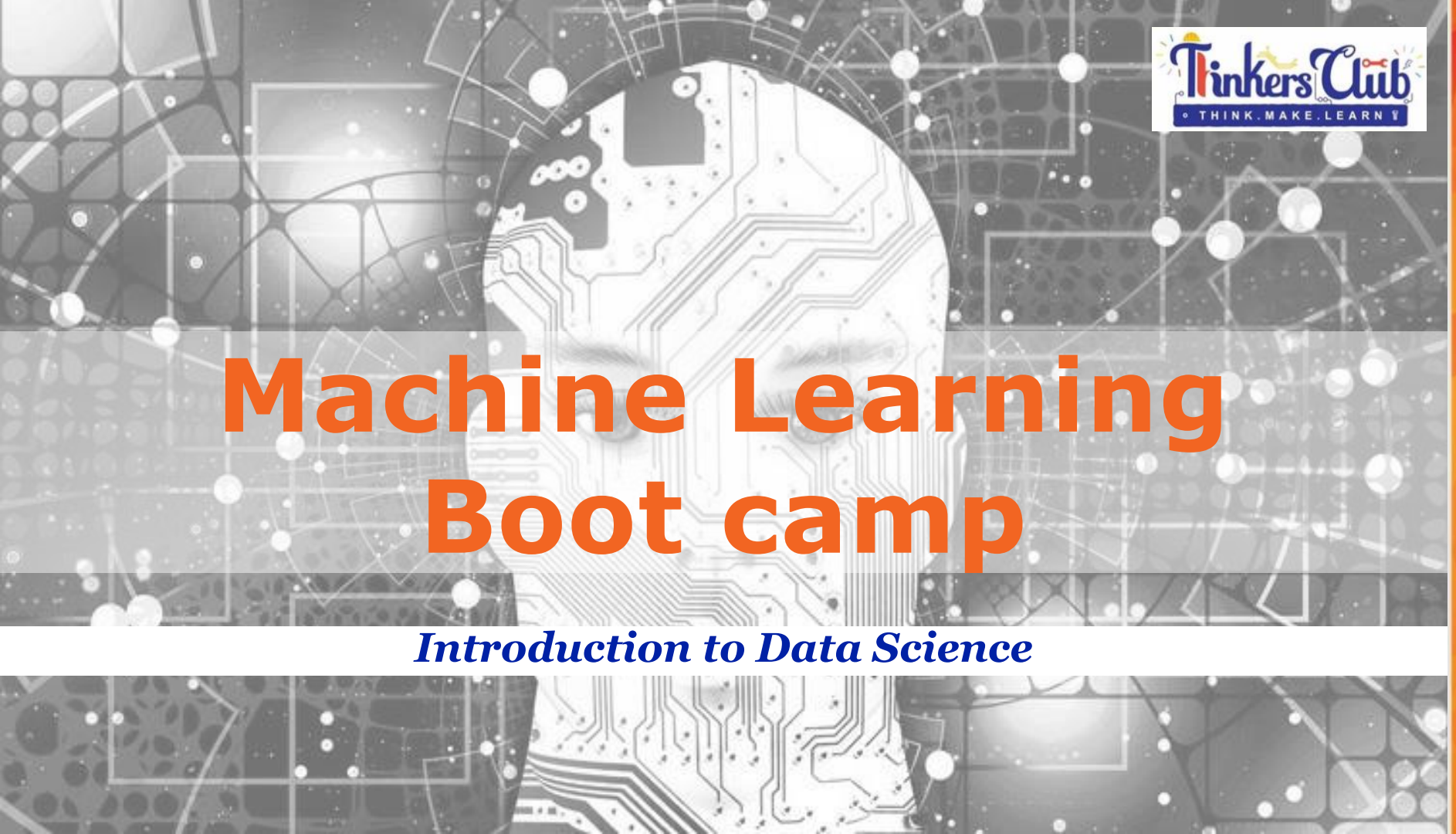


Machine Learning Boot camp

Introduction to Data Science

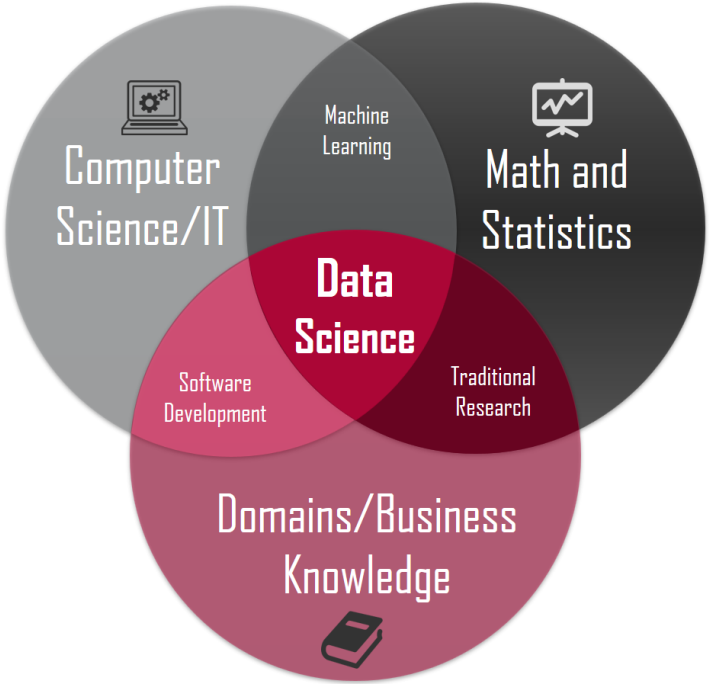
Hello and Welcome

- My name is Rajeev Agrawal.
- I am super excited to teach you all you need to know about data science and machine learning.
- *Forbes* and *Glassdoor* ranked Data Scientist as the **number one job in America**
  - due to not only the incredible salaries (**median** base salary of $120,931) for data science positions
  - but also the chance to solve some of the world's most interesting problems.

Outline

- Let's go ahead and take a quick look at the things you'll be learning in this workshop.
- You'll start off by learning the basics of R programming including working on vectors, matrices and data frames.
- Then you'll learn how to create amazing data visualizations.
- Thereafter, you will move onto machine learning.
- Exercises and projects.

What is Data Science?



- Intersection of fields
- Why this explosion:
  - We live in a data driven world with more data than ever before
  - Large computing power easily available
  - New programming tools

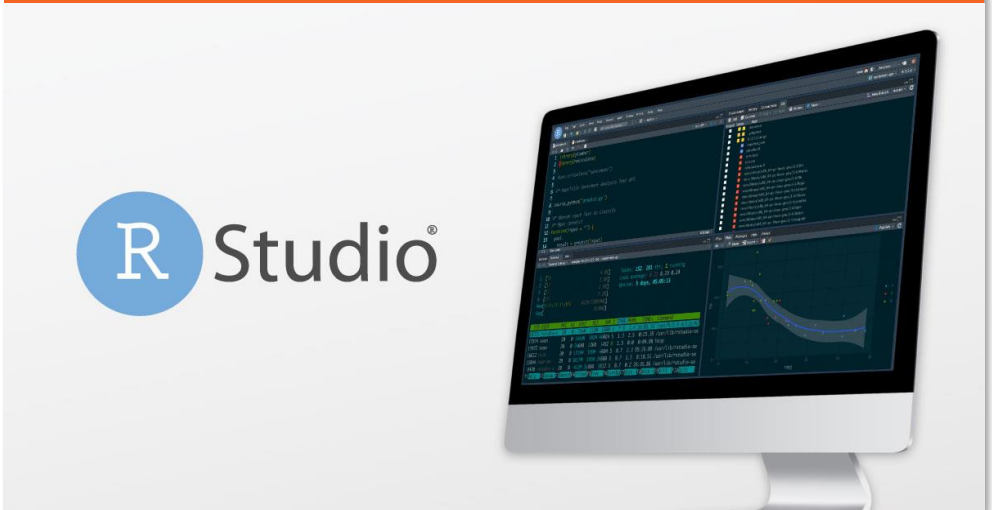
Next Step

- Download R
- Download for RStudio

Studio®

Tinkers Club
THINK.MAKE.LEARN

Development Environment Overview



Arithmetic with R

Addition

```
In [5]: 1+2
Out[5]: 3
```

Subtraction

```
In [7]: 5-3
Out[7]: 2
```

Division

```
In [8]: 1/2
Out[8]: 0.5
```

Exponents

```
In [9]: 2^3
Out[9]: 8
```

Modulo

```
In [11]: 5 %% 2
Out[11]: 1
```

Order of Operations

```
In [12]: (100 * 2) + (50 / 2)
Out[12]: 225
```

Variables

- How to assign variables in R?

```
In [1]: # Use hashtags for comments
        variable.name <- 100

In [2]: # Let's see the variable!
        variable.name

Out[2]: 100
```

