R Fundamentals

Diego López Tamayo *

Contents

*El Colegio de México, diego.lopez@colmex.mx

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." — Martin Fowler.

R data types:

- Numeric: Any number with or without a decimal point: 23, 0.03 and the numeric null value NA.
- Character (string): Any grouping of characters on your keyboard (letters, numbers, spaces, symbols, etc.) or text. Most strings are surrounded by single quotes: ' ... ' or double quotes " ... ", though we prefer single quotes. Sometimes you will hear this type referred to as "string."
- Logical (booleans): This data type only has two possible values— either TRUE or FALSE (without quotes). It's helpful to think of logical types or booleans as on and off switches or as the answers to a "yes" or "no" question. Vectors: A list of related data that is all the same type.
- NA: This data type represents the absence of a value, and is represented by the keyword NA (without quotes) but it has its own significance in the context of the different types. That is there is a numeric NA, a character NA, and a logical NA.

```
class(2) # numeric
```

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

```
class('hello') # character
```

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

```
class('23') #character
```

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

```
class (FALSE) #logical
```

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

```
class(NA) #logical
```

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

Variables

Now that you know how R classifies some of the basic information types, let's figure out how to store them.

Assignment operator, an arrow sign (<-)

Variables can't have spaces or symbols in their names other than an underscore (_). They can't begin with numbers but they can have numbers after the first letter.

Vectors

Vectors are a list-like structure that contain items of the same data type.

You can check the type of elements in a vector by using typeof(vector_name)

You can check the length of a vector by using length(vector_name)

You can access individual elements in the vector by using [] and placing the element position inside the brackets

Note: In R, you start counting elements at position one, not zero.

Conditionals

If we are analyzing data for the summer, then we will only want to look at data that falls in June, July, and September.

We can perform a task based on a condition using an if statement:

```
if (TRUE) {
  print('This message will print!')
}
```

```
## [1] "This message will print!"
```

Inside the parentheses (), a condition is provided that evaluates to TRUE or FALSE.

If the condition evaluates to true, the code inside the curly braces {} runs, or executes. If the condition evaluates to false, the code inside the block won't execute.

There is also a way to add an else statement. An else statement must be paired with an if statement, and together they are referred to as an if…else statement.

```
if(T){
  message <- 'I execute this when true!'
} else {
  message <- 'I execute this when false!'
}

print (message)
```

```
## [1] "I execute this when true!"
```

These if…else statements allow us to automate solutions to yes-or-no questions, also known as binary decisions.

Comparison Operators

When writing conditional statements, sometimes we need to use different types of operators to compare values.

- Less than: <
- Greater than: >
- Less than or equal to: <=
- Greater than or equal to: >=
- Is equal to: ==
- Is NOT equal to: !=

```
10 < 12 # Evaluates to TRUE
```

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

Logical Operators

We can use logical operators to add more sophisticated logic to our conditionals. There are three logical operators:

- the AND operator (&)
- the OR operator (|)
- the NOT operator, otherwise known as the bang operator (!)

When we use the & operator, we are checking that two things are true. Both conditions must evaluate to true for the entire condition to evaluate to true and execute.

- if (stopLight == 'green' & pedestrians == 0) { print('Go!'); } else { print('Stop'); }

If we only care about either condition being true, we can use the | operator. When using the | operator, only one of the conditions must evaluate to true for the overall statement to evaluate to true. If the first condition in an | statement evaluates to true, the second condition won't even be checked.

- if (day == 'Saturday' | day == 'Sunday') { print('Enjoy the weekend!') } else { print('Do some work.') }

The ! NOT operator reverses, or negates, the value of a TRUE value. Essentially, the ! operator will either take a true value and pass back false, or it will take a false value and pass back true.

excited <- TRUE print(!excited) # Prints FALSE

Example:

```
message <- 'Should I pack an umbrella?'
weather <- 'cloudy'
high_chance_of_rain <- T

if(weather== 'cloudy' & high_chance_of_rain == T){
  message <- 'Pack umbrella!'
  } else {
    message <- 'No need for umbrella!'
  }
print(message)
```

```
## [1] "Pack umbrella!"
```

Calling a Function

Functions are actions we can perform. Just like "print"

We call, or use, these functions by stating the name of the function and following it with an opening and closing parentheses. Between the parenthesis, we usually pass in an argument, or a value that the function uses to conduct an action, i.e. functionName(value).

```
sort(c(2,4,10,5,1)); # Outputs c(1,2,4,5,10)
```

```
## [1]  1  2  4  5 10
```

```
length(c(2,4,10,5,1)); # Outputs 5
```

```
## [1] 5
```

```
sum(5,15,10) #Outputs 30
```

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

- The sort() function is called with a parameter of the vector c(2,4,10,5,1). The result is a sorted vector c(1,2,4,5,10) with the values in ascending order.
- The length() function return the value 5 because there were five items in the vector.
- The function sum() added up all of the arguments we passed to it.
- The unique() function takes a vector argument and returns a vector with only the unique elements in that vector (removing all duplicates).

- The sqrt() functions gives you the square root of a Numeric argument.
- The floor() function rounds a decimal down to the next integer, and the ceiling() function will round up to the next integer.

Importing Packages

A package is a bundle of code that makes coding certain tasks easier. There are all sorts of packages for all sorts of purposes, ranging from visualizing and cleaning data, to ordering pizza or posting a tweet.

Base R refers the R language by itself and all that it can do without importing any packages.

. You only need to run this command the first time you install a package, after that there is no need to run it: install.packages('package-name'). To import a package you simply: library(package-name)

You can look up documentation for different packages available in R at the CRAN (Comprehensive R Archive Network).

- Dplyr is a package used to clean, process, and organize data which you will use as you learn about R.

Examples of Dplyr:

```
#Inspect data frame ""artists"
#head(artists)
#summary(artists)
```

```
#artists %>%
#  select(-country,-year_founded,-albums) %>%
#  filter(spotify_monthly_listeners > 20000000, genre != 'Hip Hop') %>%
#  arrange(desc(youtube_subscribers))
```

Data Frames

The dplyr package in R is designed to make data manipulation tasks simpler and more intuitive than working with base R functions only. dplyr provides functions that solve many challenges that arise when organizing tabular data (i.e., data in a table with rows and columns)

dplyr and readr are a part of the tidyverse, a collection of R packages designed for data science.The tidyverse is a package itself, and it can be imported at the top of your file if you need to use any of the packages it contains.

A data frame is an R object that stores tabular data in a table structure made up of rows and columns. Each column has a name and stores the values of one variable. Each row contains a set of values, one from each column. The data stored in a data frame can be of many different types: numeric, character, logical, or NA.

Note: when working with dplyr, you might see functions that take a data frame as an argument and output something called a tibble. Tibbles are modern versions of data frames in R, and they operate in essentially the same way. The terms tibble and data frame are often used interchangeably.

CSVs

CSV (comma separated values) is a text-only spreadsheet format. CSV (comma separated values) is a text-only spreadsheet format. You can find CSVs in lots of places such as: online datasets from governments and companies, exported from Excel or Google Sheets or exported from SQL.

The first row of a CSV contains column headings. All subsequent rows contain values.

When you have data in a CSV, you can load it into a data frame in R using readr's read_csv() function:

```
#df <- read_csv('my_csv_file.csv')
```

You can also save data from a data frame to a CSV using readr's write_csv() function. By default, this method will save the CSV file to your current directory:

```
#write_csv(df,'new_csv_file.csv')
```

Inspecting Data Frames

When you load a new data frame from a CSV, you want to get an understanding of what the data looks like.

The head() function returns the first 6 rows of a data frame. If you want to see more rows, you can pass an additional argument n to head(). For example, head(df,8) will show the first 8 rows.

The function summary() will return summary statistics such as mean, median, minimum and maximum for each numeric column while providing class and length information for non-numeric columns.

Piping

Each of the dplyr functions you will explore takes a data frame as its first argument.

One of the most appealing aspects of dplyr is the ability to easily manipulate data frames.

The pipe operator, or %>%, helps increase the readability of data frame code by piping the value on its left into the first argument of the function that follows it.

```
#df %>%
#  head()
```

The true power of pipes comes from the ability to link multiple function calls together.

Any time you load a package from the tidyverse, like dplyr, %>% will automatically be loaded!

Selecting columns

You can select the appropriate columns for your analysis using dplyr's select() function:

- select() takes a data frame as its first argument
- all additional arguments are the desired columns to select
- select() returns a new data frame containing only the desired columns

Example: From dataframe with customers data, you only want a data frame with age and gender.

```
#select(customers,age,gender)
```

Of course with pipes you can simplify the readability of your code by using the pipe:

```
#customers %>%
#  select(age,gender)
```

When using the pipe, you can read the code as: from the customers table, select() the age and gender columns.

Excluding columns

Sometimes rather than specify what columns you want to select from a data frame, it's easier to state what columns you do not want to select.

Example: To remove name and phone from customers data frame use select(-variable_name)

```
#customers %>%
#  select(-name,-phone)
```

Filtering Rows with Logic operators

In addition to subsetting a data frame by columns, you can also subset a data frame by rows using dplyr's filter() function and comparison operators!

Example: Consider an orders data frame that contains data related to the orders for an e-commerce shoe company: Let's say you want to find all orders made by customers with the first name 'Joyce'.

```
# orders %>%
#  filter(first_name == 'Joyce')
```

- the orders data frame is piped into filter()
- the condition first_name == 'Joyce' is given as an argument
- a new data frame containing only the rows where first_name == 'Joyce' is returned

What if you have multiple conditions you want to be met? Not a problem! To find all orders made of faux-leather AND costing more than 25:

```
# orders %>%
#  filter(shoe_material == 'faux-leather',price > 25)
```

- the orders data frame is again piped into filter()
- the conditions shoe_material == 'faux-leather' and price > 25 are given as arguments
- a new data frame containing only the rows where both conditions were met is returned

You can provide any number of conditions that you please, as long as you separate each condition by a comma as its own argument. Note: each condition that you list must be met for a row to be returned!

The filter() function also allows for more complex filtering with the help of logical operators!

Example: You are interested in seeing all orders that were for 'clogs' OR that cost less than 20. Using the or operator (|): a new data frame is returned containing only rows where shoe_type is 'clogs' or price is less than 20

```
#orders %>%
#  filter(shoe_type == 'clogs' | price < 20)
```

What if you want to find all orders where shoes in any color but red were purchased. Using the not or bang operator (!): a new data frame is returned containing only rows where shoe_color is not 'red'

```
# orders %>%
#  filter(!(shoe_color == red))
```

- the condition that should not be met is wrapped in parentheses, preceded by !, and given as an argument to filter()

Arranging rows

Sometimes all the data you want is in your data frame, but it's all unorganized! Step in the handy dandy dplyr function arrange()! arrange() will sort the rows of a data frame in ascending order by the column provided as an argument.

- For numeric columns, ascending order means from lower to higher numbers. For character columns, ascending order means alphabetical order from A to Z.

Example: To arrange the customers in ascending order by name (A to Z):

```
#customers %>%
#  arrange(name)
```

arrange() can also order rows by descending order! To arrange the customers in descending order by age:

```
#customers %>%
#  arrange(desc(age))
```

If multiple arguments are provided to arrange(), it will order the rows by the column given as the first argument and use the additional columns to break ties in the values of preceding columns.

Example of iterative pipes

```
# select columns, filter and arrange rows
#artists <- artists %>% select(-country,-year_founded,-albums) %>% filter(spotify_monthly_listeners > 2
```

Adding a Column(s)

Sometimes you might want to add a new column to a data frame. This new column could be a calculation based on the data that you already have.

You can add a new column to the data frame using the mutate() function. mutate() takes a name-value pair as an argument. The name will be the name of the new column you are adding, and the value is an expression defining the values of the new column in terms of the existing columns.

```
# df <- df %>% mutate(new_column_name = values)
```

mutate() can also take multiple arguments to add any number of new columns to a data frame:

```
# df <- df %>% mutate(new_column_name = values,second_new_column_name = values)
```

Example: We add the column "prueba" at the end of the dataframe.

```
#gpa1 %>% mutate(prueba=T)
```

Note: It does not replace the original dataframe, if you want to add this new column to your dataframe, replace the variable with the mutate command.

```
#gpa1 <- gpa1 %>% mutate(prueba=T)
```

Transmute Columns

When creating new columns from a data frame, sometimes you are interested in only keeping the new columns you add, and removing the ones you do not need. dplyr's transmute() function will add new columns while dropping the existing columns that may no longer be useful for your analysis.

Like mutate(), transmute() takes name-value pairs as arguments. The names will be the names of the new columns you are adding, and the values are expressions defining the values of the new columns. The difference, however, is that transmute() returns a data frame with only the new columns.

Same example as before: Adding the column "prueba" and dropping everything else.

```
gpa1 <- gpa1 %>% transmute(prueba=T)
```

If we dropped a column by mistake and want it back, change the command in this way:

Example: Drop everything except age column.

```
#gpa1 <- gpa1 %>% transmute(age=age, prueba=T)
```

Rename Columns

Since dplyr functions operate on data frames using column names, it is often useful to update the column names of a data frame so they are as clear and meaningful as possible. dplyr's rename() function allows you to easily do this.

rename() can take any number of arguments, where each new column name is assigned to replace an old column name in the format new_column_name = old_column_name.

rename() returns a new data frame with the updated column names.

You can confirm the names of the columns have been updated using either of the base R functions names() or colnames(), which take a data frame as an argument and return a vector containing the column names.

Example: Changing names of gpa1 column into spanish.

```
#gpa1 <- gpa1 %>% rename(edad=age)
#gpa1
```

Remember:

- add new columns to a data frame using mutate()
- add new columns to a data frame and drop existing columns using transmute()
- change the column names of a data frame using rename()

Example:

```
#dogs <- dogs %>%
#  transmute(breed = breed,
#            height_average_feet = ((height_low_inches + height_high_inches)/2)/12,
#            popularity_change_15_to_16 = rank_2016 - rank_2015) %>%
#  arrange(desc(popularity_change_15_to_16))
```

Data cleaning

Diagnose the Data

For data to be tidy, it must have: - Each variable as a separate column - Each row as a separate observation

The first step of diagnosing whether or not a dataset is tidy is using base R and dplyr functions to explore and probe the dataset.

We have seen most of the functions we often use to diagnose a dataset for cleaning. Some of the most useful ones are:

- head() — display the first 6 rows of the table
- summary() — display the summary statistics of the table
- colnames() — display the column names of the table

Dealing with Multiple Files

Often, you have the same data separated out into multiple files.

Let's say that you have a ton of files following the filename structure: 'file_1.csv', 'file_2.csv', 'file_3.csv', and so on. The power of dplyr and tidyr is mainly in being able to manipulate large amounts of structured data, so you want to be able to get all of the relevant information into one table so that you can analyze the aggregate data.

You can combine the base R functions list.files() and lapply() with readr and dplyr to organize this data better, as shown below:

```
# files <- list.files(pattern = "file_.*csv")
# df_list <- lapply(files,read_csv)
# df <- bind_rows(df_list)
```

- The first line uses list.files() and a *regular expression*, a sequence of characters describing a pattern of text that should be matched, to find any file in the current directory that starts with 'file_' and has an extension of csv, storing the name of each file in a vector files.
- The second line uses lapply() to read each file in files into a data frame with read_csv(), storing the data frames in df_list
- The third line then concatenates all of those data frames together with dplyr's bind_rows() function

Example:

You have 10 different files containing 100 students each. These files follow the naming structure:

- exams_0.csv
- exams_1.csv
- ... up to exams_9.csv

You are going to read each file into an individual data frame and then combine all of the entries into one data frame.

First, create a variable called student_files and set it equal to the list.files() of all of the CSV files we want to import.

```
# list files
#student_files <- list.files(pattern = "exams_.*csv")
# read files
#df_list <- lapply(student_files,read_csv)
# concatenate data frames
#students <- bind_rows(df_list)
```

Reshaping your Data

We can use tidyr's gather() function to do this transformation. gather() takes a data frame and the columns to unpack:

Example:

```
#df %>%
#  gather('Checking','Savings',key='Account Type',value='Amount')
```

The arguments you provide are:

- df: the data frame you want to gather, which can be piped into gather()
- Checking and Savings: the columns of the old data frame that you want to turn into variables
- key: what to call the column of the new data frame that stores the variables
- value: what to call the column of the new data frame that stores the values

Note: The dplyr function count() takes a data frame and a column as arguments and returns a table with counts of the unique values in the named column.

```
# unique value counts of exam
# exam_counts <- students %>% count(exam)
```

Dealing with Duplicates

Often we see duplicated rows of data in the data frames we are working with. This could happen due to errors in data collection or in saving and loading the data.

To check for duplicates, we can use the base R function duplicated(), which will return a logical vector telling us which rows are duplicate rows. TRUE means: Every value in this row is the same as in another row. We expect (usually to see all the results as FALSE)

```
data("gpa1")
#gpa1%>% duplicated()
```

We can use the dplyr distinct() function to remove all rows of a data frame that are duplicates of another row. (If there was no duplicated rows, we end up with the same df)

```
#gpa1%>% distinct()
```

The distinct() function removes rows only when all values are the same in across rows. If there's a single column with different values between two rows, distinct() will keep both of them.

Let's say we want to remove all the rows with duplicated values in a single column (let's say the "names" column). We would need to specify a subset:

```
#df %>%
#  distinct(names,.keep_all=TRUE)
```

By default, this keeps the first occurrence of the duplicate.

Make sure that the columns you drop duplicates from are specifically the ones where duplicates don't belong. You wouldn't want to drop duplicates from columns where it's okay if multiple entries are the same (for example "ages" column).

Note: table() is a base R function that takes any R object as an argument and returns a table with the counts of each unique value in the object.

```
#files%>%table()
#duplicates <- files%>%duplicated()
#duplicates <- duplicates %>% table()
#duplicates
```

Splitting By Index

In trying to get clean data, we want to make sure each column represents one type of measurement. Often, multiple measurements are recorded in the same column, and we want to separate these out so that we can do individual analysis on each variable.

Let's say we have a column "birthday" with data formatted in MMDDYYYY format. In other words, "11011993" represents a birthday of November 1, 1993. We want to split this data into day, month, and year so that we can use these columns as separate features.

In this case, we know the exact structure of these strings. The first two characters will always correspond to the month, the second two to the day, and the rest of the string will always correspond to year. We can easily break the data into three separate columns by splitting the strings into substrings using str_sub(), a helpful function from the stringr package:

```
#To separate "birthday" with data formatted in MMDDYYYY.

# Create the 'month' column
#df %>%
#  mutate(month = str_sub(birthday,1,2))
```

```
# Create the 'day' column
#df %>%
#  mutate(day = str_sub(birthday,3,4))

# Create the 'year' column
#df %>%
#  mutate(year = str_sub(birthday,5))
```

- The first command takes the characters starting at index 1 and ending at index 2 of each value in the birthday column and puts it into a month column.
- The second command takes the characters starting at index 3 and ending at index 4 of each value in the birthday column and puts it into a day column.
- The third command takes the characters starting at index 5 and ending at the end of the value in the birthday column and puts it into a year column.

Splitting By Character

If the two variables we want to separate within a column are not the same lenght, we cannot longer use str_sub(). Instead, if we know that we want to split along the "_" symbol. We can thus use the tidyr function separate() to split the column into two separate columns:

Example:

```
# Create the 'user_type' and 'country' columns
# df %>%
#  separate(type,c('user_type','country'),'_')
```

- type is the column to split
- c('user_type','country') is a vector with the names of the two new columns
- '_' is the character to split on

Another example (Provide as an extra argument to the separate() function extra ='merge'. This will ensure that middle names or two-word last names will all end up in the last_name column.)

```
#students <- students %>% separate(full_name,c('first_name','last_name'),' ',extra='merge')
```

Looking at Data Types

Each column of a data frame can hold items of the same data type. The data types that R uses are: - character (chr) - numeric (real or decimal) (num) - integer (int) - logical (logi) - complex.

Often, we want to convert between types so that we can do better analysis. If a numerical category like "num_users" is stored as a vector of characters (string) instead of numerics, for example, it makes it more difficult to do something like make a line graph of users over time.

To see the types of each column of a data frame, we can use: str(df) str() displays the internal structure of an R object. Calling str() with a data frame as an argument will return a variety of information, including the data types.

If we try to use a numeric funtion into a non numeric variable well get something like

```
# Warning in mean.default(score): argument is not numeric or logical: returning NA
```

String Parsing

Sometimes we need to modify strings in our data frames to help us transform them into more meaningful metrics.

For example, suposse you have a column named “Prices” composed of character strings representing dollar amounts This column could be much better represented as numeric, so that we could take the mean, calculate other aggregate statistics, or compare different observations to one another in terms of price.

First, we can use a regular expression, a sequence of characters that describe a pattern of text to be matched, to remove all of the dollar signs. The base R function gsub() will remove the $ from the price column, replacing the symbol with an empty string ”:

```
#df %>%
#  mutate(price=gsub('\\$','',price))
```

Then, we can use the base R function as.numeric() to convert character strings containing numerical values to numeric:

```
#df %>%
#  mutate(price = as.numeric(price))
```

Great! We have looked at a number of different methods we may use to get data into the format we want for analysis.

Specifically, we have covered:

- diagnosing the “tidiness” of data
- combining multiple files
- reshaping data
- changing the types of values
- manipulating strings to represent data better

Data visualization in R

Layers and Geoms

When you learn grammar in school you learn about the basic units to construct a sentence. The basic units in the “grammar of graphics” consist of:

- The data or the actual information you wish to visualize.
- The geometries, shortened to “geoms”, describe the shapes that represent our data. Whether it be dots on a scatter plot, bar charts on the graph, or a line to plot the data! The list goes on. Geoms are the shapes that “map” our data.
- The aesthetics, or the visual attributes of the plot, including the scales on the axes, the color, the fill, and other attributes concerning appearance.

Another key component to understand is that in ggplot2, geoms are “added” as layers to the original canvas which is just an empty plot with data associated to it. Once you learn these three basic grammatical units, you can create the equivalent of a basic sentence, or a basic plot.

The ggplot() function

We’ll use the Dataframe ‘mtcars’ from the base library in R.

```
data(mtcars)
```

The first thing you’ll need to do to create a ggplot object is invoke the ggplot() function. Conceptualize this step as initializing the “canvas” of the visualization. In this step, it’s also standard to associate the data frame the rest of the visualization will use with the canvas. What do we mean by “the rest” of the visualization? We mean all the layers you’ll add as you build out your plot. As we mentioned, at its heart, a ggplot visualization is a combination of layers that each display information or add style to the final graph. You “add” these layers to a starting canvas, or ggplot object, with a + sign. For now, let’s stop to understand that any arguments inside the ggplot() function call are inherited by the rest of the layers on the plot.

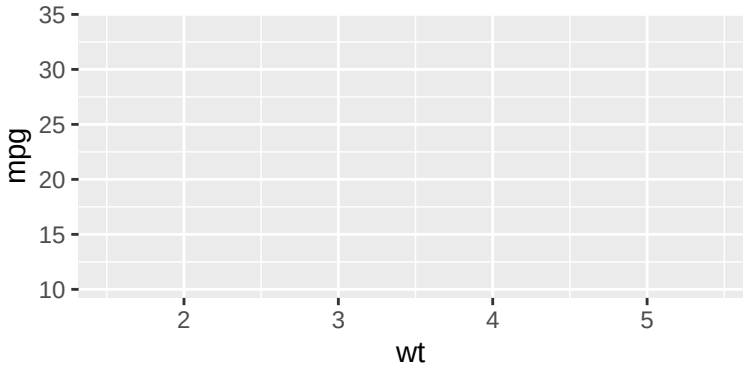
Here we invoke ggplot() to create a ggplot object and assign the dataframe df, saving it inside a variable named plot1. Note: The code assigns the value of the canvas to plot1 and then states the variable name plot1 after so that the visualization is rendered in the notebook.

Any layers we add to plot1 would have access to the dataframe. We mentioned the idea of aesthetics before. It's important to understand that any aesthetics that you assign as the ggplot() arguments will also be inherited by other layers. We'll explore what this means in depth too, but for now, it's sufficient to conceptualize that arguments defined inside ggplot() are inherited by other layers.

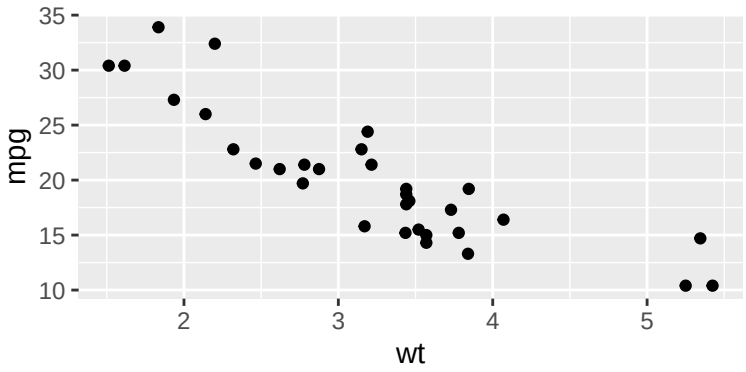
```
plot1 <- ggplot(data=mtcars)
#Notice how initially, a ggplot object is created as a blank canvas. Notice that the aesthetics are the
plot1
```



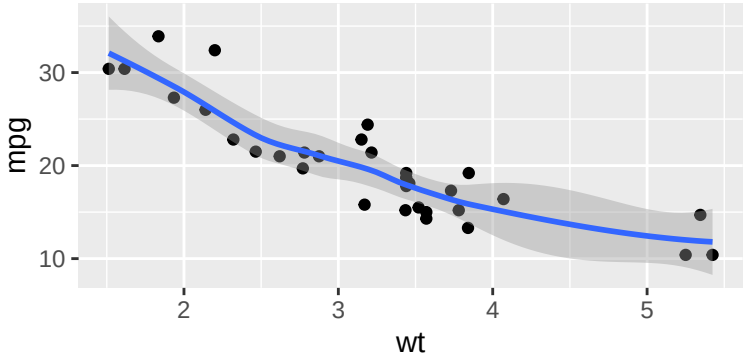
```
plot1 <- ggplot(data=mtcars, aes(x=wt, y=mpg))
#You could individually set the scales for each layer, but if all layers will use the same scales, it m
plot1
```




```
plot1 <- ggplot(data=mtcars, aes(x=wt, y=mpg)) + geom_point()
plot1
```



```
plot1 <- ggplot(data=mtcars, aes(x=wt, y=mpg)) + geom_point() + geom_smooth()
#When two subsequent layers are added, a scatter plot and a line of best fit, both of those layers are
plot1
```



Associating the Data

Before we go any further, let's stop to understand when the data gets bound to the visualization:

- Data is bound to a ggplot2 visualization by passing a data frame as the first argument in the ggplot() function call. You can include the named argument like ggplot(data=df_variable) or simply pass in the data frame like ggplot(data frame).
- Because the data is bound at this step, this means that the rest of our layers, which are function calls we add with a + plus sign, all have access to the data frame and can use the column names as variables.

For example, assume we have a data frame sales with the columns cost and profit. In this example, we assign the data frame sales to the ggplot() object that is initailized:

```
#viz <- ggplot(data=sales) +
#       geom_point(aes(x=cost, y=profit))
#viz   # renders plot
```

In the example above:

- The ggplot object or canvas was initialized with the data frame sales assigned to it
- The subsequent geom_point layer used the cost and profit columns to define the scales of the axes for that particular geom. Notice that it simply referred to those columns with their column names.
- We state the variable name of the visualization ggplot object so we can see the plot.

Note: There are other ways to bind data to layers if you want each layer to have a different dataset, but the most readable and popular way to bind the dataframe happens at the ggplot() step and your layers use data from that dataframe.

What are aesthetics?

In the context of ggplot, aesthetics are the instructions that determine the visual properties of the plot and its geometries. Aesthetics can include things like the scales for the x and y axes, the color of the data on the plot based on a property or simply on a color preference, or the size or shape of different geometries. There are two ways to set aesthetics, by manually specifying individual attributes or by providing aesthetic mappings . We'll explore aesthetic mappings first.

Aesthetic mappings "map" variables from the data frame to visual properties in the plot. You can provide aesthetic mappings in two ways using the aes() mapping function:

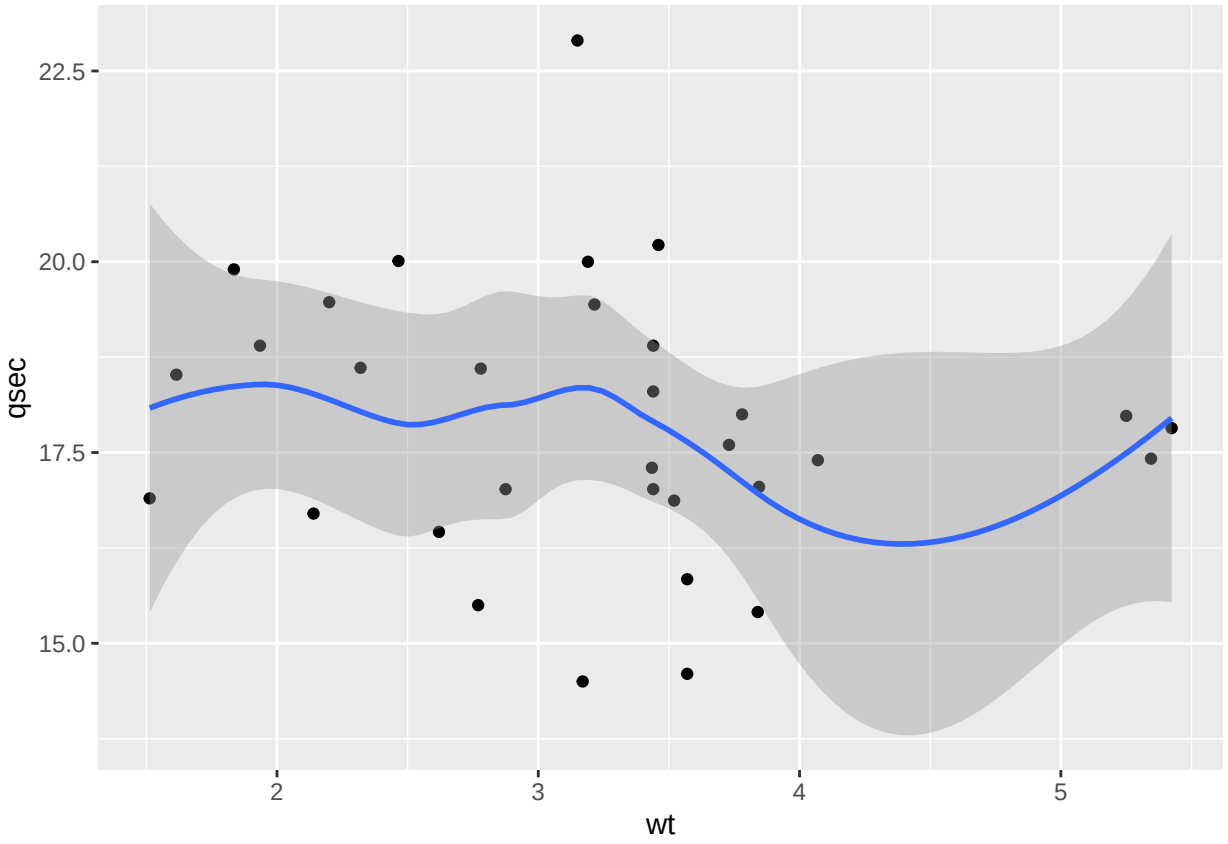
$$\text{Aesthetics} = aes() = f : Variables \Rightarrow Plot$$

- At the canvas level: All subsequent layers on the canvas will inherit the aesthetic mappings you define when you create a ggplot object with ggplot().
- At the geom level: Only that layer will use the aesthetic mappings you provide.

Let's discuss inherited aesthetics first, or aesthetics you define at the canvas level.