- We can use variables together and work with them. For example,

```
In [3]: bank.account <- 100

In [5]: deposit <- 10

In [6]: bank.account <- bank.account + deposit

In [7]: bank.account

Out[7]: 110
```

R Data Types

- **Numerics**
  - Decimal (floating point values) are part of the numeric class in R.
  - Natural (whole) numbers are known as integers are also part of the numeric class.
- **Logical -** Boolean values (True and False) are part of the logical class. In R these are written in All Caps.
- **Characters -** Text/string values are known as characters in R. Use quotation marks (single or double) to create a text character string.

Checking Data Type Classes - The class() function is used to check the data type of a variable.

Vector Basics

- You looked at some of the basic data types in R. Now you will learn about one of the key data structures in R, the vector!
- A vector is a 1-dimensional array that can hold character, numeric, or logical data elements.
- You can create a vector by using the combine function **c()**. To use the function, we pass in the elements we want in the vector, with each individual element separated by a comma.
- **Note:** We can't mix data types of the elements in an array.

Vector Indexing & Slicing

- You can use bracket notation to index and access individual elements from a vector.
  - Keep in mind index starts at 1 (in some programming languages indexing starts at 0).
- **Multiple Indexing**
  - You can grab multiple items from a vector by passing a vector of index positions inside the square brackets.
- **Slicing**
  - You can use a colon (**:**) to indicate a slice of a vector.
  - The format is vector[start_index:stop_index] and you will get that "slice" of the vector returned to you.
- **Indexing with Names**
  - We've seen how we can assign names to the elements in a vector. We can use those names to grab individual elements from the array!

Vector Indexing & Slicing

- **Comparison Operators and Selection**
  - You can use comparison operators to filter out elements from a vector.
  - Sometimes this is referred to as boolean/logical masking, because you are creating a vector of logicals to filter out results you want.

R Basics Exercise

R Matrices

- We've learned about vectors which allow us to store indexed elements. A matrix will allow us to have a 2-dimensional data structure which contains elements consisting of the same data type.
- Before we talk about the Matrix, we should show a quick tip for quickly creating sequential numeric vectors, you can use the colon notation from slicing to create sequential vectors:

```
In [6]: 1:10

Out[6]:  1  2  3  4  5  6  7  8  9  10

In [8]: v <- 1:10
        v

Out[8]:  1  2  3  4  5  6  7  8  9  10
```

Creating a Matrix

- To create a matrix in R, you use the **matrix()** function. We can pass in a vector into the matrix.
- **Creating Matrices from Vectors**
  - We can combine vectors to later input them into a matrix.
- **Naming Matrices**
  - It would be nice to name the rows and columns for reference. We can do this similarly to the **names()** function for vectors, but in this case we define **colnames()** and **rownames()**.

Matrix Arithmetic

- We can perform element by element mathematical operations on a matrix with a scalar (single number) just like we could with vectors.
- **Comparison operators with matrices**
  - We can similarly perform comparison operations across an entire matrix to return a matrix of logicals.
- **Matrix Arithmetic with multiple matrices**
  - We can use multiple matrices with arithmetic as well.

Matrix Operations

- Now that we've learned how to create a matrix, let's learn how to use functions and perform operations on it!
- **Run the following code to create the stock.matrix**

```
# Prices
goog <- c(450,451,452,445,468)
msft <- c(230,231,232,236,228)

# Put vectors into matrix
stocks <- c(goog,msft)
stock.matrix <- matrix(stocks,byrow=TRUE,nrow=2)

# Name matrix
days <- c('Mon','Tue','Wed','Thu','Fri')
st.names <- c('GOOG','MSFT')
colnames(stock.matrix) <- days
rownames(stock.matrix) <- st.names

# Display
stock.matrix
```

Out[18]:

	Mon	Tue	Wed	Thu	Fri
GOOG	450	451	452	445	468
MSFT	230	231	232	236	228

Matrix Operations

- We can perform functions across the columns and rows, such as
  - colSums()
  - rowSums()
  - rowMeans()
- **Binding columns and rows**
  - Let's go ahead and see how we can add columns and rows to a matrix, we can use the **cbind()** to bind a new column, and **rbind()** to bind a new row.

Matrix Selection & Indexing

- Just like with vectors, we use the square bracket notation to select elements from a matrix. Since we have two dimensions to work with, we'll use a comma to separate our indexing for each dimension.
- So the syntax is then:
  - **example.matrix[*rows*,*columns*]**
  - where the index notation (e.g. 1:5) is put in place of the *rows* or *columns*.
  - If either *rows* or *columns* is left blank, then we are selecting all the rows and columns.

R Matrix Exercise

Data Frame Basics

- We've learned about vectors and their two-dimensional counterpart, matrices. Now we will learn about Data frames, one of the main tools for data analysis with R!
- Matrix inputs were limited because all the data inside of the matrix had to be of the same data type (numerics, logicals, etc). With Data frames we will be able to organize and mix data types to create a very powerful data structure tool!
- R actually has built in Data frames for quick reference to play around with!
  - Check out the following Data frames that are built-in!

```
In [2]: # Dataframe about states
        state.x77
```

```
In [8]: # US personal expense
        USPersonalExpenditure
```

  - To get a list of all available built-in dataFrames, use **data()**

Working with Data Frames

- You'll notice the states Data frame was really big, we can use the **head()** and **tail()** functions to view the first and last 6 rows respectively.
- **Overview of information**
  - We can use the **str()** to get the structure of a data frame, which gives information on the structure of the data frame and the data it contains, such as variable names and data types.
  - We can use **summary()** to get a quick statistical summary of all the columns of a data frame, depending on the data, this may or may not be useful!
- **Creating Data frames**
  - We can create data frames using the **data.frame()** function and pass vectors as arguments, which will then convert the vectors into columns of the data frame.

Data Frame Selection & Indexing

- We can use the same bracket notation we used for matrices:
  - **df[rows,columns]**
- **Selecting using column names**
  - Here is where data frames become very powerful, we can use column names to select data for the columns instead of having to remember numbers.
  - If you want all the values of a particular column you can use the dollar sign directly after the dataframe: **df.name$column.name**
  - You can also use bracket notation to return a data frame format of the same information.
- **Filtering with a subset condition**
  - We can use the **subset()** function to grab a subset of values from our data frame based off some condition.

Data Frame Selection & Indexing

- **Ordering a Data frame**
  - We can sort the order of our data frame by using the **order()** function. You pass in the column you want to sort by into the **order()** function, then you use that vector to select from the data frame.
  - We can pass a negative sign to do descending order.
  - We could also use the other column selection method we learned earlier using the $ sign.

Data Frame Operations

- Data frames are the workhorse of R, so we will basically be creating a "cheat sheet" of common operations used with data frames and R.
- We're going to do an overview of the following common operations:
  - Creating Data Frames
  - Importing and Exporting Data
  - Getting Information about Data Frame
  - Referencing Cells
  - Referencing Rows and Columns
  - Adding Rows and Columns
  - Setting Column Names
  - Selecting Multiple Rows
  - Selecting Multiple Columns
  - Dealing with Missing Data

R Data Frame Exercise

Data Input and Output with R

- **CSV Input and Output**
  - CSV stands for comma separated variable and its one of the most common ways we'll be working with data.
  - The basic format of a csv file is the first line indicating the column names and the rest of the rows/lines being data points separated by commas.
  - One of the most basic ways to read in csv files in R is to use **read.csv()** which is built-in to R.
  - When using **read.csv()** you'll need to either pass in the entire path of the file or have the file be in the same directory as your R script.

Data Input and Output with R

- **Excel Files with R**
  - R has the ability to read and write to excel, which makes it very convenient to work on the same datasets as business analysts or colleagues who only know excel, meaning they can work with excel and hand you the files, then you work with them in R!
  - To do this, we need the readxl package for R. Remember you can download it by using: **install.packages('readxl')**
  - You may need to specify **repos="http://cran.rstudio.com/"** as an argument in the packages call if you get a mirror error.
- **Writing to Excel**
  - Writing to excel requires the xlsx package.
- You may need to use **help()** to find out additional arguments you may need to get exactly what you want.

Web Scraping with R

- Data on the web is growing exponentially.
  - All of us today use Google as our first source of knowledge – be it about finding reviews about a place to understanding a new term.
- With the amount of data available over the web, it opens new horizons of possibility for a data scientist.
  - That is why, **web scraping** is an important skill for any data scientist.
- In today's world, all the data that you need is already available on the internet – the only thing limiting you from using it is the ability to access it.
  - Most of the data available over the web is not readily available. It is present in an unstructured format (HTML format) and is not downloadable. Therefore, it requires knowledge & expertise to use this data.

Web Scraping with R

- Web scraping is a technique for converting the data present in unstructured format (HTML tags) over the web to the structured format which can easily be accessed and used.
- To fully understand web scraping, you will need to know the
  - ***PIPE OPERATOR IN R (%>%)***
  - ***HTML and CSS*** in order to know what you are trying to grab off the website
- If you don't know HTML/CSS or R, you may be able to use an auto-web-scrape tool, like import.io.
  - Check it out, it will auto scrape and create a csv file for you.

Web Scraping with R

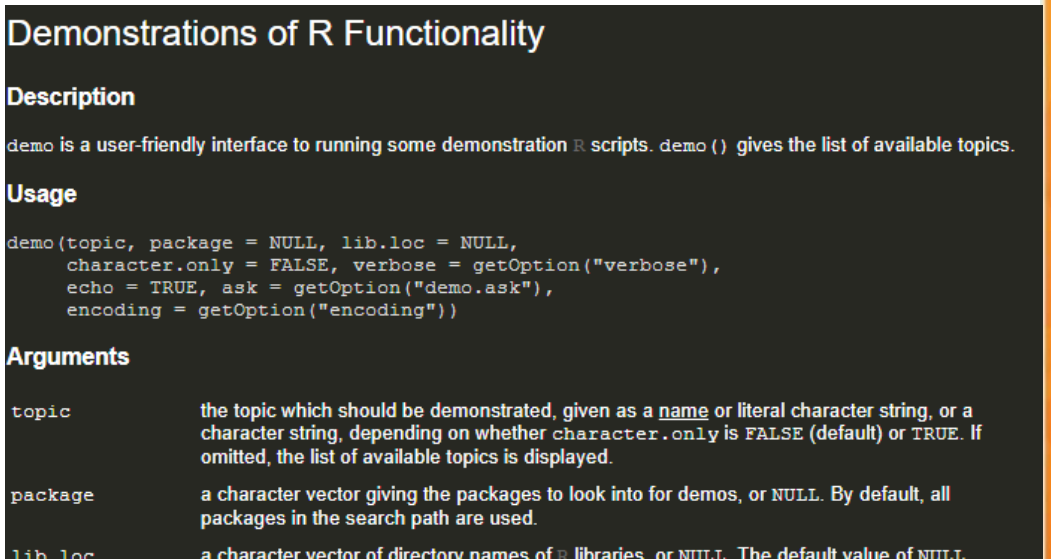
- **Prerequisites**:
  - If you are familiar with HTML and CSS a very useful library is **rvest**.

```
In [ ]: # Will also install dependencies
        install.packages('rvest')
```

  - Data Scientists generally are not very sound with technical knowledge of HTML and CSS. Therefore, we'll be using an open source software such as **Selector Gadget** to select the parts of any website and get the relevant tags to get access to that part by simply clicking on that part of the website.

Web Scraping with R

- **Demo**:
  - ??demo
  - demo(package = 'rvest')
  - demo(package = 'rvest‘, topic = ‘tripadvisor’)



Demonstrations of R Functionality

Description

`demo` is a user-friendly interface to running some demonstration `R` scripts. `demo()` gives the list of available topics.

Usage

```
demo(topic, package = NULL, lib.loc = NULL,
     character.only = FALSE, verbose = getOption("verbose"),
     echo = TRUE, ask = getOption("demo.ask"),
     encoding = getOption("encoding"))
```

Arguments

`topic`	the topic which should be demonstrated, given as a name or literal character string, or a character string, depending on whether `character.only` is `FALSE` (default) or `TRUE`. If omitted, the list of available topics is displayed.
`package`	a character vector giving the packages to look into for demos, or `NULL`. By default, all packages in the search path are used.
`lib.loc`	a character vector of directory names of `R` libraries, or `NULL`. The default value of `NULL`

Web Scraping with R

- **Web page basics**:
  - **HTML** (*Hyper Text Markup Language*) defines the content and structure of a web page. A series of elements, like paragraphs, headers, and tables, make up every HTML page.
  - The words surrounded by < > are HTML tags. Tags define where an element starts and ends. e.g. paragraph (<p>), headings (<h1>), and tables (<table>).
  - The HTML contains all the information we'd need if we wanted to read the data into R, but we'll need **rvest** to extract the table and turn it into a data frame.

  - **CSS** (Cascading Style Sheets) defines the appearance of HTML elements.
  - CSS selectors are often used to style particular subsets of elements, but you can also use them to extract elements from a web page.

File | /Users/sarakai24/GitHub/dcl-docs/wrangl...

A page heading!

This is a paragraph full of words.

Another paragraph, full of words.

A table

animal	n
alpaca	3
llama	8

view-source:/Users/sarakai24/GitHub/dcl-docs/...

```
<!doctype html>
<html>

<head>
  <title>Page Title!</title>
</head>

<body>

  <h1>A page heading!</h1>

  <p>This is a paragraph full of words.</p>

  <p>Another paragraph, full of words.</p>

  <h2>A table</h2>

  <table>
   <thead>
    <tr>
     <th style="text-align:left;"> animal </th>
     <th style="text-align:right;"> n </th>
    </tr>
   </thead>
   <tbody>
    <tr>
     <td style="text-align:left;"> alpaca </td>
     <td style="text-align:right;"> 3 </td>
    </tr>
    <tr>
     <td style="text-align:left;"> llama </td>
     <td style="text-align:right;"> 8 </td>
    </tr>
   </tbody>
  </table>

</body>
</html>

```

Web Scraping with R

- **Import** the relevant libraries

```
library(rvest)
library(tidyverse)
```

- We will use *Our World in Data* dataset on world famines available in a tabular form.
  - Using this table as an example, we'll show you how to use rvest to scrape a web page's HTML, read in a particular element, and then convert HTML to a data frame.
- **Read HTML**
  - First, copy the url of the web page and store it in a parameter.

```
url_data <- "https://ourworldindata.org/famines"
```

Web Scraping with R

- Next, use rvest::read_html() to read all of the HTML into R.
- read_html() reads in all the html for the page. The page contains far more information than we need, so next we'll extract just the famines data table.

```
url_data %>%
  read_html()
```

- **Find the CSS selector**
  - We'll find the CSS selector of the famines table and then use that selector to extract the data.
  - In Chrome, right click on a cell near the top of the table, then click Inspect (or Inspect element in Safari or Firefox).
  - Hovering over different HTML elements in the Elements pane will highlight different parts of the web page.
  - Move your mouse up the HTML document, hovering over different lines until the entire table (and only the table) is highlighted. This will often be a line with a <table> tag.
  - Right click on the line, then click *Copy* > *Copy selector* (Firefox: *Copy* > *CSS selector*; Safari: *Copy* > *Selector Path*).