Here's an example of code that assigns aes() mappings for the x and y scales at the canvas level:

```
plot1 <- ggplot(data=mtcars, aes(x=wt, y=qsec)) +
       geom_point() +
       geom_smooth()
plot1
```



In the example above:

- The aesthetic mapping is wrapped in the aes() aesthetic mapping function as an additional argument to ggplot().
- Both of the subsequent geom layers, geom_point() and geom_smooth() use the scales defined inside the aesthetic mapping assigned at the canvas level.

You should set aesthetics for subsequent layers at the canvas level if all layers will share those aesthetics.

Adding Geoms

Before we teach you how to add aesthetics specific to a geom layer, let's create our first geom! As mentioned before, geometries or geoms are the shapes that represent our data. In ggplot, there are many types of geoms for representing different relationships in data. You can read all about each one in the layers section of the ggplot2 documentation.

Once you learn the basic grammar of graphics, all you'll have to do is read the documentation of a particular geom and you'll be prepared to make a plot with it following the general pattern.

For simplicity's sake, let's start with the scatterplot geom, or geom_point(), which simply represents each datum as a point on the grid. Scatterplots are great for graphing paired numerical data or to detect a correlation between two variables.

The following code adds a scatterplot layer to the visualization:

```
#plot <- ggplot(data=df, aes(x=col1,y=col2)) +
#       geom_point()
```

In the code above:

- Notice the layer is being added by using a + sign which comes after the ggplot object is created, and it comes on the same line.
- The geom_point() function call is what adds the points layer to the plot. This call can take arguments but we are keeping it simple for now.

Another popular layer that allows you to eye patterns in the data by completing a line of best fit is the geom_smooth() layer. This layer, by nature, comes with a gray error band. You could add a smoooth layer to the plot by typing the following:

```
#plot <- ggplot(data=df, aes(x=col1,y=col2)) +
#       geom_point()+
#       geom_smooth()
```

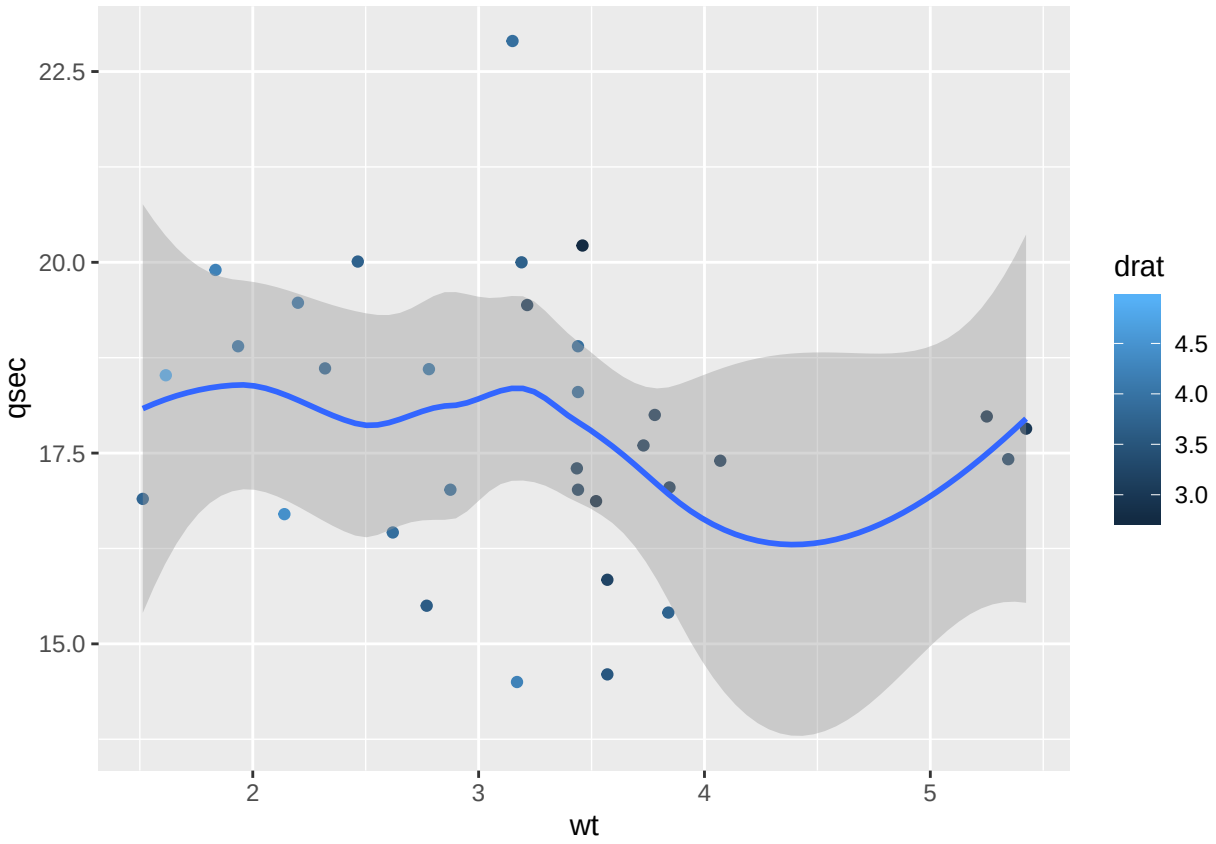
- Notice that you can add layers one on top of the other. We added the smooth line after adding the geom_point() layer. We could have just included the point layer, or just the line-of-best-fit layer. But the combination of the two enhances our visual understanding of the data, so they make a great pairing.
- It is nice to put each layer on its own line although it is not necessary, since it improves readability in the long run if you're collaborating with other people.

Geom Aesthetics

Sometimes, you'll want individual layers to have their own mappings. For example, what if we wanted the scatterplot layer to classify the points based on a data-driven property? We achieve this by providing an aesthetic mapping for that layer only.

Let's explore the aesthetic mappings for the geom_point() layer. What if we wanted to color-code the points on the scatterplot based on a property? It's possible to customize the color by passing in an aes() aesthetic mapping with the color based on a data-driven property. Observe this example:

```
plot1 <- ggplot(data=mtcars, aes(x=wt, y=qsec))
plot1 +
       geom_point(aes(color=drat)) +
       geom_smooth()
```



The code above would only change the color of the point layer, it would not affect the color of the smooth layer since the aes() aesthetic mapping is passed at the point layer.

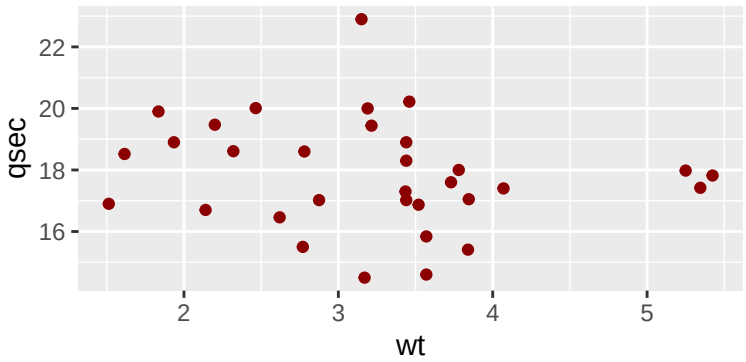
Note: You can read about the individual aesthetics available for each geom when you read its documentation. There are some aesthetics shared across geoms and others that are specific to a particular ones.

Manual Aesthetics

We've reviewed how to assign data-driven aesthetic mappings at the canvas level and at the geom level. However, sometimes you'll want to change an aesthetic based on visual preference and not data. You might think of this as "manually" changing an aesthetic.

If you have a pre-determined value in mind, you provide a named aesthetic parameter and the value for that property without wrapping it in an aes(). For example, if you wanted to make all the points on the scatter plot layer dark red because that's in line with the branding of the visualization you are preparing, you could simply pass in a color parameter with a manual value darkred or any color value like so:

```
plot1 + geom_point(color='darkred')
```



- Note that we did not wrap the color argument inside aes() because we are manually setting that aesthetic
- Here are more aesthetics for the geom_point() layer: x, y, alpha, color, fill, group, shape, size, stroke.
- The alpha aesthetic describes opacity of the points, and the shape of the dots could be different than a dot. Read more about the values each of these aesthetics take in the geom_point() layer documentation. .

We advise that your aesthetic choices have intention behind them. Too much styling can overcomplicate the appearance of a plot, making it difficult to read.

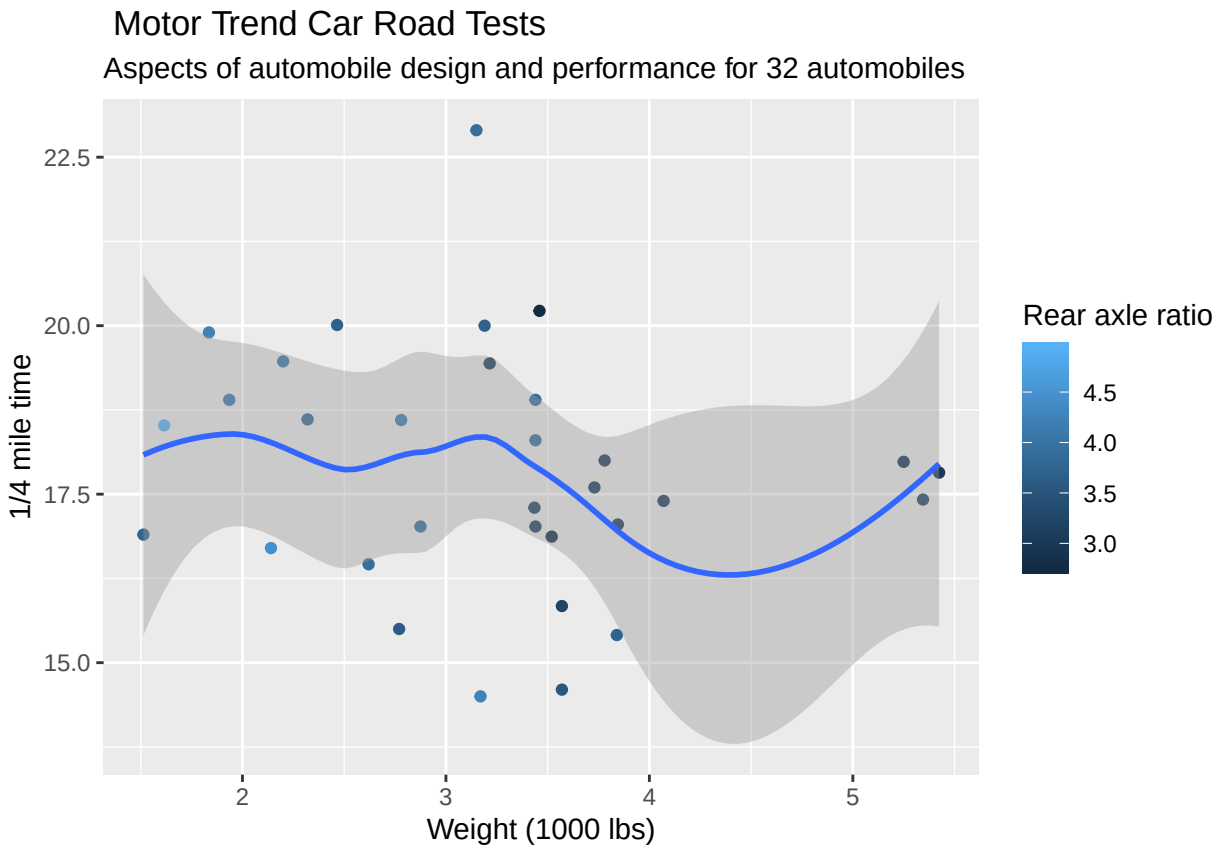
Labels

Another big part of creating a plot is in making sure it has reader-friendly labels. The ggplot2 package automatically assigns the name of the variable corresponding to the different components on the plot as the initial label. Code variable names are unfortunately not always legible to outside readers with no context.

If you wish to customize your labels, you can add a labs() function call to your ggplot object. Inside the function call to labs() you can provide new labels for the x and y axes as well as a title, subtitle, or caption. You can check out the list of available label arguments in the labs() documentation here.

The following labs() function call and these specified arguments would render the following plot:

```
plot1 <- ggplot(data=mtcars, aes(x=wt, y=qsec))
plot1 +geom_point(aes(color=drat)) +
       geom_smooth() +
      labs(title=" Motor Trend Car Road Tests", subtitle="Aspects of automobile design and performance
```



Extending The Grammar

We'll use now the Dataframe 'mpg' from the base library in R. The mpg dataset in R is a built-in dataset describing fuel economy data from 1999 and 2008 for 38 popular models of cars and is included with ggplot.

```
data(mpg)
```

We've gone over each of the basic units in the grammar of graphics: data, geometries, and aesthetics. Let's extend this new knowledge to create a new type of plot: the bar chart. See the labs() documentation here.

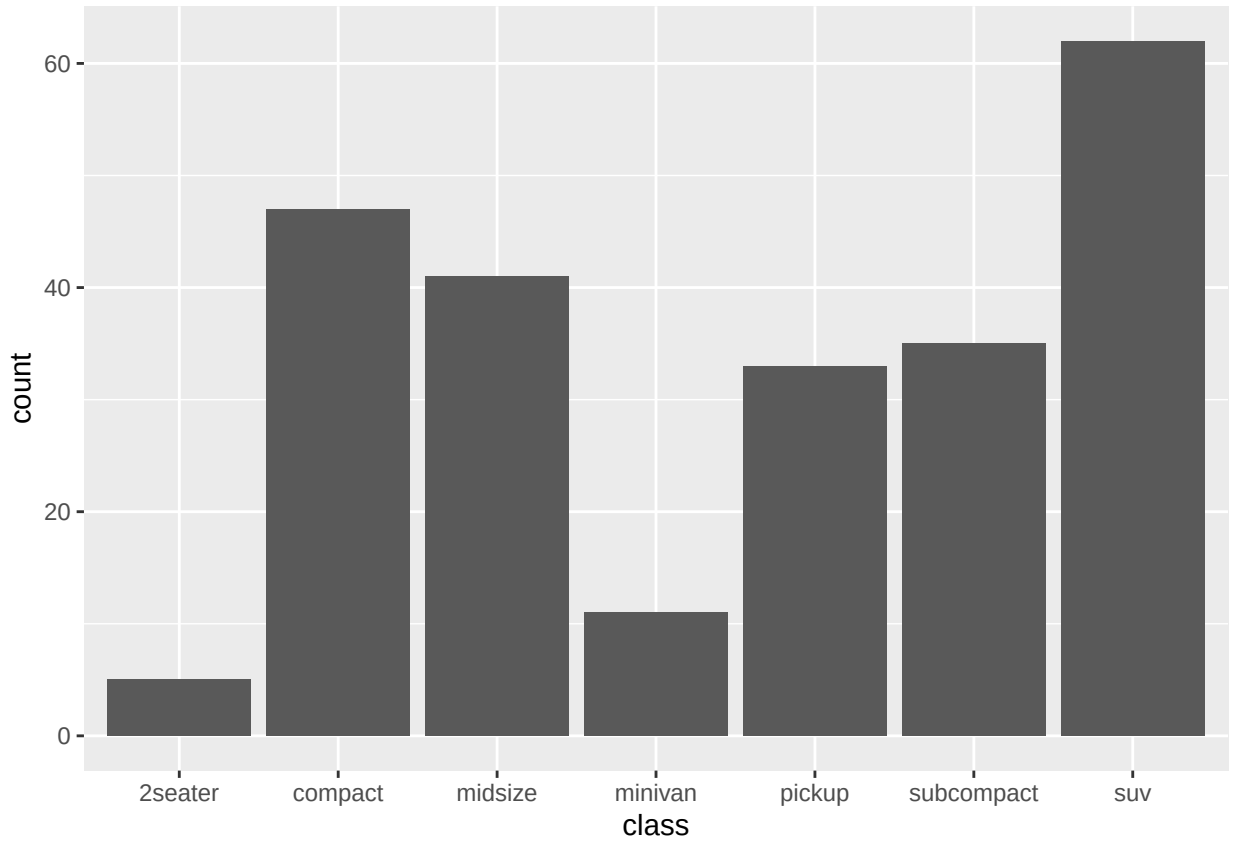
Bar charts are great for showing the distribution of categorical data. Typically, one of the axes on a bar chart will have numerical values and the other will have the names of the different categories you wish to understand.