Web Scraping with R

- Next, return to RStudio, create a variable for your CSS selector, and paste in the selector you copied.

```
css_selector <- "#tablepress-73"
```

- **Extract the table**
  - You already saw how to read HTML into R with rvest::read_html(). Next, use rvest::html_node() to find the first node that matches your CSS selector.

```
url_data <- "https://ourworldindata.org/famines"
css_selector <- "#tablepress-73"

url_data %>%
  read_html() %>%
  html_node(css = css_selector)
```

Web Scraping with R

- The data is still in HTML. Use rvest::html_table() to turn the output into a data frame. Note that rvest::html_table() returns a data.frame object, not a **tibble**. To convert it to a tibble, use as_tibble().

```
url_data %>%
  read_html() %>%
  html_node(css = css_selector) %>%
  html_table() %>%
  as_tibble()
```

- Now, the data is ready for wrangling in R.
- **Note** that html_table() will only work if the HTML element you've supplied is a table. If, for example, we wanted to extract a paragraph of text, we'd use html_text() instead.

Pipe Operator %>%

- The pipe operator is really useful and allows us to chain together multiple operations or functions on a data set.

R Programming Basics

Logical Operators

- Logical Operators will allow us to combine multiple comparison operators.
- The logical operators we will learn about are:
  - AND - &
  - OR - |
  - NOT - !
- **Logical Operators with Vectors:** We have two options –
  - a comparison of the entire vectors element by element,
  - or just a comparison of the first elements in the vectors, to make sure the output is a single Logical. To compare first elements use && or ||

if, else, else if Statements

- Now it is time to finally start learning how we can program some sort of logic using R! Our first step in this learning journey for programming will be simple if, else, and else if statements.
- Here is the syntax for an **if** statement in R:

```
if (condition){
    # Execute some code
}
```

- We say *if* some condition is *true* then execute the code inside of the curly brackets.
- For example, let's say we have two variables, **hot** and **temp**. Imagine that **hot** starts off as FALSE and **temp** is some number in degrees. If the **temp** is greater than 80 than we want to assign **hot==TRUE**.

if, else, else if Statements

- If we want to execute another block that occurs if the **if** statement is false, we can use an **else** statement to do this! It has the syntax:

```
if (condition) {
  # Code to execute if true
} else {
  # Code to execute if above was not true
}
```

- What if we wanted more options to print out, rather than just two, the *if* and the *else*?
  - This is where we can use the **else if** statement to add multiple condition checks, using **else** at the end to execute code if none of our conditions match up with and if or else if.

Conditional Statements Exercise

while Loops

- **while** loops are a while to have your program continuously run some block of code until a condition is met (made TRUE). The syntax is:

```
while (condition){
    # Code executed here
    # while condition is true
}
```

- A major concern when working with a while loop is to make sure that at some point the condition should become true, otherwise the while loop will go forever!
- Remember you can use Crtl-C to kill a process in R Studio!

while Loops

- You can use **break** to break out of a loop.
  - Previously we showed an **if** statement checking for 10, but this wasn't actually stopping the loop.

for Loops

- A **for** loop allows us to iterate over an object (such as a vector) and we can then perform and execute blocks of codes *for* every loop we go through. The syntax for a for loop is:

```
for (temporary_variable in object){
    # Execute some code at every loop
}
```

- **For loop over a vector**
  - We can think of looping through a vector in two different ways; the first way would be to create a temporary variable with the use of the **in** keyword.
  - The other way would be to loop a numbered amount of times and then use indexing to continually grab from the vector.
- **For loop with a matrix**
  - We can similarly loop through each individual element in a matrix.

for Loops

- **Nested for loops**
  - We can nest **for** loops inside one another.
  - However be careful when doing this, as every additional for loop nested inside another may cause a significant amount of additional time for your code to finish executing.

Functions

- Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.
- So what is a function?
  - It is a useful tool that groups together a set of statements so they can be run more than once.
  - They can also let us specify parameters that can serve as inputs to the functions.
- Functions will be one of most basic levels of reusing code in R, and it will also allow us to start thinking about program design.
- We already have seen built-in functions and we can use the **help** function to discover the *arguments* that the functions take in.

Functions

- **Default values**
  - Notice that so far we've had to define every single argument in the function when using it, but we can also have default values by using an equals sign.
- **Returning Values**
  - So far we've only been printing out results, but what if we wanted to **return** the results so that we could assign them to a variable, we can use the *return* keyword for this task.

Functions

- **Scope**
  - Scope is the term we use to describe how objects and variable get defined within R. When discussing scope with functions, as a general rule we can say that if a variable is defined only inside a function than its scope is limited to that function.

Built-in R Functions

Data Structures

- R contains quite a few useful built-in functions to work with data structures. Here are some of the key functions to know:
  - seq(): Create sequences
  - sort(): Sort a vector
  - rev(): Reverse elements in object
  - str(): Show the structure of an object
  - append(): Merge objects together (works on vectors and lists)
- **Data Types**
  - is.*(): Check the class of an R object
  - as.*(): Convert R objects

Built-in R Functions

Mathematical

- We've talked a bit about some of the built-in mathematical functions and features in R, but let's have one more look at a few of them:
  - abs(): computes the absolute value.
  - sum(): returns the sum of all the values present in the input.
  - mean(): computes the arithmetic mean.
  - round(): rounds values (additional arguments to nearest)
  - sqrt(): computes the square root.
  - exp(): computes the exponential.

R Functions Exercise

Advanced R Programming

Apply

- There are actually quite a few different apply() type functions in R. We will learn about 2 different apply() functions.
- The basic idea of an apply() is to apply a function over some iterable object.
- Let's start with **lapply():**
  - **lapply()** will apply a function over a list or vector:

```
lapply(X, FUN, ...)
```

  - where X is your list/vector and FUN is your function.

Apply

- **Anonymous Functions**
  - So you noticed that in the last example we had to write out an entire function to apply to the vector, but in reality that function is just doing something pretty simple, adding a random number. Do we really want to have to formally define an entire function for this? We don't want to, especially if we only plan to use this function a single time!
  - To address this issue, we can create an anonymous function (called this because we won't ever name it).
  - Here's the syntax for an anonymous function in R:

```
function(a){code here}
```

Apply

- **sapply() vs. lapply()**
  - Notice that lapply returned a list. We can use sapply, which simplifies the process by returning a vector or matrix.
- **sapply() limitations**
  - sapply() won't be able to automatically return a vector if your applied function doesn't return something for all elements in that vector.

Regular Expressions

- Regular expressions is a general term which covers the idea of pattern searching, typically in a string (or a vector of strings).
- For now we'll learn about two useful functions for regular expressions and pattern searching:
  - grepl(), which returns a logical indicating if the pattern was found
  - grep(), which returns a vector of index locations of matching pattern instances
- For both of these functions you'll pass in a pattern and then the object you want to search.

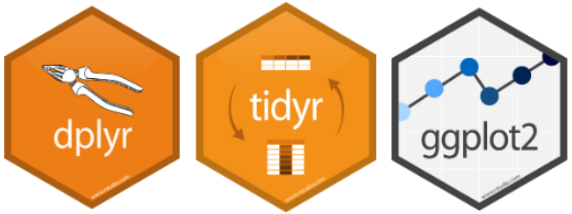
Timestamps

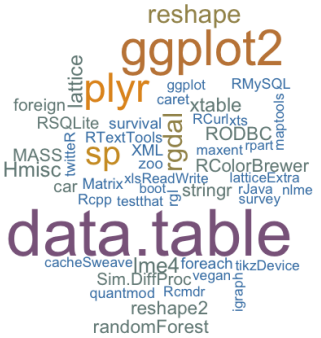
- R gives us a variety of tools for working with timestamp information.
- **Dates**
  - We can ask for today's date by asking the system using **Sys.Date()** function
  - You can convert character strings in R to a Date object using **as.Date()**. You'll need to make sure its in the correct format, or use % symbols that correlate with your given format:
- **Time**
  - Just like with Dates, we can also convert strings and work with them for time information.
  - Most times, we'll actually be using the **strptime()** function.