The geom_bar() layer adds a bar chart to the canvas. Typically when creating a bar chart, you assign an aes() aesthetic mapping with a single categorical value on the x axes and the aes() function will compute the count for each category and display the count values on the y axis.

Since we're extending the grammar of graphics, let's also learn about how to save our visuals as local image files..

The following code maps the count of each category in the "class" column in a dataset of 38 popular models of cars to a bar length and then saves the visualization as a .png file named "bar-example.png" on your directory:

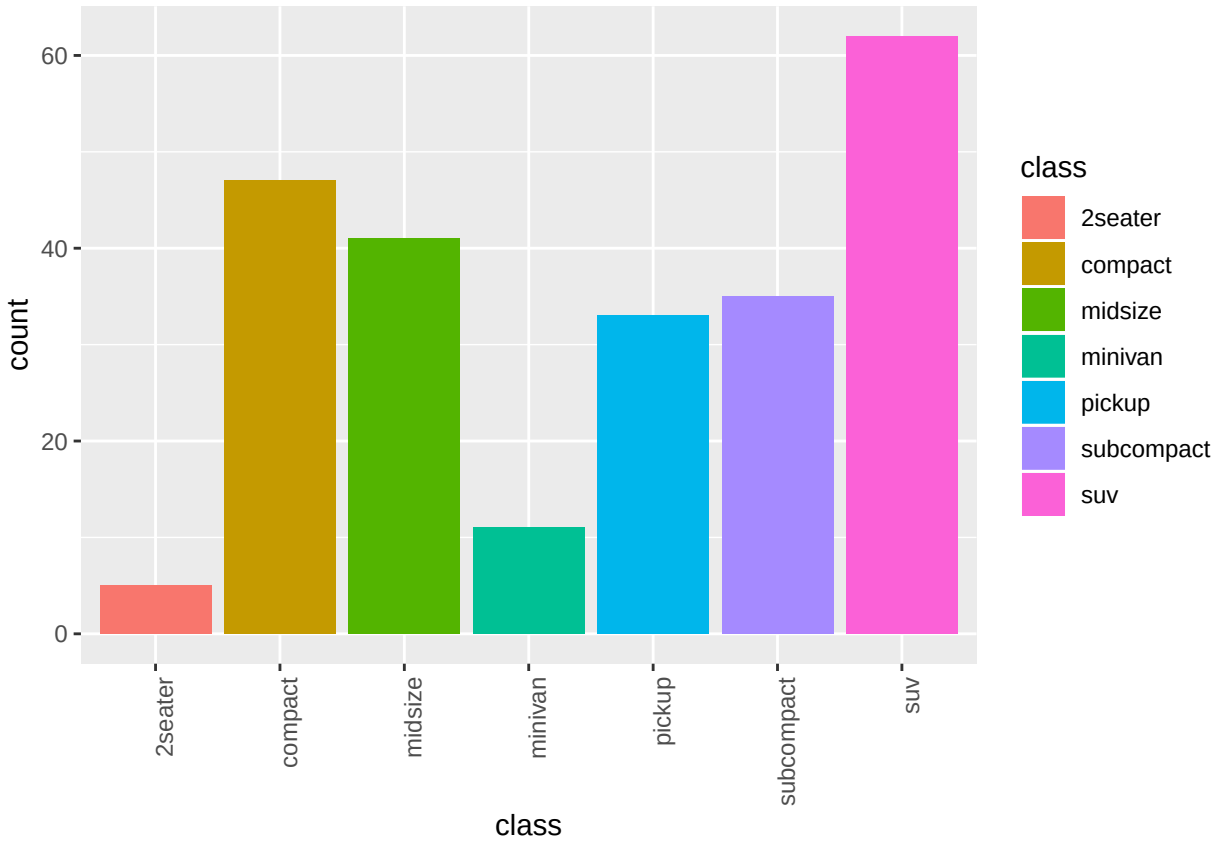
```
bar <- ggplot(mpg, aes(x=class)) + geom_bar()
bar
```




```
ggsave("bar-example.png")
```

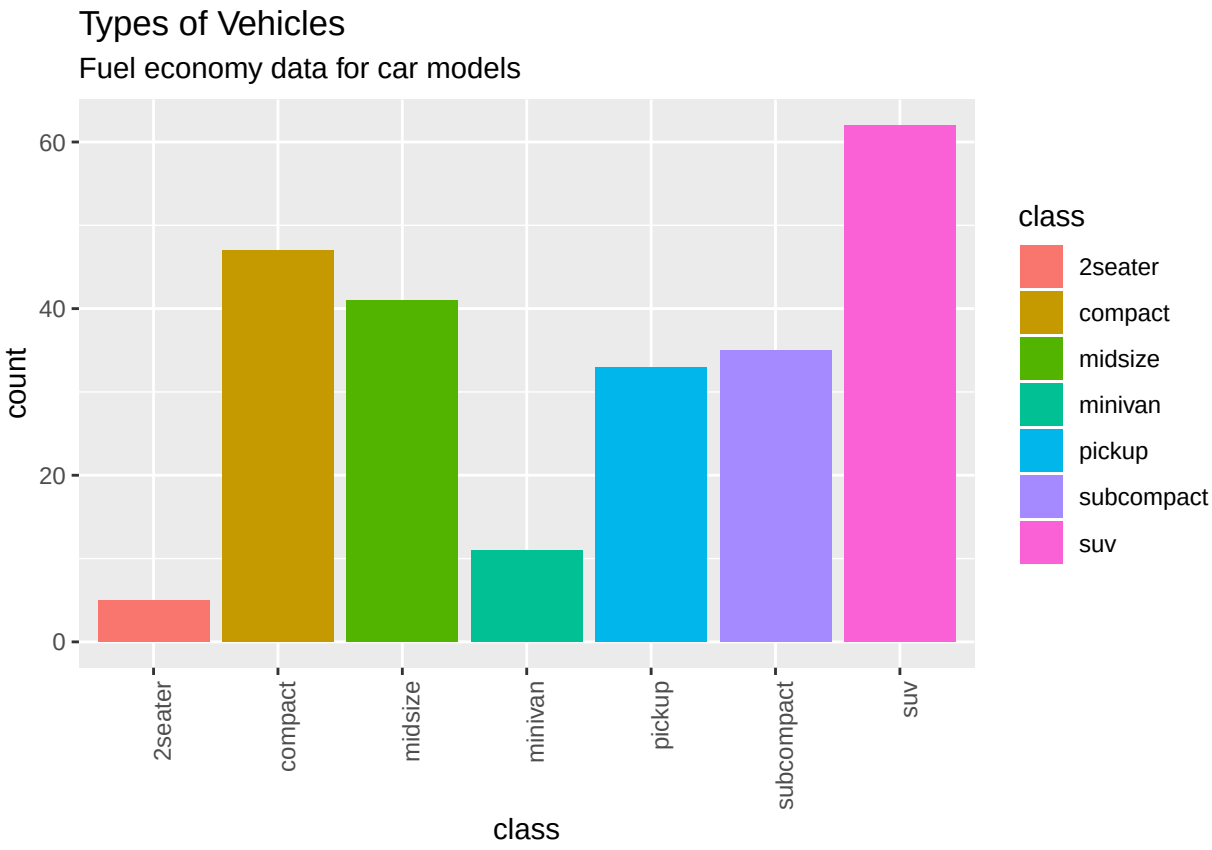
Let's add some color to the bar chart, by adding an aes() aesthetic mapping to the geom_bar() layer that fills the color of each bar based on the class value. Note: we use the theme() Geom function to rotate the labels.

```
bar <- ggplot(mpg, aes(x=class)) + geom_bar(aes(fill=class)) + theme(axis.text.x = element_text(angle =
bar
```



Our plot could use some context, let's add a title and a sub-title so that users can understand more about what we are displaying with this bar chart and the mpg dataset. Use the labs() function to assign a new title that describes this plot is illustrating the Types of Vehicles and a subtitle describing the data as From fuel economy data for popular car models (1999-2008)

```
bar <- ggplot(mpg, aes(x=class)) + geom_bar(aes(fill=class))+labs(title="Types of Vehicles",subtitle="Fu
bar
```



Review ggplot

General pattern for creating a visualization:

- Determine what relationship you wish to explore in your data
- Find the right geom(s) in the labs() ggplot2 documentation to display that relationship and read about the arguments and aesthetics specific to that geom
- Extend the grammar of graphics to follow the pattern learned in this lesson to add layers and create a visualization. Improve graph legibility by polishing labels and styles.

Aggregates in R

Calculating Column Statistics

In this exercise, you will learn how to combine all of the values from a column for a single calculation. This can be done with the help of the dplyr function summarize(), which returns a new data frame containing the desired calculation.

Examples:

- The data frame customers contains the names and ages of all of your customers. You want to find the median age:

```
#customers %>%
#  select(age)
# c(23, 25, 31, 35, 35, 46, 62)
#customers %>%
#  summarize(median_age = median(age))
```

```
# 35
```

- The data frame shipments contains address information for all shipments that you've sent out in the past year. You want to know how many different states you have shipped to.

```
#shipments %>%
#  select(states)
# c('CA', 'CA', 'CA', 'CA', 'NY', 'NY', 'NJ', 'NJ', 'NJ', 'NJ', 'NJ', 'NJ', 'NJ')
#shipments %>%
#  summarize(n_distinct_states = n_distinct(states))
# 3
```

- The data frame inventory contains a list of types of t-shirts that your company makes. You want to know the standard deviation of the prices of your inventory.

```
#inventory %>%
#  select(price)
# c(31, 23, 30, 27, 30, 22, 27, 22, 39, 27, 36)
#inventory %>%
#  summarize(sd_price = sd(price))
# 5.465595
```

The general syntax for these calculations is:

```
#df %>%
#  summarize(var_name = command(column_name))
```

- df is the data frame you are working with
- summarize is a dplyr function that reduces multiple values to a single value
- var_name is the name you assign to the column that stores the results of the summary function in the returned data frame
- command is the summary function that is applied to the column by summarize()
- column_name is the name of the column of df that is being summarized

The following table includes common summary functions that can be given as an argument to summarize():

Command	Description
mean()	Average of all values in column
median()	Median value of column
sd()	Standard deviation of column
var()	Variance of column
min()	Minimum value in column
max()	Maximum value in column
IQR()	Interquartile range of column
n_distinct()	Number of unique values in column
sum()	Sum values of column

In the folowing examples we're using **mpg** dataset.

```
head(mpg,3)
```

```
## # A tibble: 3 x 11
##   manufacturer model displ  year   cyl trans      drv     cty   hwy fl    class
##   <chr>        <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>
## 1 audi         a4      1.8  1999     4 auto(l5)   f        18    29 p     compa~
## 2 audi         a4      1.8  1999     4 manual(m5) f        21    29 p     compa~
```

```
## 3 audi          a4      2    2008     4 manual(m6) f        20    31 p     compa~
```

In order to know, for example the largest highway miles per gallon (hwy) Note: We add the command *na.rm = TRUE* to avoid NA missing values from our dataset.

```
mpg%>% summarize(max(hwy,na.rm = T))
```

```
## # A tibble: 1 x 1
##   `max(hwy, na.rm = T)`
##                   <int>
## 1                    44
```

If we want to know how many differents manufacturers we have in our dataset.

```
mpg%>% summarize(n_distinct(manufacturer,na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   `n_distinct(manufacturer, na.rm = TRUE)`
##                                      <int>
## 1                                       15
```

Calculating Aggregate Functions I

When we have a bunch of data, we often want to calculate aggregate statistics (mean, standard deviation, median, percentiles, etc.) over certain subsets of the data. Suppose we have a grade book with columns student, assignment_name, and grade:

student	assignment_name	grade
Amy	Assignment 1	96
Amy	Assignment 2	87
Bob	Assignment 1	91
Bob	Assignment 2	75
Chris	Assignment 1	83
Chris	Assignment 2	88

We want to get an average grade for each student across all assignments. We can do this using the helpful dplyr function group_by().

```
#grades <- df %>%
#  group_by(student) %>%
#  summarize(mean_grade = mean(grade)) or weighted.mean(variable,factor)
```

student	mean_grade
Amy	91.5
Bob	83
Chris	85.5

In general, we use the following syntax to calculate aggregates:

```
#df %>%
#  group_by(column_1) %>%
#  summarize(aggregate_name = command(column_2))
```

- column_1 (student in our example) is the column that we want to group_by()

- column_2 (grade in our example) is the column that we want to apply command(), a summary function, to using summarize()
- aggregate_name is the name assigned to the calculated aggregate

In addition to the summary functions discussed in the last exercise **(mean(), median(), sd(), var(), min(), max(), IQR() and n_distinct())**, another helpful summary function, especially for grouped data, is **n()**. n() will return the count of the rows within a group, and does not require a column as an argument.

In the folowing examples we're using **mpg** dataset.

Now, they want to know the highest highway miles per gallon (hwy) for car class (class):

```
data(mpg)
hwy_cars <- mpg %>% group_by(class)  %>% summarize(max_hwy=max(hwy, na.rm = TRUE))
hwy_cars
```

```
## # A tibble: 7 x 2
##   class      max_hwy
##   <chr>        <int>
## 1 2seater         26
## 2 compact         44
## 3 midsize         32
## 4 minivan         24
## 5 pickup          22
## 6 subcompact      44
## 7 suv             27
```

If we want to know how many cars per class:

```
count_class <- mpg %>% group_by(class)  %>% summarize(count=n())
```

Calculating Aggregate Functions II

Sometimes, we want to group by more than one column. We can do this by passing multiple column names as arguments to the **group_by** function.

Imagine that we run a chain of stores and have data about the number of sales at different locations on different days:

location	date	day_of_week	total_sales
West Village	February 1	W	400
West Village	February 2	Th	450
Chelsea	February 1	W	375
Chelsea	February 2	Th	390
...	...	...	...

We suspect that sales are different at different locations on different days of the week. In order to test this hypothesis, we could calculate the average sales for each store on each day of the week across multiple months. The code would look like this:

```
#df %>%
#  group_by(location,day_of_week) %>%
#  summarize(mean_total_sales = mean(total_sales))
```

And the results might look something like this:

location	day_of_week	mean_total_sales
Chelsea	M	402.50
Chelsea	Tu	422.75
Chelsea	W	452.00
...	...	...
West Village	M	390
West Village	Tu	400
...	...	...

Example, let's say we want to group by manufacturer and class and see which combination has most the highway miles per gallon (hwy), let's orther them from max hwy to min hwy.

```
hwy_cars <- mpg %>% group_by(manufacturer,class)  %>% summarize(max_hwy=max(hwy, na.rm = TRUE)) %>% arra
hwy_cars
```

```
## # A tibble: 32 x 3
## # Groups:   manufacturer [15]
##    manufacturer class      max_hwy
##    <chr>        <chr>        <int>
##  1 volkswagen   compact         44
##  2 volkswagen   subcompact      44
##  3 toyota       compact         37
##  4 honda        subcompact      36
##  5 nissan       midsize         32
##  6 audi         compact         31
##  7 hyundai      midsize         31
##  8 toyota       midsize         31
##  9 chevrolet    midsize         30
## 10 hyundai      subcompact      29
## # ... with 22 more rows
```

Combining Grouping with Filter

While group_by() is most often used with summarize() to calculate summary statistics, it can also be used with the dplyr function filter() to filter rows of a data frame based on per-group metrics.

Suppose you work at an educational technology company that offers online courses and collects user data in an enrollments data frame:

user_id	course	quiz_score
1234	learn_r	80
1234	learn_python	95
4567	learn_r 90	
4567	learn_python	55
...	...	...

You want to identify all the enrollments in difficult courses, which you define as courses with an average quiz_score less than 80. To filter the data frame to just these rows:

```
#enrollments %>%
#  group_by(course) %>%
#  filter(mean(quiz_score) < 80)
```

- group_by() groups the data frame by course into two groups: learn-r and learn-python
- filter() will keep all the rows of the data frame whose per-group (per-course) average quiz_score is less than 80

Rather than filtering rows by the individual column values, the rows will be filtered by their group value since a summary function is used! The resulting data frame would look like this:

user_id	course	quiz_score
1234	learn_python	95
4567	learn_python	55

- The average quiz_score for the learn-r course is 85, so all the rows of enrollments with a value of learn-r in the course column are filtered out.
- The average quiz_score for the learn-python course is 75, so all the rows of enrollments with a value of learn-python in the course column remain.

Example with mpg let's say we want to find all the manufacturers who's average highway miles per gallon (hwy) is larger than 30, so we group by manufacturer and filter with mean(hwy) > 30, notice we are taking the average hwy by manufacturer, not in the whole dataset.

```
hwy_bymanufacturer <- mpg %>% group_by(manufacturer)  %>% filter(mean(hwy) > 30)
hwy_bymanufacturer
```

```
## # A tibble: 9 x 11
## # Groups:   manufacturer [1]
##   manufacturer model displ  year   cyl trans    drv     cty   hwy fl    class
##   <chr>        <chr> <dbl> <int> <int> <chr>    <chr> <int> <int> <chr> <chr>
## 1 honda        civic   1.6  1999     4 manual(~ f        28    33 r     subcomp~
## 2 honda        civic   1.6  1999     4 auto(14) f        24    32 r     subcomp~
## 3 honda        civic   1.6  1999     4 manual(~ f        25    32 r     subcomp~
## 4 honda        civic   1.6  1999     4 manual(~ f        23    29 p     subcomp~
## 5 honda        civic   1.6  1999     4 auto(14) f        24    32 r     subcomp~
## 6 honda        civic   1.8  2008     4 manual(~ f        26    34 r     subcomp~
## 7 honda        civic   1.8  2008     4 auto(15) f        25    36 r     subcomp~
## 8 honda        civic   1.8  2008     4 auto(15) f        24    36 c     subcomp~
## 9 honda        civic   2    2008     4 manual(~ f        21    29 p     subcomp~
```

We can see only the Honda Civic cars fulfill these requirement, we can further count them using a summarise() pipe.

```
hwy_bymanufacturer <- mpg %>% group_by(manufacturer)  %>% filter(mean(hwy) > 30) %>% summarise(count=n()
hwy_bymanufacturer
```

```
## # A tibble: 1 x 2
##   manufacturer count
##   <chr>        <int>
## 1 honda            9
```

Combining Grouping with Mutate

group_by() can also be used with the dplyr function mutate() to add columns to a data frame that involve per-group metrics. Consider the same educational technology company's enrollments table from the previous exercise:

user_id	course	quiz_score
1234	learn_r	80

user_id	course	quiz_score
1234	learn_python	95
4567	learn_r 90	
4567	learn_python	55
...	...	...

You want to add a new column to the data frame that stores the difference between a row's quiz_score and the average quiz_score for that row's course. To add the column:

```
#enrollments %>%
#  group_by(course) %>%
#  mutate(diff_from_course_mean = quiz_score - mean(quiz_score))
```

- group_by() groups the data frame by course into two groups: learn-r and learn-python
- mutate() will add a new column diff_from_course_mean which is calculated as the difference between a row's individual quiz_score and the mean(quiz_score) for that row's group (course)

The resulting data frame would look like this:

user_id	course	quiz_score	diff_from_course_mean
1234	learn_r	80	-5
1234	learn_python	95	20
4567	learn_r	90 5	
4567	learn_python	55	-20
...	...	...	...

- The average quiz_score for the learn-r course is 85, so diff_from_course_mean is calculated as quiz_score - 85 for all the rows of enrollments with a value of learn-r in the course column.
- The average quiz_score for the learn-python course is 75, so diff_from_course_mean is calculated as quiz_score - 75 for all the rows of enrollments with a value of learn-python in the course column.

Joining tables in R

Introduction

In order to efficiently store data, we often spread related information across multiple tables. For instance, imagine that we own an e-commerce business and we want to track the products that have been ordered from our website. We could have one table with all of the following information: order_id, customer_id, customer_name, customer_address, customer_phone_number ,product_id, product_description, product_price, quantity, timestamp.

However, a lot of this information would be repeated. If the same customer makes multiple orders, that customer's name, address, and phone number will be reported multiple times. If the same product is ordered by multiple customers, then the product price and description will be repeated. This will make our orders table big and unmanageable.

So instead, we can split our data into three tables:

- orders would contain the information necessary to describe an order: order_id, customer_id, product_id, quantity, and timestamp
- products would contain the information to describe each product: product_id, product_description and product_price
- customers would contain the information for each customer: customer_id, customer_name, customer_address, and customer_phone_number

Joining

Let's say we have information relevant for Table1 in a column of Table2, so we would like to bring that information into Table1. More so if all the information in Table2 is relevant. In order to make this join of information, we need to have common elements accros the tables.

Let's say we have three tables: Orders (id:1,2,3...), Customers (id:1,2,3...), Products (id:1,2,3...) and each Table row contains information describing the Ordern, the Customer and the Product associated with each id.

Let say we want to know more about Order1, so if we look at the Table order we see that Order:1 is from Customer:3 for the Product:2, so in order to know more about each one of these variable, we need to go to each corresponding table and look for the corresponding row. This would take a lot of time.

Inner Join I

It is easy to do this kind of matching for one row, but hard to do it for multiple rows. Luckily, dplyr can efficiently do this for the entire table using the inner_join() method.

The inner_join() method looks for columns that are common between two data frames and then looks for rows where those columns' values are the same. It then combines the matching rows into a single row in a new table. We can call the inner_join() method with two data frames like this:

```
#joined_df <- orders %>%
#  inner_join(customers)
```

This will match up all of the customer information to the orders that each customer made.

Example: We have the following table "sales" which contains the monthly revenue. It has two columns: month and revenue.

month	revenue
January	300
February	290
March	310
April	325
May	475
June	495
...	...

We also have the table "targets" which contains the goals for monthly revenue for each month. It has two columns: month and target.

month	targets
January	310
February	270
March	300
April	350
May	475
June	500
...	...

So we could use the inner_join() function to join these two tables by the month column into a new dataframe called "sales_vs_targets".

```
#sales_vs_targets <- sales %>% inner_join(targets)
```

month	revenue	targets
January	300	310
February	290	270
March	310	300
April	325	350
May	475	475
June	495	500
...	...	...

Notice the funtion automatically joined by = "month", because is the only common column among the tables. What if there are many common columns? We'll se this in the following section "Join on Specific Columns I"

Inner Join II

In addition to using inner_join() to join two data frames together, we can use the pipe %>% to join multiple data frames together at once. Going back into our three tables example from previus section, the following command would join orders with customers, and then join the resulting data frame with products:

```
#big_df <- orders %>%
#  inner_join(customers) %>%
#  inner_join(products)
```

Join on Specific Columns I

In the previous example, the inner_join() function "knew" how to combine tables based on the columns that were the same between two tables. For instance, orders and customers both had a column called customer_id. This won't always be true when we want to perform a join.

Generally, the orders data frame would not have the column order_id and the customers data frame would not have the column customer_id. Instead, they would both have a column id and it would be implied that the id was the order_id for the orders table and the customer_id for the customers table. So if we use inner_join() the common column id won't we usefull for the pairing.Because the id columns would mean something different in each table, our default joins would be wrong.

One way that we could address this problem is to use the dplyr function rename() to rename the columns for our joins. In the example, we will rename the column id in the customers data frame to customer_id (recall that te column customer_id already exist in the Orders table), so that orders and customers now have a common column to join on.

```
#customers <- customers %>%
#  rename(customer_id = id)
#inner_join(orders, customers)
```

Join on Specific Columns II

Exist another option better than rename() columns to fit the inner_join(). We can add the **by** argument when calling inner_join() to specify which columns we want to join on. In the example below, the "left" table is the one that comes first (orders), and the "right" table is the one that comes second (customers). This syntax says that we should match the customer_id from orders to the id in customers.

```
#orders %>%
#  inner_join(customers,
#             by = c('customer_id' = 'id'))
```

If we use this syntax, we'll end up with two columns called id, one from the first table and one from the second. R won't let you have two columns with the same name, so it will change them to id_x and id_y. The new column names id_x and id_y aren't very helpful for us when we read the table. We can help make them more useful by using the keyword suffix. We can provide a vector of suffixes to use instead of "_x" and "_y". We could use the following code to make the suffixes reflect the table names:

```
#orders %>%
#  inner_join(customers,
#             by = c('customer_id' = 'id'),
#             suffix = c('_order','_customer'))
```

More generally

```
#joined_dfs <- df_1 %>%
#  inner_join(df_2,
#             by = c('id from df_1' = 'other_id from df_2'),
#             suffix = c('_suffix1','_suffix2'))
```

Mismatched Joins

In our previous examples, there were always matching values when we were performing our joins. What happens when that isn't true? If we have a row in df_1 associated with a value that does not exist in df_2, the inner_join() will drop the conflicted row. This also applies with multiple rows.

Full Join

In the previous exercise, we saw that when we join two data frames whose rows don't match perfectly, we lose the unmatched rows. This type of join (where we only include matching rows) is called an inner join. There are other types of joins that we can use when we want to keep information from the unmatched rows.

Suppose that two companies, Company A and Company B have just merged. They each have a list of customers, but they keep slightly different data. Company A has each customer's name and email. Company B has each customer's name and phone number. They have some customers in common, but some are different.

If we wanted to combine the data from both companies without losing the customers who are missing from one of the tables, we could use a Full Join. A Full Join would include all rows from both tables, even if they don't match. Any missing values are filled in with NA.

Company_a

name	email
Sally	Sparrow sally.sparrow@gmail.com
Peter	Grant pgrant@yahoo.com
Leslie	May leslie_may@gmail.com

Company_b name| phone ——| —— Peter Grant |212-345-6789 Leslie May| 626-987-6543 Aaron Burr| 303-456-7891

```
#full_joined_dfs <- company_a %>%
#  full_join(company_b)
```

full_joined_dfs name |email| phone ——| —— | —— Sally Sparrow |sally.sparrow@gmail.com| NA Peter Grant| pgrant@yahoo.com| 212-345-6789 Leslie May| leslie_may@gmail.com| 626-987-6543 Aaron Burr |NA| 303-456-7891

This automatically joins both dfs by the common column with some common elements ("name")

Left and Right Joins

Let's return to the join of Company a and Company b.

Left Join

Suppose we want to identify which customers are missing phone information. We would want a list of all customers who have email, but don't have phone. We could get this by performing a Left Join. A Left Join includes all rows from the first (left) table, but only rows from the second (right) table that match the first table.

For this command, the order of the arguments matters. If the first data frame is company_a and we do a left join, we'll only end up with rows that appear in company_a. By listing company_a first, we get all customers from Company A, and only customers from Company B who are also customers of Company A.

```
#left_joined_df <- company_a %>%
#  left_join(company_b)
```

left_joined_df

name	email	phone
Sally Sparrow	sally.sparrow@gmail.com	NA
Peter Grant	pgrant@yahoo.com	212-345-6789
Leslie May	leslie_may@gmail.com	626-987-6543

Right Join

Now let's say we want a list of all customers who have phone but no email. We can do this by performing a Right Join. A Right Join is the exact opposite of left join. Here, the joined table will include all rows from the second (right) table, but only rows from the first (left) table that match the second table. By listing company_a first and company_b second, we get all customers from Company B, and only customers from Company A who are also customers of Company B.

```
#right_joined_df <- company_a %>%
#  right_join(company_b)
```

right_joined_df

name	email	phone
Aaron Burr	NA	303-456-7891
Peter Grant	pgrant@yahoo.com	212-345-6789
Leslie May	leslie_may@gmail.com	626-987-6543

Notice that we could simply revert the order using a left_join() and get the same result.

Concatenate Data Frames

Sometimes, a dataset is broken into multiple tables. For instance, data is often split into multiple CSV files so that each download is smaller. When we need to reconstruct a single data frame from multiple smaller

data frames, we can use the dplyr bind_rows() method. This method **only works if all of the columns are the same in all of the data frames.**

```
#concatenated_dfs <- df1 %>%
#  bind_rows(df_2)
```

Review

- Creating a data frame made by matching the common columns of two data frames is called a join
- We can specify which columns should be matched by using the by argument
- We can combine data frames whose rows don't all match using left, right, and full joins
- We can stack or concatenate data frames with the same columns using bind_rows()

Central tendency measures

Mean

Finding the center of a dataset is one of the most common ways to summarize statistical findings. Often, people communicate the center of data using words like, on average, usually, or often. The mean, often referred to as the average, is a way to measure the center of a dataset.

The average of a set is calculated using a two-step process:

- Add all of the observations in your dataset.
- Divide the total sum from step one by the number of points in your dataset.

$$\bar{x} = \frac{x_1 + x_2 + ... + x_n}{n}$$

Where $x_1, x_2, \ldots x_n$ are observations from a dataset of n observations.

Example. Imagine that we wanted to calculate average of a dataset with the following four observations:

```
numbers_ex <- c(4, 6, 2, 8)
sum_numbers_ex <- 4+ 6+ 2+ 8
cat("The mean value is:", sum_numbers_ex / 4)
```

```
## The mean value is: 5
```

Faster:

```
numbers_ex <- c(4, 6, 2, 8)
sum_numbers_ex <- sum(numbers_ex)
cat("The mean value is:", sum_numbers_ex / 4)
```

```
## The mean value is: 5
```

Even faster:

```
cat("The mean value is:", mean(numbers_ex))
```

```
## The mean value is: 5
```

The R mean() function can do the work of adding and dividing for you. In the example below, we use mean() to calculate the average of a dataset with ten values:

```
example_data <- c(24, 16, 30, 10, 12, 28, 38, 2, 4, 36)

example_average <- mean(example_data)

cat("The example average is:", example_average)
```

```
## The example average is: 20
```

The most important outcome is that we're able to use a single number as a measure of centrality. Although histograms provide more information, they are not a concise or precise measure of centrality — you must interpret it for yourself.

Notice we use the cat() function. cat() is valid only for atomic types (logical, integer, real, complex, character) and names. It means you cannot call cat on a non-empty list or any type of object. In practice it simply converts arguments to characters and concatenates so you can think of something like as.character() %>% paste().

Median

You can think of the median as being the observation in your dataset that falls right in the middle. The formal definition for the median of a dataset is:

The value that, assuming the dataset is ordered from smallest to largest, falls in the middle. If there are an even number of values in a dataset, you either report both of the middle two values or their average.

There are always two steps to finding the median of a dataset:

- Order the values in the dataset from smallest to largest
- Identify the number(s) that fall(s) in the middle

Example: Even Number of Values

Say we have a dataset with the following ten numbers:

$$24, 16, 30, 10, 12, 28, 38, 2, 4, 36$$

The first step is to order these numbers from smallest to largest:

$$2, 4, 10, 12, [16, 24], 28, 30, 36, 38$$

Because this dataset has an even number of values, there are two medians: 16 and 24 — 16 has four datapoints to the left, and 24 has four datapoints to the right. Although you can report both values as the median, people often average them. If you averaged 16 and 24, you could report the median as 20.

Example: Odd Number of Values

If we added another value (say, 24) to the dataset and sorted it, we would have:

$$2, 4, 10, 12, 16, [24], 24, 28, 30, 36, 38$$

The new median is equal to 24, because there are 5 values to the left of it, and 5 values to the right of it.

Finding the median of a dataset becomes increasingly time-consuming as the size of your dataset increases — imagine finding the median of an unsorted dataset with 10,000 observations. The R median() function can do the work of sorting, then finding the median for you. In the example below, we use median() to calculate the median of a dataset with ten values:

```
example_data = c(24, 16, 30, 10, 12, 28, 38, 2, 4, 36, 42)

example_median = median(example_data)

cat("The example mdeian is:", example_median)
```

```
## The example mdeian is: 24
```

The code above prints the median of the dataset, 24. The mean of this dataset is 22. **It's worth noting these two values are close to one another, but not equal.**

Notice that the mean and the median are nearly equal. This is not a surprising result, as both statistics are a measure of the dataset's center. However, it's worth noting that these results will not always be so close.

Mode

The formal definition for the mode of a dataset is:

The most frequently occurring observation in the dataset. A dataset can have multiple modes if there is more than one value with the same maximum frequency.