Code	Value
%d	Day of the month (decimal number)
%m	Month (decimal number)
%b	Month (abbreviated)
%B	Month (full name)
%y	Year (2 digit)
%Y	Year (4 digit)

R Packages

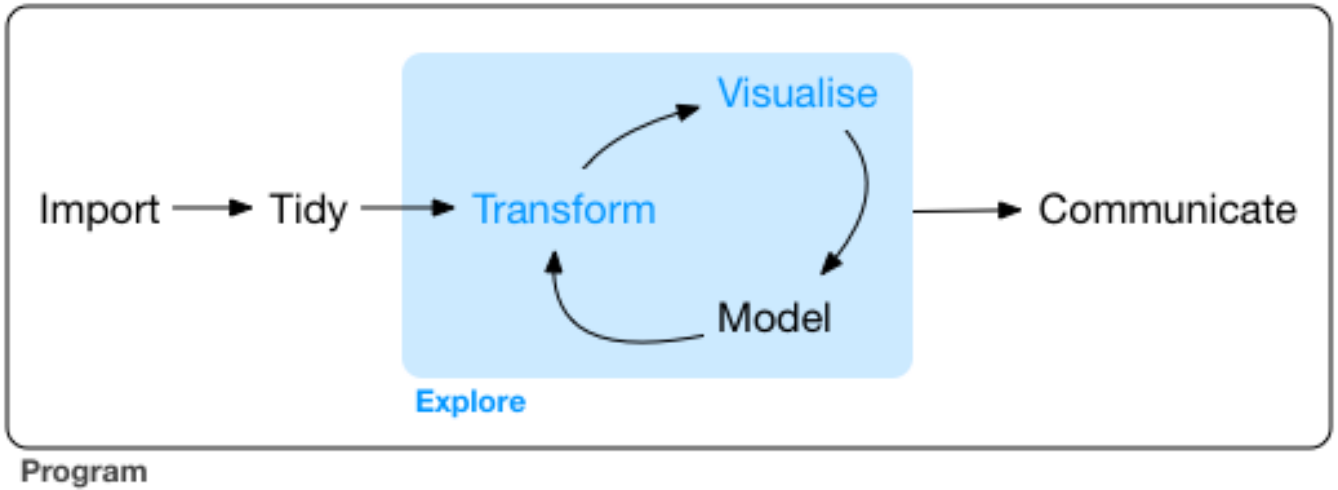
https://rstudio.com/products/rpackages/





Introduction

- The goal of data exploration is to generate many promising leads that you can later explore in more depth.



Dplyr Package

Manipulating Data

- We'll be covering the following functions:
  - **filter()** allows you to select a subset of rows in a data frame.
  - **slice()** allows us to select rows by position.
  - **arrange()** works similarly to filter() except that instead of filtering or selecting rows, it reorders them.
  - **select()** (and **rename()**): Often you work with large datasets with many columns but only a few are actually of interest to you. select() allows you to rapidly zoom in on a useful subset using operations that usually only work on numeric variable positions.
  - **distinct()** returns the unique values in a table.

Dplyr Package

Manipulating Data

- Some more functions:
  - **mutate()** and **transmute()**: Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. This is the job of mutate(). Use transmute if you only want the new columns.
  - **summarise()** quickly collapses data frames into single rows using functions that aggregate results. Remember to use na.rm=TRUE to remove NA values.
  - **sample_n()** and **sample_frac()** can be used to take a random sample of rows: use sample_n() for a fixed number and sample_frac() for a fixed fraction.

Dplyr Exercise

Tidyr Package

Cleaning Data

- Hopefully you've seen how **dplyr()** can save you lots of time and headaches! Remember to use **help()** if you ever need help using the package.
- Now we can begin to learn about **tidyr** which is a complementary package that will help us create tidy data sets!
- So what do we mean when we say "tidy data"?
    - Tidy data is when we have a data set where every row is an observation and every column is a variable, this way the data is organized in such a way where every cell is a value for a specific variable of a specific observation. Having your data in this format will help build an understanding of your data and allow you to analyze or visualize it quickly and efficiently.

Tidyr Package

Cleaning Data

- **Installing tidyr**

```
In [ ]: install.packages('tidyr',repos = 'http://cran.us.r-project.org')
```

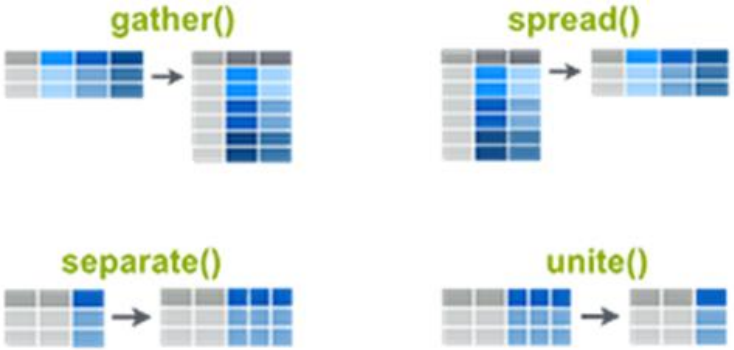
```
In [5]: library(tidyr)
        library(data.table)
```

- **Data.frames vs. data.tables**
  - Loosely speaking, you can think of data.tables as data.frames with extra features.
  - data.frame is part of base R.
  - data.table is a package that extends data.frames.
  - Two of its most notable features are speed and cleaner syntax.

Tidyr Package

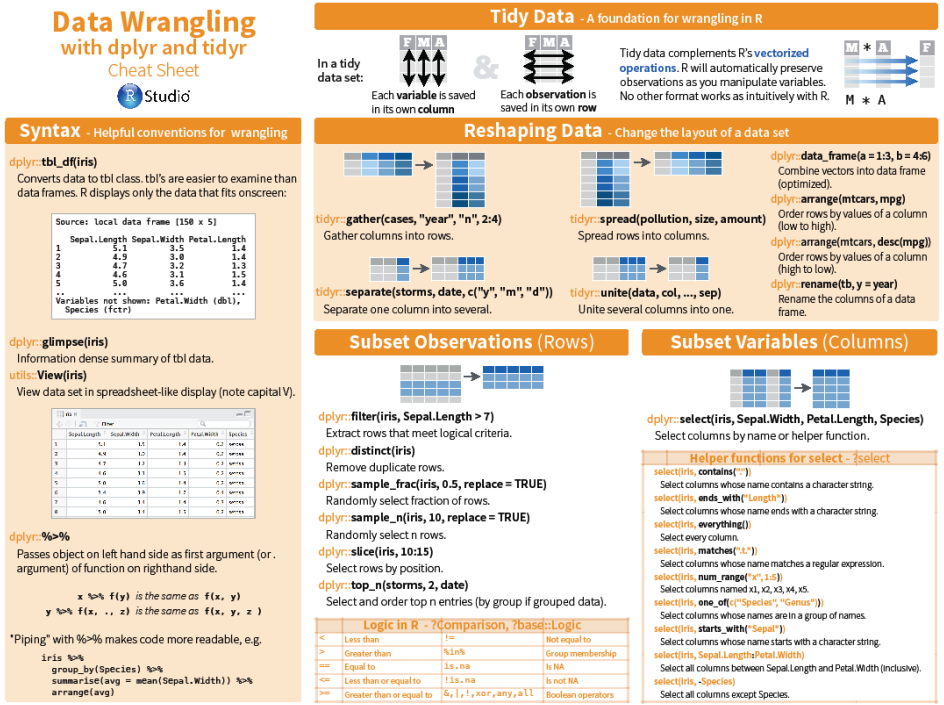
Cleaning Data

- We'll be covering the following functions:
  - **gather()** collapses multiple columns into key-pair values.
  - **spread()** reverse of gather().
  - **separate()** separates one column into multiple columns.
  - **unite()** reverse of separate().



Dplyr and Tidyr Packages

Cheat Sheet



Ggplot2 Package

Data Visualization

- **ggplot2** is an enhanced data visualization package for R, which creates stunning multi-layered graphics with ease.
- It is one of the most common and popular libraries for data visualization in R.
- It was created by Hadley Wickham.
  - He is a statistician from New Zealand who is currently Chief Scientist at RStudio and an adjunct Professor of statistics at the University of Auckland, Stanford University, and Rice University.
  - He also created packages like Dplyr and Tidyr.
- It follows a distinct **Grammar of Graphics** philosophy.
  - Built on the idea of adding layers to your visualization.
  - We can get a better understanding through some quick examples.

Ggplot2 Package

Data Visualization

- The first 3 layers are the most basic:
  - **Data** is a data frame
  - **Aesthetics** is used to indicate x and y variables. It can also be used to control the color, the size or the shape of points, the height of bars, etc…..
  - **Geometries** define the type of graphics (histogram, box plot, line plot, density plot, dot plot, ….)

- Plot = Data + Aesthetics + Geometry.

Ggplot2 Package

Data Visualization

- Step 1

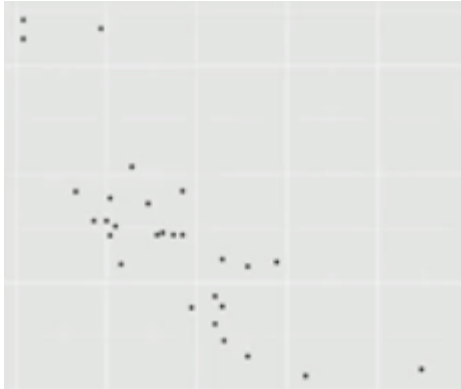
```
library(ggplot2) # Library

ggplot(data=mtcars) # Data (no plot shown yet)
```

- Step 2

```
ggplot(data=mtcars,aes(x=mpg,y=hp) # Data and Aesthetics
```

- Step 3: Will actually see some output!

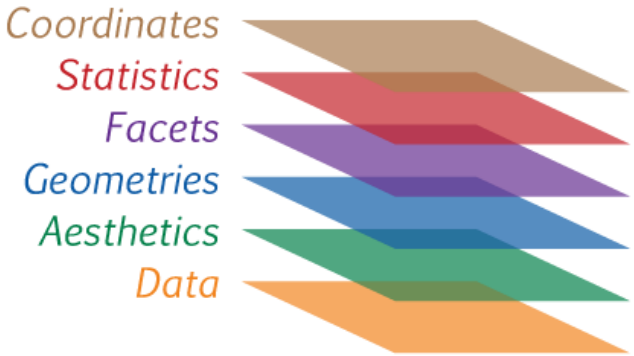
```
pl <- ggplot(data=mtcars,aes(x=mpg,y=hp)

pl + geom_point()
```



Ggplot2 Package

Data Visualization

- Next 3 layers to customize our data visualization:
  - Facets
  - Statistics
  - Coordinates

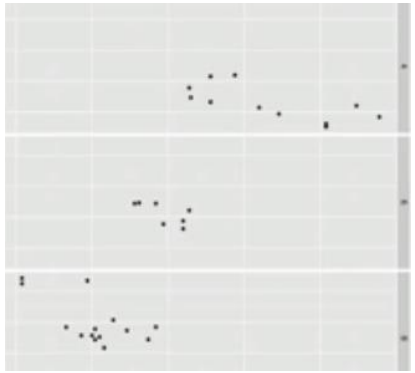


Ggplot2 Package

Data Visualization

- **Facets**: Allows us to put multiple plots on next to each other, these plots are usually related by the same dataset.

```
pl <- ggplot(data=mtcars,aes(x=mpg,y=hp) + geom_point()

pl + facet_grid(cyl ~ .)
```

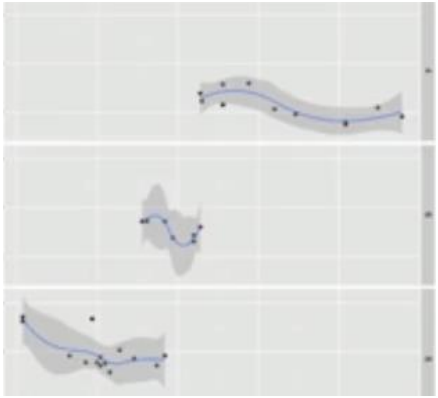


Ggplot2 Package

Data Visualization

- **Statistics**

```
pl <- ggplot(data=mtcars,aes(x=mpg,y=hp) + geom_point()

pl + facet_grid(cyl ~ .) + stat_smooth()
```



Ggplot2 Package

Data Visualization

- **Coordinates:** Learning how to deal with coordinates will allow us to size our plots correctly.

```
pl <- ggplot(data=mtcars,aes(x=mpg,y=hp) + geom_point()
pl2 <- pl + facet_grid(cyl ~ .) + stat_smooth()
pl2 + coord_cartesian(xlim = c(15, 25))
```

- *Themes*

```
pl <- ggplot(data=mtcars,aes(x=mpg,y=hp) + geom_point()
pl2 <-pl + facet_grid(cyl ~ .) +  stat_smooth()
pl2 +  coord_cartesian(xlim = c(20, 25))  + theme_bw()
```

Ggplot2 Package

What have we learned?

- **ggplot2** has a lot of options for plot types!
- The main idea is to add layers together.
- Let us now discover how to create data visualizations!
  - Scatterplots
  - Histograms
  - Barplots
  - Boxplots
  - Variable Plotting

Ggplot2 Package

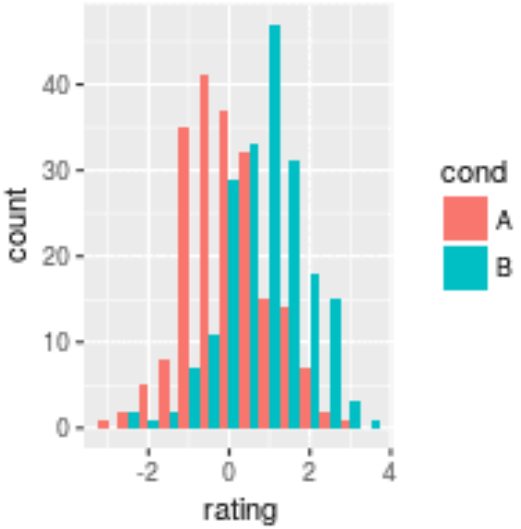
Data Visualization

- There are two major functions in **ggplot2** package: **qplot()** and **ggplot()** functions.
- **qplot()** stands for quick plot, which can be used to produce easily simple plots.
- **ggplot()** function is more flexible and robust than **qplot** for building a plot piece by piece.

Ggplot2 Package

Histograms

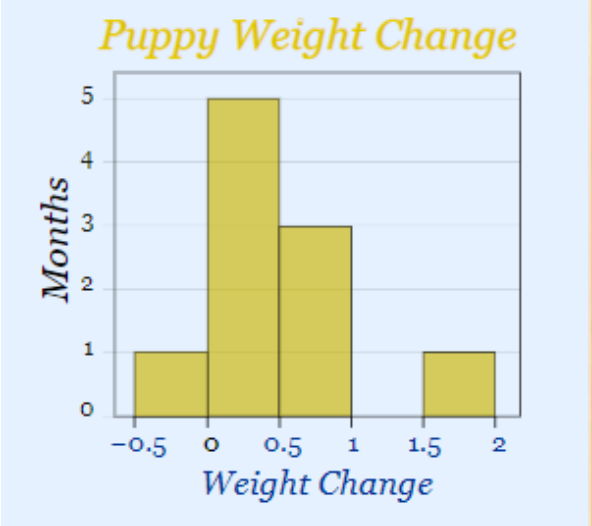
- A **histogram** is a graphical display of data using bars of different heights. In a **histogram**, each bar groups numbers into ranges.
- Taller bars show that more data falls in that range.
- To construct a histogram, the first step is to "bin" (or "bucket") the range of values—that is, divide the entire range of values into a series of intervals—and then count how many values fall into each interval. The bins are usually specified as consecutive, non-overlapping intervals of a variable. The bins (intervals) must be adjacent, and are often (but not required to be) of equal size.



Histogram Example

How much is that puppy growing?

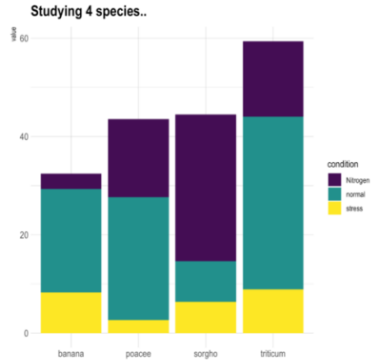
- Each month you measure how much weight your pup has gained and get these results:
  - **0.5, 0.5, 0.3, –0.2, 1.6, 0, 0.1, 0.1, 0.6, 0.4**
  - They vary from −0.2 (the pup lost weight that month) to 1.6
- Put in order from lowest to highest weight gain:
  - **–0.2, 0, 0.1, 0.1, 0.3, 0.4, 0.5, 0.5, 0.6, 1.6**
- You decide to put the results into groups of 0.5:
  - The **–0.5 to just below 0** range,
  - The **0 to just below 0.5** range,
  - etc...