While you may be able to find the mode of a small dataset by simply looking through it, if you have trouble, we recommend you follow these two steps:

- Find the frequency of every unique number in the dataset
- Determine which number has the highest frequency

Example Say we have a dataset with the following ten numbers:

$$24, 16, 12, 10, 12, 28, 38, 12, 28, 24$$

Let's find the frequency of each number:

24	16	12	10	28	38
2	1	3	1	2	1

From the table, we can see that our mode is 12, the most frequent number in our dataset.

Finding the mode of a dataset becomes increasingly time-consuming as the size of your dataset increases — imagine finding the mode of a dataset with 10,000 observations.

The R package DescTools includes a handy Mode() function which can do the work of finding the mode for us. In the example below, we use Mode() to calculate the mode of a dataset with ten values:

Notice the function needs a **Capital "M"**

Example: One Mode

```
library(DescTools)
example_data <- c(24, 16, 12, 10, 12, 28, 38, 12, 28, 24)
example_mode <- Mode(example_data)
example_mode
```

```
## [1] 12
## attr(,"freq")
## [1] 3
```

The code above calculates the mode of the values in example_data and saves it to example_mode. The result of Mode() is a vector with the mode value:

Example: Two Mode

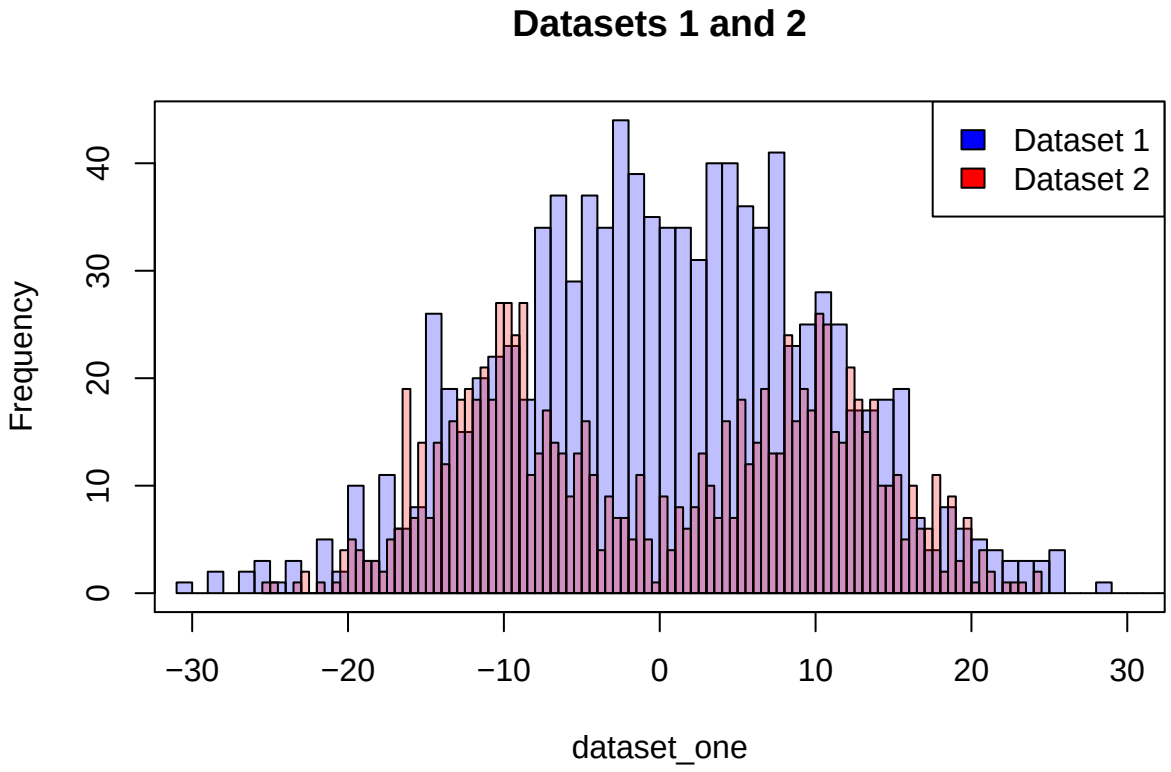
If there are multiple modes, the Mode() function will return them as a vector. Let's look at a vector with two modes, 12 and 24:

```
example_data = c(24, 16, 12, 10, 12, 24, 38, 12, 28, 24)
example_mode = Mode(example_data)
example_mode
```

```
## [1] 12 24
## attr(,"freq")
## [1] 3
```

Example: We have to Data sets and print the mean and variance

```
hist(dataset_one, col=rgb(0,0,1,1/4),xlim=c(-30,30), main="Datasets 1 and 2",breaks=80)
hist(dataset_two,  col=rgb(1,0,0,1/4), add=T, breaks=80)
legend("topright", c("Dataset 1", "Dataset 2"), fill=c("blue", "red"))
box()
```




```
print(paste("The mean of the first dataset is ", mean(dataset_one)))
```

```
## [1] "The mean of the first dataset is  0.3133598953587"
```

```
print(paste("The mean of the second dataset is ",mean(dataset_two)))
```

```
## [1] "The mean of the second dataset is  0.05276583677259"
```

Variance

Distance From Mean

Now that you have learned the importance of describing the spread of a dataset, let's figure out how to mathematically compute this number. How would you attempt to capture the spread of the data in a single number?

Let's start with our intuition — we want the variance of a dataset to be a large number if the data is spread out, and a small number if the data is close together. A lot of people may initially consider using the range of the data. But that only considers two points in your entire dataset. Instead, we can include every point in our calculation by finding the difference between every data point and the mean.

If the data is close together, then each data point will tend to be close to the mean, and the difference will

be small. If the data is spread out, the difference between every data point and the mean will be larger. Mathematically, we can write this comparison as:

$$difference = x - \mu$$

Where x is a single data point and the Greek letter mu μ is the mean.

Average Distances

Once we have all our differences, we can sum them and divide between the number of observations in order to obtain an average distance:

$$\sum_{i}^{n} \frac{1}{n} difference_i = \sum_{i}^{n} \frac{1}{n} x_i - \mu$$

Do you think this sum accurately captures the spread of your data?

Square the Differences

This summ of differences has a little problem. Consider this very small dataset: c(-5, 5)

The mean of this dataset is 0, so when we find the difference between each point and the mean we get -5 - 0 = -5 and 5 - 0 = 5. When we take the average of -5 and 5 to get the variance, we get 0: $\frac{-5+5}{2} = 0$

Now think about what would happen if the dataset were c(-200, 200). We'd get the same result! That can't possibly be right — the dataset with 200 is much more spread out than the dataset with 5, so the variance should be much larger!

The problem here is with negative numbers. Because one of our data points was 5 units below the mean and the other was 5 units above the mean, they canceled each other out!

When calculating variance, if a data point was above or below the mean — all we care about is how far away it was. To get rid of those pesky negative numbers, we'll square the difference between each data point and the mean.

Our equation for finding the difference between a data point and the mean now looks like this:

$$\sum_{i}^{n} \frac{1}{n} (difference_i)^2 = \sum_{i}^{n} \frac{1}{n} (x_i - \mu)^2$$

Now we have the variance of our data set. The full equation for the variance is as follows:

$$\sigma^2 = \frac{\sum_{i}^{n} (x_i - \mu)^2}{n}$$

Let's dissect this equation a bit.

- Variance is usually represented by the symbol sigma squared.
- We start by taking every point in the dataset — from point number 1 to point number n — and finding the difference between that point and the mean.
- Next, we square each difference to make all differences positive.
- Finally, we average those squared differences by adding them together and dividing by n, the total number of points in the dataset.

All of this work can be done quickly using a function we provided. The **var()** function takes a list of numbers as a parameter and returns the variance of that dataset.

```r
dataset <- c(3, 5, -2, 49, 10)
var <- var(dataset)
cat("The example variance is:", var)
```

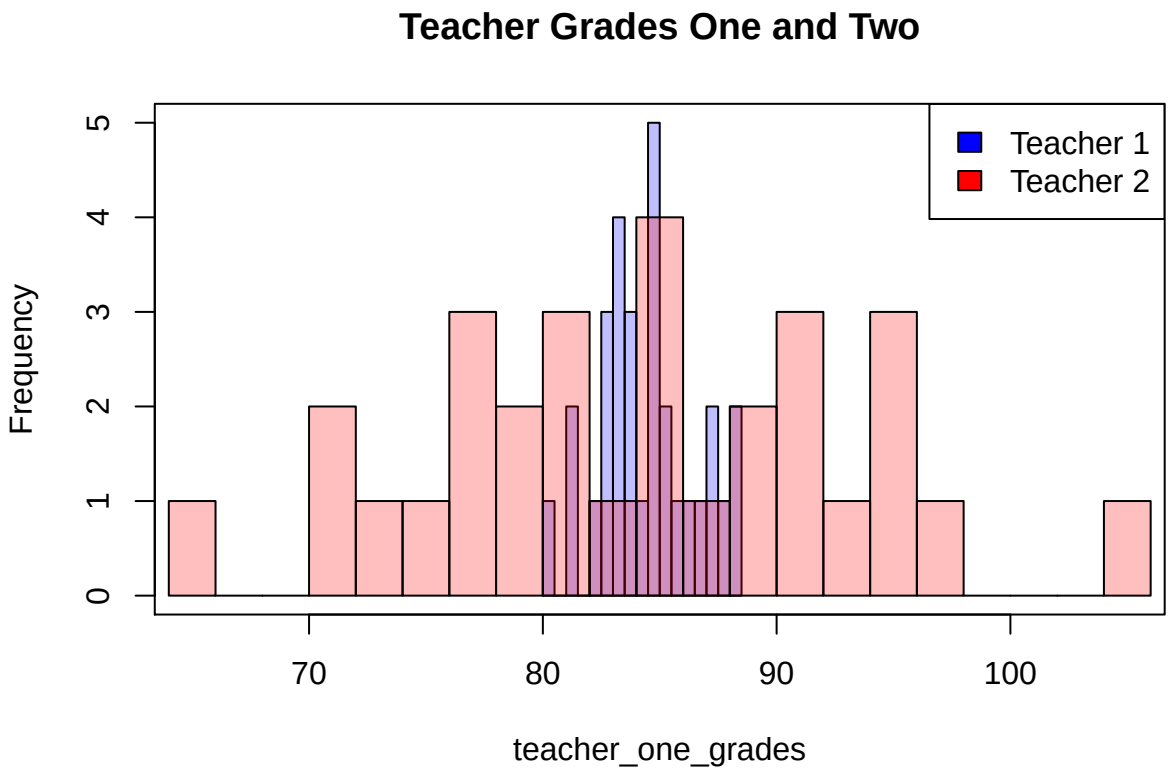
```
## The example variance is: 423.5
```

Example

Consider two professors in a school with a set of grades from their students.

```r
teacher_one_variance <- var(teacher_one_grades)
teacher_two_variance <- var(teacher_two_grades)
```

```r
#IGNORE THE CODE BELOW HERE
hist(teacher_one_grades, col=rgb(0,0,1,1/4),xlim=c(65,105), main="Teacher Grades One and Two", breaks=15
hist(teacher_two_grades,  col=rgb(1,0,0,1/4), add=T, breaks=15)
legend("topright", c("Teacher 1", "Teacher 2"), fill=c("blue", "red"))
box()
```




```r
print(paste("The mean of the test scores in teacher one's class is ", mean(teacher_one_grades)))
```

```
## [1] "The mean of the test scores in teacher one's class is  84.4676666666667"
```

```r
print(paste("The mean of the test scores in teacher two's class is ", mean(teacher_two_grades)))
```

```
## [1] "The mean of the test scores in teacher two's class is  84.298"
```

```r
print(paste("The variance of the test scores in teacher one's class is ", teacher_one_variance))
```

```
## [1] "The variance of the test scores in teacher one's class is  4.4136391954023"
```

```r
print(paste("The variance of the test scores in teacher two's class is ", teacher_two_variance))
```

```
## [1] "The variance of the test scores in teacher two's class is  80.826195862069"
```

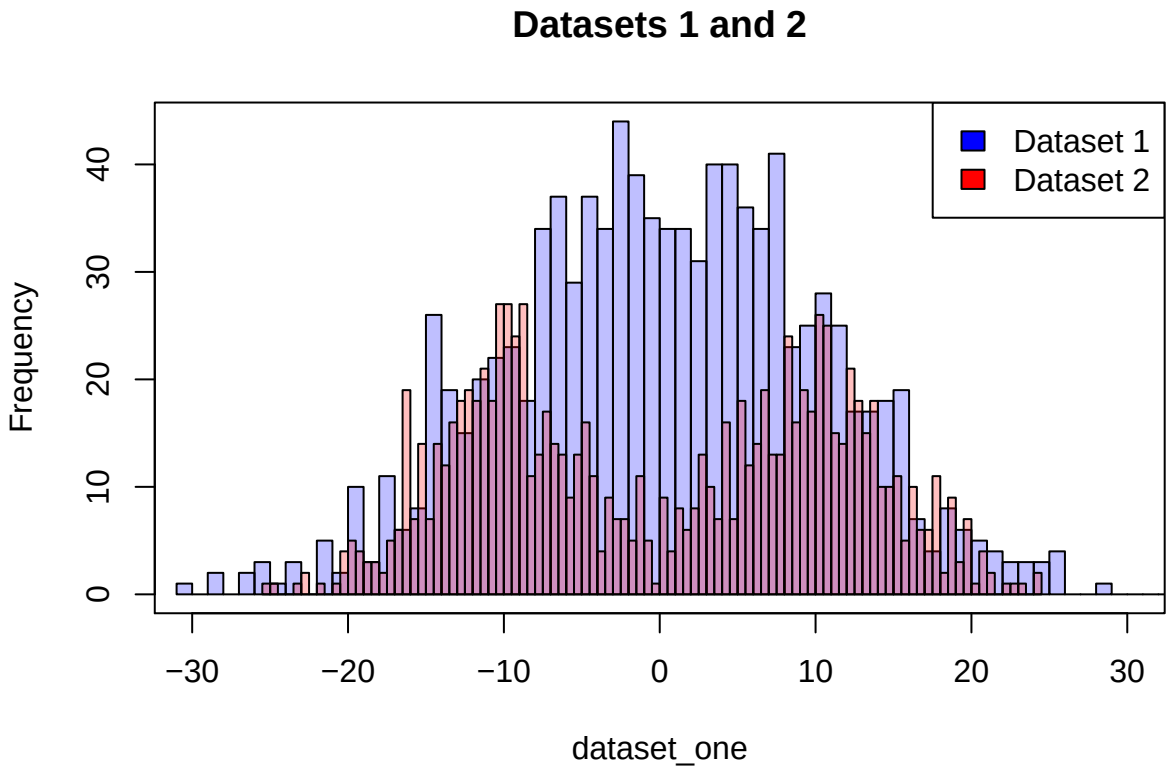
What story does variance tell? What conclusions can we draw from this statistic? In the class with low variance, it seems like the teacher strives to make sure all students have a firm understanding of the subject, but nobody is exemplary. In the class with high variance, the teacher might focus more of their attention on certain students. This might enable some students to ace their tests, but other students get left behind.

If we only looked at statistics like mean, median, and mode, these nuances in the data wouldn't be represented.

Variance is useful because it is a measure of spread. While we might get a general understanding of the spread by looking at a histogram, computing the variance gives us a numerical value that helps us describe the level of confidence of our comparison.

Consider again the following example and look that both datasets that have a similar mean, but look very different.

```
hist(dataset_one, col=rgb(0,0,1,1/4),xlim=c(-30,30), main="Datasets 1 and 2",breaks=80)
hist(dataset_two,  col=rgb(1,0,0,1/4), add=T, breaks=80)
legend("topright", c("Dataset 1", "Dataset 2"), fill=c("blue", "red"))
box()
```




```
print(paste("The mean of the first dataset is ", mean(dataset_one)))
```

```
## [1] "The mean of the first dataset is  0.3133598953587"
```

```
print(paste("The mean of the second dataset is ",mean(dataset_two)))
```

```
## [1] "The mean of the second dataset is  0.05276583677259"
```

```
print(paste("The variance of the first dataset is ", var(dataset_one)))
```

```
## [1] "The variance of the first dataset is  101.181673885126"
```

```
print(paste("The variance of the second dataset is ", var(dataset_two)))
```

```
## [1] "The variance of the second dataset is  124.870053481285"
```

Standar Deviation

Variance Recap

By finding the variance of a dataset, we can get a numeric representation of the spread of the data. But what does that number really mean? How can we use this number to interpret the spread? It turns out, using variance isn't necessarily the best statistic to use to describe spread. Luckily, there is another statistic — standard deviation — that can be used instead.