Ggplot2 Package

Barplots

- **Barplots** are a useful way of displaying occurrence counts when a histogram isn't quite what you're looking for!
- In ggplot2, there are two types of bar charts, determined by what is mapped to bar height. By default, geom_bar uses stat="count" which makes the height of the bar proportional to the number of cases in each group (or if the weight aesthetic is supplied, the sum of the weights).
- If you want the heights of the bars to represent values in the data, use stat="identity" and map a variable to the y aesthetic.



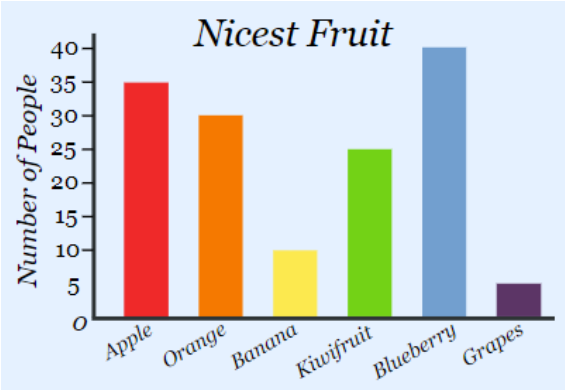
Barplot Example

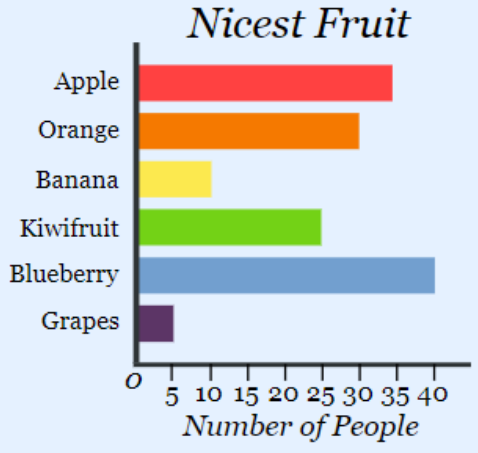
Nicest Fruit?

- A survey of 145 people asked them "Which is the nicest fruit?“:

Fruit:	Apple	Orange	Banana	Kiwifruit	Blueberry	Grapes
People:	35	30	10	25	40	5

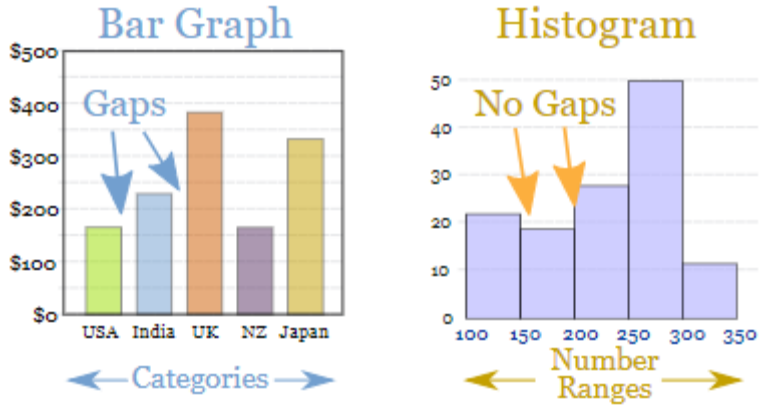
- That group of people think Blueberries are the nicest.





Histograms vs. Barplot

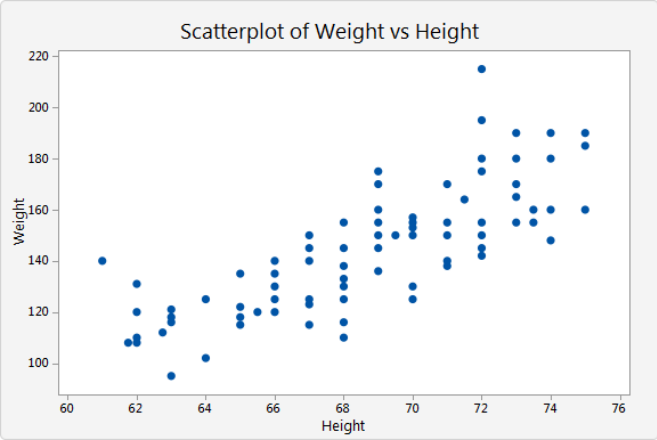
- Bar Graphs are good when your data is in **categories** (such as "Comedy", "Drama", etc).
- But when you have continuous data (such as a person's height) then use a Histogram.
- It is best to leave gaps between the bars of a Bar Graph, so it doesn't look like a Histogram.



Ggplot2 Package

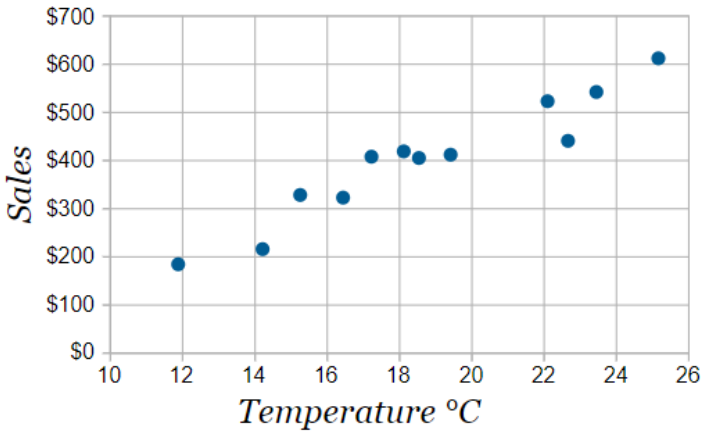
Scatterplots

- A **scatterplot** is a type of data display or mathematical diagram using Cartesian coordinates that typically shows the relationship between two numerical variables.
- Each member of the dataset gets plotted as a point whose (x , y) where x and y relates to its values for the two variables.



Scatterplot Example

- The local ice cream shop keeps track of how much ice cream they sell versus the noon temperature on that day. Here are their figures for the last 12 days:

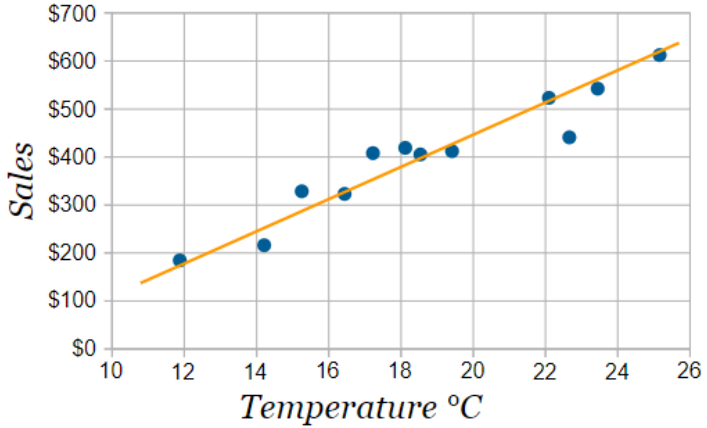


Ice Cream Sales vs Temperature	
Temperature °C	**Ice Cream Sales**
14.2°	$215
16.4°	$325
11.9°	$185
15.2°	$332
18.5°	$406
22.1°	$522
19.4°	$412
25.1°	$614
23.4°	$544
18.1°	$421
22.6°	$445
17.2°	$408

- It is now easy to see that **warmer weather leads to more sales**, but the relationship is not perfect.

Line of Best Fit

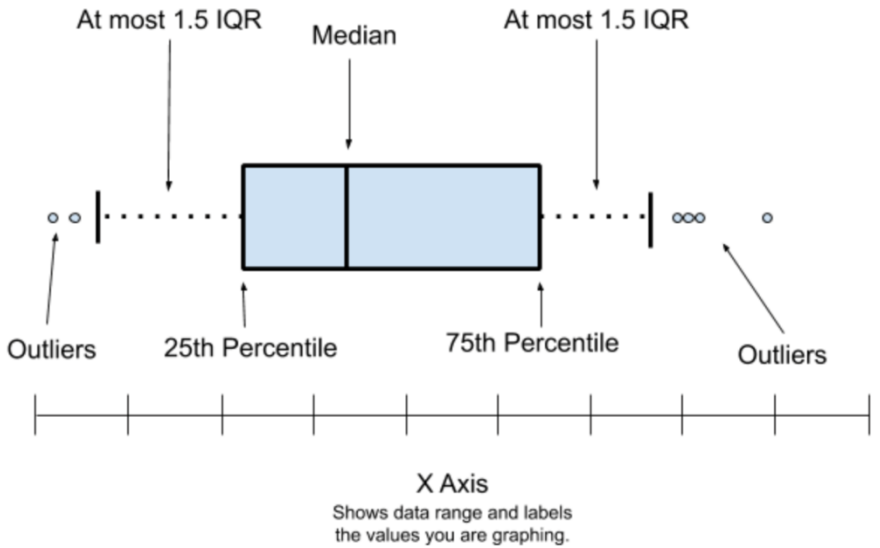
- Try to have the line **as close as possible to all points**, and as many points above the line as below.



Ggplot2 Package

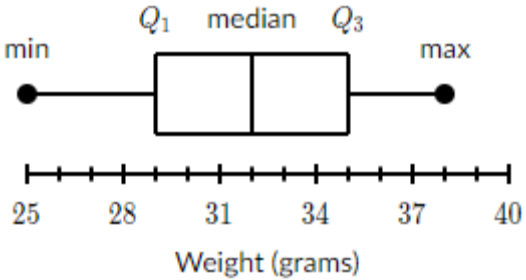
Boxplots

- **Boxplots** are convenient way of graphically depicting groups of numerical data through their quartiles.
- Box plots may also have lines extending from the boxes (whiskers) indicating variability outside the 1st and 3rd quartiles, hence the term box-and-whisker plot/diagram is also used.
- Outliers may be plotted as individual points.



Boxplot Example

- A sample of 10 boxes of raisins has these weights (in grams):
  - 30, 38, 25, 28, 29, 37, 29, 34, 35, 35

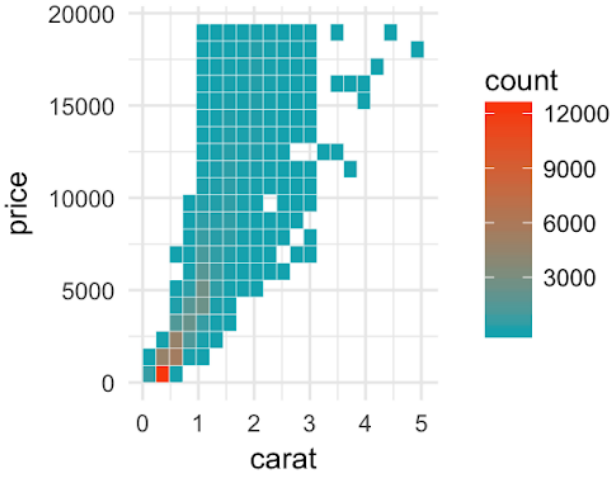


- **Step 1**: Order the data from smallest to largest.
- **Step 2:** Find the median.
- **Step 3:** Find the quartiles.
- **Step 4:** Complete the five-number summary by finding the min and the max.
  - The five-number summary divides the data into sections that each contain approximately 25% of the data in that set.

Ggplot2 Package

2 Variable Plotting

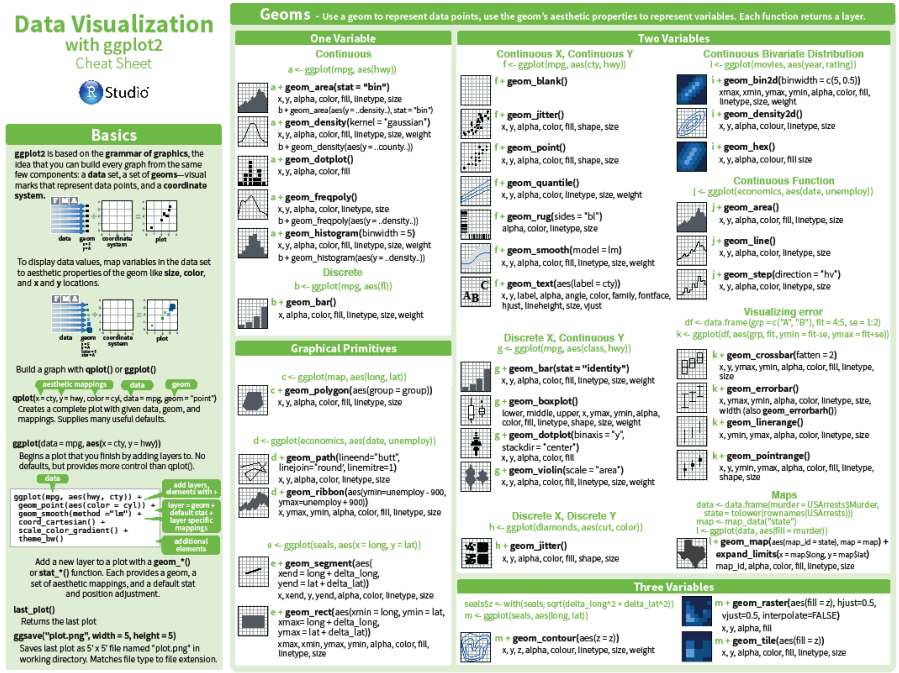
- We will briefly show some examples of how you can compare two variables from a dataset.



Ggplot2 Package

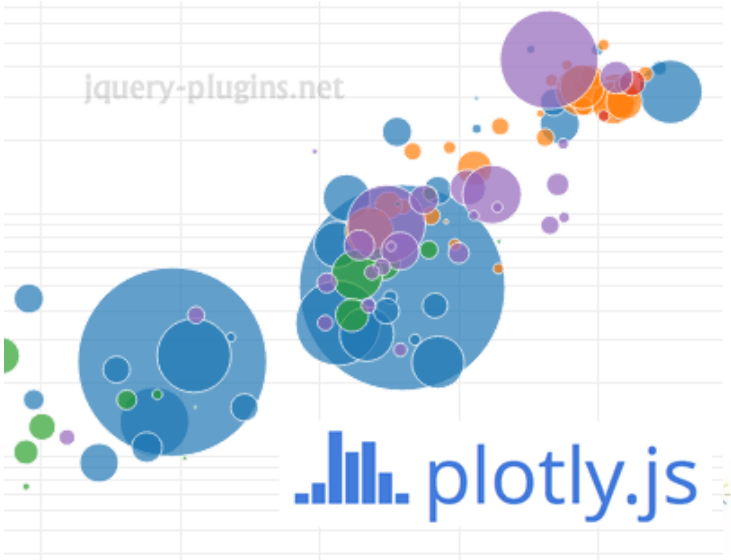
Data Visualization

- Coordinates and Faceting
- Themes
- Cheat sheet



Plotly Package

- **plotly** is an R package for creating interactive web-based graphs via plotly's JavaScript graphing library, plotly.js.
- The plotly R library contains a function ggplotly which will convert ggplot2 figures into graphs drawn with plotly.js.



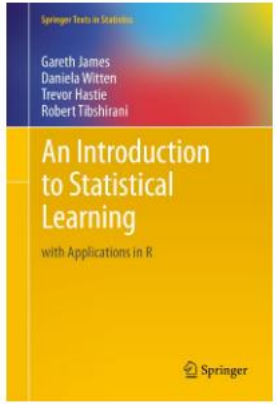
Ggplot2 Exercise

Data Visualization Project

Summary

- What is data science?
- R Data Types
- Vectors, matrices and data frames
- Data input and output with R
- Programming in R
- Functions
- R Packages
  - Dplyr for data manipulation
  - Tidyr for data cleaning
  - Ggplot2 and Plotly for data visualization

Machine Learning



http://faculty.marshall.usc.edu/gareth-james/ISL/