Variance is a tricky statistic to use because its units are different from both the mean and the data itself. Because the formula for variance includes squaring the difference between the data and the mean, the variance is measured in units squared. This result is hard to interpret in context with the mean or the data because their units are different. This is where the statistic standard deviation is useful.

Standard deviation is computed by taking the square root of the variance. sigma σ is the symbol commonly used for standard deviation. Conveniently, sigma squared $\sigma^2$is the symbol commonly used for variance:

$$\sigma = \sqrt{\sigma^2} = \sqrt{\frac{\sum_i^n (x_i - \mu)^2}{n}}$$

In R, you can take the square root of a number using ^ 0.5 or sqrt(), up to you which one you prefer:

```
num <- 25
num ^ 0.5
```

```
## [1] 5
```

```
sqrt(num)
```

```
## [1] 5
```

Standard Deviation in R

There is an R function dedicated to finding the standard deviation of a dataset — we can cut out the step of first finding the variance. The R function sd() takes a dataset as a parameter and returns the standard deviation of that dataset:

```
dataset <- c(4, 8, 15, 16, 23, 42)
standard_deviation <- sd(dataset)
standard_deviation
```

```
## [1] 13.49074
```

Using Standard Deviation

Now that we're able to compute the standard deviation of a dataset, what can we do with it? Now that our units match, our measure of spread is easier to interpret. By finding the number of standard deviations a data point is away from the mean, we can begin to investigate how unusual that datapoint truly is. In fact, you can usually expect around 68% of your data to fall within one standard deviation of the mean, 95% of your data to fall within two standard deviations of the mean, and 99.7% of your data to fall within three standard deviations of the mean. For a better understanding of this property see Three-sigma rule or if you have some stats knowledge you may already know this result as a corolary of Chebyshev's inequality

If you have a data point that is over three standard deviations away from the mean, that's an incredibly unusual piece of data!

Quartiles

A common way to communicate a high-level overview of a dataset is to find the values that split the data into four groups of equal size. By doing this, we can then say whether a new datapoint falls in the first, second, third, or fourth quarter of the data.

The values that split the data into fourths are the quartiles. Those values are called the first quartile (Q1), the second quartile (Q2), and the third quartile (Q3)

Q2

Let's begin by finding the second quartile (Q2). **Q2 happens to be exactly the median**. Half of the data falls below Q2 and half of the data falls above Q2.

The first step in finding the quartiles of a dataset is to sort the data from smallest to largest. For example, below is an unsorted dataset:

```
example_set <- c(8, 15, 4, -108, 16, 23, 42)
#Sort from smallest to largest.
example_set <- sort(example_set)
example_set
```

```
## [1] -108    4    8   15   16   23   42
```

Now that the list is sorted, we can find Q2. In the example dataset above, Q2 (and the median) is 15 — there are three points below 15 and three points above 15.

You might be wondering what happens if there is an even number of points in the dataset. For example, if we remove the -108 from our dataset, it will now look like this c(8, 15, 4, 16, 23, 42) Q2 now falls somewhere between 15 and 16. There are a couple of different strategies that you can use to calculate Q2 in this situation. One of the more common ways is to take the average of those two numbers. In this case, that would be 15.5.

Q1 and Q3

Now that we've found Q2, we can use that value to help us find Q1 and Q3. Recall our demo dataset:

```
example_set <- c(8, 15, 4, -108, 16, 23, 42)
#Sort from smallest to largest.
example_set <- sort(example_set)
example_set
```

```
## [1] -108    4    8   15   16   23   42
```

In this example, Q2 is 15. To find Q1, we take all of the data points smaller than Q2 and find the median of those points. In this case, the points smaller than Q2 are c(-108,4,8) The median of that smaller dataset is 4. That's Q1!

To find Q3, do the same process using the points that are larger than Q2. We have the following points: c(16, 23, 42)c(16,23,42) The median of those points is 23. That's Q3! We now have three points that split the original dataset into groups of four equal sizes.

Method Two: Including Q2

You just learned a commonly used method to calculate the quartiles of a dataset. However, there is another method that is equally accepted that results in different values! Note that there is no universally agreed upon method of calculating quartiles, and as a result, two different tools might report different results.

The second method includes Q2 when trying to calculate Q1 and Q3. Let's take a look at an example with our same vector:

```r
c(-108, 4, 8, 15, 16, 23, 42)
```

```
## [1] -108    4    8   15   16   23   42
```

Using the first method, we found Q1 to be 4. When looking at all of the points below Q2, we excluded Q2. Using this second method, we include Q2 in each half. For example, when calculating Q1 using this new method, we would now find the median of this dataset: c(-108, 4, 8, 15) Using this method, Q1 is 6. (Take the mean value between 4,8)

Notice that when using an even number of observations both methods have different results, but with an uneven number both methods are the samen.

Quartiles in R

We were able to find quartiles manually by looking at the dataset and finding the correct division points. But that gets much harder when the dataset starts to get bigger. Luckily, there is a function in base R that will find the quartiles for you. The base R function that we'll be using is named quantile().

The code below calculates the third quartile of the given dataset:

```r
dataset <- c(50, 10, 4, -3, 4, -20, 2)
third_quartile <- quantile(dataset, 0.75)
```

The quantile() function takes two parameters. The first is the dataset you're interested in. The second is a number between 0 and 1. Since we calculated the third quartile, we used 0.75 — we want the point that splits the first 75% of the data from the rest. For the second quartile, we'd use 0.5. This will give you the point that 50% of the data is below and 50% is above.

Notice that the dataset doesn't need to be sorted for R's function to work!

Quartiles are some of the most commonly used descriptive statistics. For example, You might see schools or universities think about quartiles when considering which students to accept. Businesses might compare their revenue to other companies by looking at quartiles.

In fact quartiles are so commonly used that the three quartiles, along with the minimum and the maximum values of a dataset, are called the five-number summary() of the dataset. These five numbers help you quickly get a sense of the range, centrality, and spread of the dataset.

```r
dataset <- c(50, 10, 4, -3, 4, -20, 2)
summary(dataset)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -20.000  -0.500   4.000   6.714   7.000  50.000
```

Quantiles in R

Base R has a function named quantile() that will quickly calculate the quantiles of a dataset for you. quantile() takes two parameters. The first is the dataset that you are using. The second parameter is a single number or a vector of numbers between 0 and 1. These numbers represent the places in the data where you want to split.

For example, if you only wanted the value that split the first 10% of the data apart from the remaining 90%, you could use this code:

```r
dataset <- c(5, 10, -20, 42, -9, 10)
ten_percent <- quantile(dataset, 0.10)
ten_percent
```

```
##   10%
## -14.5
```

ten_percent now holds the value -14.5. This result technically isn't a quantile, because it isn't splitting the dataset into groups of equal sizes — this value splits the data into one group with 10% of the data and another with 90%. However, it would still be useful if you were curious about whether a data point was in the bottom 10% of the dataset.

Many Quantiles

Quantiles are usually a set of values that split the data into groups of equal size. For example, you wanted to get the 5-quantiles, or the four values that split the data into five groups of equal size, you could use this code:

```
dataset <- c(5, 10, -20, 42, -9, 10)
ten_percent <- quantile(dataset, seq(0.2, 1, 0.2))
ten_percent
```

```
##  20%  40%  60%  80% 100%
##   -9    5   10   10   42
```

Note: We can use here the function seq() that generates a sequence of numbers of numbers according to three parameters: The first is the initial number, the second is the final number, and the third parameter is the step or increment.

```
seq(0.1, 1, 0.1)
```

```
##  [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

If we used the values c(0.2, 0.4, 0.7, 0.8), the function would return the four values at those split points. However, those values wouldn't split the data into five equally sized groups. One group would only have 10% of the data and another group would have 30% of the data!

Common Quantiles

One of the most common quantiles is the 2-quantile. This value splits the data into two groups of equal size. Half the data will be above this value, and half the data will be below it. This is also known as the median!

The 4-quantiles, or the quartiles, split the data into four groups of equal size. We found the quartiles in the previous exercise.

Finally, the percentiles, or the values that split the data into 100 groups, are commonly used to compare new data points to the dataset. You might hear statements like "You are above the 80th percentile in height". This means that your height is above whatever value splits the first 80% of the data from the remaining 20%.

- Quantiles are values that split a dataset into groups of equal size.
- If you have n quantiles, the dataset will be split into n+1 groups of equal size.
- The median is a quantile. It is the only 2-quantile. Half the data falls below the median and half falls above the median.
- Quartiles and percentiles are other common quantiles. Quartiles split the data into 4 groups while percentiles split the data into 100 groups.

Interquantile Range

Range Review

One of the most common statistics to describe a dataset is the range. The range of a dataset is the difference between the maximum and minimum values. While this descriptive statistic is a good start, it is important to

consider the impact outliers have on the results. The interquartile range (IQR) is a descriptive statistic that tries to solve this problem. The IQR ignores the tails of the dataset, so you know the range around-which your data is centered.

```
#range <- max(df_column)-min(df_column)
```

The interquartile range is the difference between the third quartile (Q3) and the first quartile (Q1). Remember that the first quartile is the value that separates the first 25% of the data from the remaining 75%. The third quartile is the opposite — it separates the first 75% of the data from the remaining 25%. The interquartile range is the difference between these two values.

```
#interquartile_range <- quantile(df,0.75) - quantile(df,0.25)
```

IQR in R

The stats library has a function that can calculate the IQR all in one step. The IQR() function takes a dataset as a parameter and returns the Interquartile Range.

```
dataset = c(4, 10, 38, 85, 193)
interquartile_range = IQR(dataset)
interquartile_range
```

```
## [1] 75
```

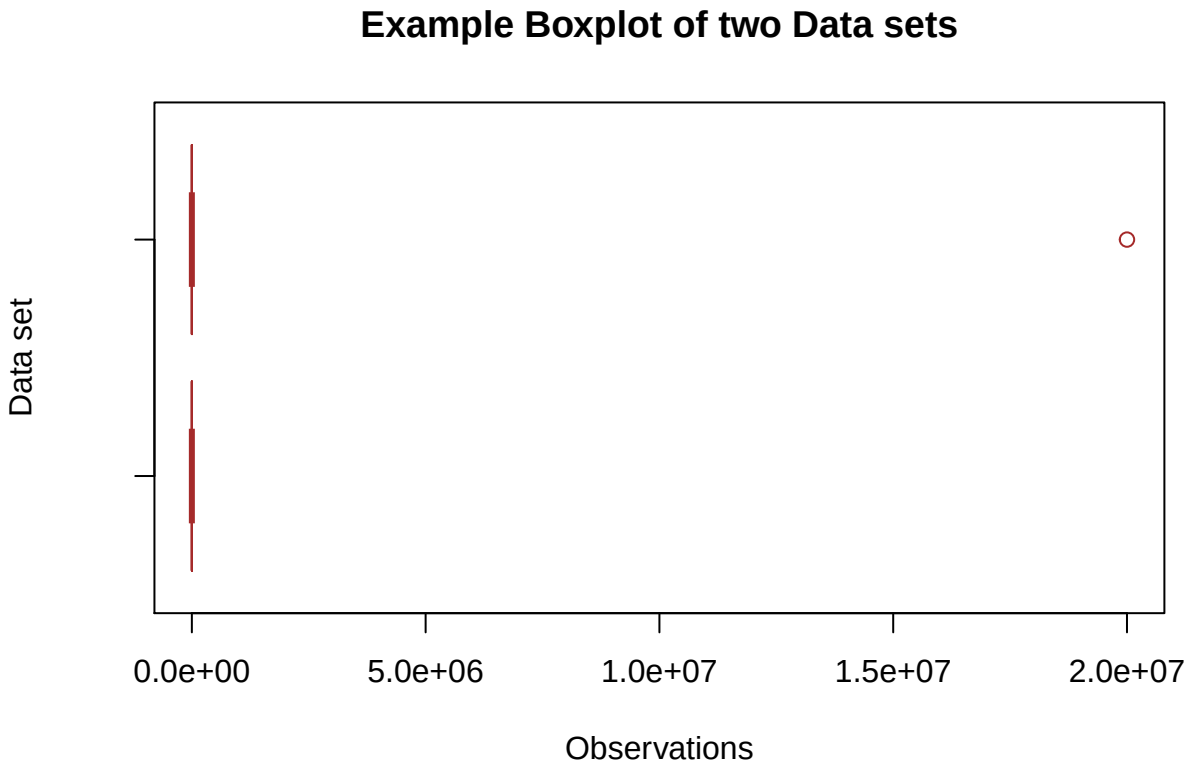
The main takeaway of the IQR is that it is a statistic, like the range, that helps describe the spread of the center of the data. However, unlike the range, the IQR is robust. A statistic is robust when outliers have little impact on it. For example, the IQRs of the two datasets below are identical, even though one has a massive outlier.

```
dataset_one = c(6, 9, 10, 45, 190, 200) # IQR is 144.5
dataset_two = c(6, 9, 10, 45, 190, 20000000) # IQR is 144.5
```

By looking at the IQR instead of the range, you can get a better sense of the spread of the middle of the data.

The interquartile range is displayed in a commonly-used graph — the box plot. In a box plot, the ends of the box are Q1 and Q3. So the length of the box is the IQR.

```
boxplot(dataset_one,dataset_two, main = "Example Boxplot of two Data sets",
xlab = "Observations", ylab = "Data set", col = "orange", border = "brown", horizontal = TRUE, notch = 
```



Hypothesis testing.

Introduction

Say you work for a major social media website. Your boss comes two you with two questions:

- Does the demographic of users on your site match the company's expectation?
- Did the new interface update affect user engagement?

With terabytes of user data at your hands, you decide the best way to answer these questions is with statistical hypothesis tests!

Statistical hypothesis testing is a process that allows you to evaluate if a change or difference seen in a dataset is "real", or if it's just a result of random fluctuation in the data.Hypothesis testing can be an integral component of any decision making process. It provides a framework for evaluating how confident one can be in making conclusions based on data. Some instances where this might come up include:

- A professor expects an exam average to be roughly 75%, and wants to know if the actual scores line up with this expectation. Was the test actually too easy or too hard?
- A product manager for a website wants to compare the time spent on different versions of a homepage. Does one version make users stay on the page significantly longer?

In this section, you will cover the fundamental concepts that will help you run and evaluate hypothesis tests:

- Sample and Population Mean
- P-Values
- Significance Level
- Type I and Type II Errors

You will then learn about three different hypothesis tests you can perform to answer the kinds of questions discussed above:

- One Sample T-Test

- Two Sample T-Test
- ANOVA (Analysis of Variance)

Let's get started!

Sample Mean and Population Mean - I

Suppose you want to know the average height of an oak tree in your local park. On Monday, you measure 10 trees and get an average height of 32 ft. On Tuesday, you measure 12 different trees and reach an average height of 35 ft. On Wednesday, you measure the remaining 11 trees in the park, whose average height is 31 ft. The average height for all 33 trees in your local park is 32.8 ft.

The collection of individual height measurements on Monday, Tuesday, and Wednesday are each called samples. A sample is a subset of the entire population (all the oak trees in the park). The mean of each sample is a sample mean and it is an estimate of the population mean.

Note: the sample means (32 ft., 35 ft., and 31 ft.) were all close to the population mean (32.8 ft.), but were all slightly different from the population mean and from each other.

For a population, the mean is a constant value no matter how many times it's recalculated. But with a set of samples, the mean will depend on exactly which samples are selected. From a sample mean, we can then extrapolate the mean of the population as a whole. There are three main reasons we might use sampling:

- Data on the entire population is not available
- Data on the entire population is available, but it is so large that it is unfeasible to analyze
- Meaningful answers to questions can be found faster with sampling

Example: We've generated a random population of size 300 that follows a normal distribution with a mean of 65. Notice we use the function rnorm(), to look more deeply in this function check rnorm.

```
# generate random population
population <- rnorm(300, mean=65, sd=3.5)

# calculate population mean here:
population_mean <- mean(population)
cat("The population mean is:", population_mean)
```

```
## The population mean is: 64.88982
```

Let's look at how the means of different samples can vary within the same population.

```
# generate sample 1
sample_1 <- sample(population, size=30)
# calculate sample 1 mean
sample_1_mean <- mean(sample_1)
sample_1_mean
```

```
## [1] 65.35516
```

Look at the population mean and the sample means. Are they all the same? All different? Why?

```
# generate samples 2,3,4 and 5
sample_2 <- sample(population, size=30)
sample_3 <- sample(population, size=30)
sample_4 <- sample(population, size=30)
sample_5 <- sample(population, size=30)
sample_2_mean <- mean(sample_2)
sample_2_mean
```

```
## [1] 64.91847
```

```r
sample_3_mean <- mean(sample_3)
sample_3_mean
```

```
## [1] 64.53202
```

```r
sample_4_mean <- mean(sample_4)
sample_4_mean
```

```
## [1] 63.93118
```

```r
sample_5_mean <- mean(sample_5)
sample_5_mean
```

```
## [1] 65.27522
```

Sample Mean and Population Mean - II

In the previous exercise, the sample means you calculated closely approximated the population mean. This won't always be the case!

Consider a tailor of school uniforms at a school for students aged 11 to 13. The tailor needs to know the average height of all the students in order to know which sizes to make the uniforms. The tailor measures the heights of a random sample of 20 students out of the 300 in the school. The average height of the sample is 57.5 inches. Using this sample mean, the tailor makes uniforms that fit students of this height, some smaller, and some larger.

After delivering the uniforms, the tailor starts to receive some feedback — many of the uniforms are too small! They go back to take measurements on the rest of the students, collecting the following data:

- 11 year olds average height: 56.7 inches
- 12 year olds average height: 59 inches
- 13 year olds average height: 62.8 inches
- All students average height (population mean): 59.5 inches

The original sample mean was off from the population mean by 2 inches! How did this happen? The random sample of 20 students was skewed to one direction of the total population. More 11 year olds were chosen in the sample than is representative of the whole school, bringing down the average height of the sample. This is called a **sampling error**, and occurs when a sample is not representative of the population it comes from. How do you get an average sample height that looks more like the average population height, and reduce the chance of a sampling error?

Selecting only 20 students for the sample allowed for the chance that only younger, shorter students were included. This is a natural consequence of the fact that a sample has less data than the population to which it belongs. If the sample selection is poor, then you will have a sample mean seriously skewed from the population mean. There is one sure way to mitigate the risk of having a skewed sample mean — take a larger set of samples! The sample mean of a larger sample set will more closely approximate the population mean, and reduce the chance of a sampling error.

Hypothesis Formulation

You begin the statistical hypothesis testing process by defining a hypothesis, or an assumption about your population that you want to test. A hypothesis can be written in words, but can also be explained in terms of the sample and population means you just learned about.

Say you are developing a website and want to compare the time spent on different versions of a homepage. You could run a hypothesis test to see if version A or B makes users stay on the page significantly longer. Your hypothesis might be: *"The average time spent on homepage A is greater than the average time spent on*

homepage B." You could also restate this in terms of population mean: *"The population mean of time spent on homepage A is the same as the population mean of time spent on homepage B."*

After collecting some sample data on how users interact with each homepage, you can then run a hypothesis test using the data collected to determine whether your null hypothesis is true or false, or can be rejected (i.e. there is a difference in time spent on homepage A or B).

Other examples of hypothesis:

A researcher at a pharmaceutical company is working on the development of a new medication to lower blood pressure, DeePressurize. They run an experiment with a control group of 100 patients that receive a placebo (a sugar pill), and an experimental group of 100 patients that receive DeePressurize. Blood pressure measurements are taken after a 3 month period on both groups of patients. The researcher wants to run a hypothesis test to compare the resulting datasets.

```
hypo_a <- "DeePressurize lowers blood pressure in patients."
hypo_b <- "DeePressurize has no effect on blood pressure in patients."
```

A product manager at a dating app company is developing a new user profile page with a different picture layout. They want to see if the new layout results in more matches between users than the current layout. 50% of profiles are updated to the new layout, and over a 1 month period the number of matches for users with the new layout and the original layout are recorded. The product manager wants to run a hypothesis test to compare the resulting datasets.

```
hypo_c <- "The new profile layout has no effect on number of matches with other users."
hypo_d <- "The new profile layout results in more matches with other users than the original layout."
```

Designing an Experiment

Suppose you want to know if students who study history are more interested in volleyball than students who study chemistry. Before doing anything else to answer your original question, you come up with a null hypothesis: *"History and chemistry students are interested in volleyball at the same rates."*

To test this hypothesis, you need to design an experiment and collect data. You invite 100 history majors and 100 chemistry majors from your university to join an extracurricular volleyball team. After one week, 34 history majors sign up (34%), and 39 chemistry majors sign up (39%). More chemistry majors than history majors signed up, but is this a "real", or significant difference? Can you conclude that students who study chemistry are more interested in volleyball than students who study history?

In your experiment, the 100 history and 100 chemistry majors at your university are samples of their respective populations (all history and chemistry majors). The sample means are the percentages of history majors (34%) and chemistry majors (39%) that signed up for the team, and the difference in sample means is 39% - 34% = 5%. The population means are the percentage of history and chemistry majors worldwide that would sign up for an extracurricular volleyball team if given the chance

You want to know if the difference you observed in these sample means (5%) reflects a difference in the population means, or if the difference was caused by sampling error, and the samples of students you chose do not represent the greater populations of history and chemistry students.

Restating the null hypothesis in terms of the population means yields the following:

"*The percentage of all history majors who would sign up for volleyball is the same as the percentage of all chemistry majors who would sign up for volleyball, and the observed difference in sample means is due to sampling error."*

This is the same as saying, "If you gave the same volleyball invitation to every history and chemistry major in the world, they would sign up at the same rate, and the sample of 200 students you selected are not representative of their populations."

Type I and Type II Errors

When using automated processes to make decisions, you need to be aware of how this automation can lead to mistakes. Computer programs can be as fallible as the humans who design them. Because of this, there is a responsibility to understand what can go wrong and what can be done to contain these foreseeable problems. In statistical hypothesis testing, there are two types of error.

A **Type I** error occurs when a hypothesis test finds a correlation between things that are not related. This error is sometimes called a **"false positive"** and occurs when the **null hypothesis is rejected even though it is true**.

For example, consider the history and chemistry major experiment from the previous exercise. Say you run a hypothesis test on the sample data you collected and conclude that there is a significant difference in interest in volleyball between history and chemistry majors. You have rejected the null hypothesis that there is no difference between the two populations of students. If, in reality, your results were due to the groups you happened to pick (sampling error), and there actually is no significant difference in interest in volleyball between history and chemistry majors in the greater population, you have become the victim of a false positive, or a Type I error.

The second kind of error, a **Type II** error, is failing to find a correlation between things that are actually related. This error is referred to as a **"false negative"** and occurs when the **null hypothesis is not rejected even though it is false**.

For example, with the history and chemistry student experiment, say that after you perform the hypothesis test, you conclude that there is no significant difference in interest in volleyball between history and chemistry majors. You did not reject the null hypothesis. If there actually is a difference in the populations as a whole, and there is a significant difference in interest in volleyball between history and chemistry majors, your test has resulted in a false negative, or a Type II error.

Example:

You will find four vectors: actual_positive, actual_negative, experimental_positive, and experimental_negative. These vectors represent outcomes from a statistical experiment.

```
# the true positives and negatives:
actual_positive <- c(2, 5, 6, 7, 8, 10, 18, 21, 24, 25, 29, 30, 32, 33, 38, 39, 42, 44, 45, 47)
actual_negative <- c(1, 3, 4, 9, 11, 12, 13, 14, 15, 16, 17, 19, 20, 22, 23, 26, 27, 28, 31, 34, 35, 36

# the positives and negatives we determine by running the experiment:
experimental_positive <- c(2, 4, 5, 7, 8, 9, 10, 11, 13, 15, 16, 17, 18, 19, 20, 21, 22, 24, 26, 27, 28
experimental_negative <- c(1, 3, 6, 12, 14, 23, 25, 29, 30, 31, 33, 34, 37, 41, 42, 43, 44, 47, 48)
```

The base R intersect() function can take two vectors as arguments and returns a vector containing the common elements.

Type 1: False positives occur when the experiment indicates a positive outcome, but the actual data is negative (the null hypothesis is rejected even though it is true).

```
# Define type_i_errors and type_ii_errors:
type_i_errors <- intersect(experimental_positive,actual_negative)
type_i_errors
```

```
##  [1]  4  9 11 13 15 16 17 19 20 22 26 27 28 35 36 40 46 49
```

```
type_ii_errors <- intersect(experimental_negative,actual_positive)
type_ii_errors
```

```
## [1]  6 25 29 30 33 42 44 47
```

P-Values

You know that a hypothesis test is used to determine the validity of a null hypothesis. Once again, the null hypothesis states that there is no actual difference between the two populations of data. But what result does a hypothesis test actually return, and how can you interpret it?

A hypothesis test returns a few numeric measures, most of which are out of the scope of this introductory lesson. Here we will focus on one: p-values. P-values help determine how confident you can be in validating the null hypothesis. In this context, a p-value is the probability that, assuming the null hypothesis is true, you would see at least such a difference in the sample means of your data.

Consider the experiment on history and chemistry majors and their interest in volleball from a previous exercise:

- Null Hypothesis: "History and chemistry students are interested in volleyball at the same rates"
- Experiment Sample Means: 34% of history majors and 39% of chemistry majors sign up for the volleyball class

A hypothesis test on the experiment data that returns a p-value of 0.04 would indicate that, assuming the null hypothesis is true and there is no difference in preference for volleyball between all history and chemistry majors, you would see at least such a difference in sample mean (39% - 34% = 5%) only 4% of the time due to sampling error.

Essentially, if you ran this same experiment 100 times, you would expect to see as large a difference in the sample means only 4 times given the assumption that there is no actual difference between the populations (i.e. they have the same mean).

Seems like a really small probability, right? Are you thinking about rejecting the null hypothesis you originally stated?

Another example:

You are big fan of apples, so you gather 10 green and 10 red apples to compare their weights. The green apples average 150 grams in weight, and the red apples average 160 grams in weight. You run a hypothesis test to see if there is a significant difference in the weight of green and red apples. The test returns a p-value of 0.2. Remember that a p-value is a probability that, assuming the null hypothesis is true, you would see at least such a difference in the sample means of your data

So you should interpret this p-value of 0.2 as: *"There is a 20% chance that the difference in average weight of green and red apples is due to random sampling"*

Significance Level

While a hypothesis test will return a p-value indicating a level of confidence in the null hypothesis, it does not definitively claim whether you should reject the null hypothesis. To make this decision, you need to determine a threshold p-value for which all p-values below it will result in rejecting the null hypothesis. This threshold is known as the *significance level.*

A higher significance level is more likely to give a false positive, as it makes is "easier" to state that there is a difference in the populations of your data when such a difference might not actually exist. If you want to be very sure that the result is not due to sampling error, you should select a very small significance level.

It is important to choose the significance level before you perform a statistical hypothesis test. If you wait until after you receive a p-value from a test, you might pick a significance level such that you get the result you want to see. For instance, if someone is trying to publish the results of their scientific study in a journal, they might set a higher significance level that makes their results appear statistically significant. Choosing a significance level in advance helps keep everyone honest.

It is an industry standard to set a significance level of 0.05 or less, meaning that there is a 5% or less chance that your result is due to sampling error. If the significance level is set to 0.05, any test that returns a p-value

less than 0.05 would lead you to reject the null hypothesis. If a test returns a p-value greater than 0.05, you would not be able to reject the null hypothesis.

One Sample T-Test

Consider the fictional business BuyPie, which sends ingredients for pies to your household so that you can make them from scratch.

Suppose that a product manager hypothesizes the average age of visitors to BuyPie.com is 30. In the past hour, the website had 100 visitors and the average age was 31. Are the visitors older than expected? Or is this just the result of chance (sampling error) and a small sample size?

You can test this using a One Sample T-Test. A One Sample T-Test compares a sample mean to a hypothetical population mean. It answers the question "What is the probability that the sample came from a distribution with the desired mean?"

The first step is formulating a null hypothesis, which again is the hypothesis that *there is no difference between the populations you are comparing.* The second population in a One Sample T-Test is the hypothetical population you choose. The **null hypothesis** that this test examines can be phrased as follows: **"The set of samples belongs to a population with the target mean".**

One result of a One Sample T-Test will be a p-value, which tells you whether or not you can reject this null hypothesis. If the **p-value you receive is less than your significance level, normally 0.05, you can reject the null hypothesis and state that there is a significant difference.**

R has a function called t.test() in the stats package which can perform a One Sample T-Test for you. t.test() requires two arguments, a distribution of values and an expected mean:

```
# results <- t.test(sample_distribution, mu = expected_mean)
```

- Sample_distribution is the sample of values that were collected.
- Mu is an argument indicating the desired mean of the hypothetical population.
- Expected_mean is the value of the desired mean.

t.test() will return, among other information we will not cover here, a p-value — this tells you how confident you can be that the sample of values came from a distribution with the specified mean. P-values give you an idea of how confident you can be in a result. Just because you don't have enough data to detect a difference doesn't mean that there isn't one. Generally, the more samples you have, the smaller a difference you can detect.

Example:

We have a small dataset called ages, representing the ages of customers to BuyPie.com in the past hour.

```
ages <- c(32, 34, 29, 29, 22, 39, 38, 37, 38, 36, 30, 26, 22, 22)
```

Even with a small dataset like this, it is hard to make judgments from just looking at the numbers. To understand the data better, let's look at the mean.

```
ages_mean <- mean(ages)
ages_mean
```

```
## [1] 31
```

We use the t.test() function with ages to see what p-value the experiment returns for this distribution, where we expect the mean to be 30.

```
results <- t.test(ages, mu = 30)
results
```

```
##
##  One Sample t-test
```

```
##
## data:  ages
## t = 0.59738, df = 13, p-value = 0.5605
## alternative hypothesis: true mean is not equal to 30
## 95 percent confidence interval:
##  27.38359 34.61641
## sample estimates:
## mean of x
##        31
```

Remember that the p-value tells you whether or not you can reject this null hypothesis. If the p-value you receive is less than your significance level, normally 0.05, you can reject the null hypothesis and state that there is a significant difference. In these case our p-value is way over 0.05,**we can not reject the null hypothesis, so the set of samples belongs to a population with the target mean**

Two Sample T-Test

Suppose that last week, the average amount of time spent per visitor to a website was 25 minutes. This week, the average amount of time spent per visitor to a website was 29 minutes. Did the average time spent per visitor change (i.e. was there a statistically significant bump in user time on the site)? Or is this just part of natural fluctuations?

One way of testing whether this difference is significant is by using a Two Sample T-Test. A Two Sample T-Test compares two sets of data, which are both approximately normally distributed. The **null hypothesis, in this case, is that the two distributions have the same mean.**

You can use R's t.test() function to perform a Two Sample T-Test, as shown below:

```
#results <- t.test(distribution_1, distribution_2)
```

When performing a Two Sample T-Test, t.test() takes two distributions as arguments and returns, among other information, a p-value. Remember, the p-value let's you know the probability that the difference in the means happened by chance (sampling error).

Example: Suppose we have two distributions representing the time spent per visitor to BuyPie.com last week, week_1, and the time spent per visitor to BuyPie.com this week, week_2. They mean and standar deviation looks as follows:

```
#week_1_mean <- 25.44806
#week_2_mean <- 29.02157
#week_1_sd <- 4.577702
#week_2_sd <- 5.553785
```

Performing a Two Sample T-Test using the t.test() function.

```
# results <- t.test(week_1,week_2)
```

Welch Two Sample t-test p-value = 0.0006863 Alternative hypothesis: true difference in means is not equal to 0

We can reject with a significance level of 95% the null hypothesis in favor of the Alternative hypothesis.

Dangers of Multiple T-Tests

Suppose that you own a chain of stores that sell ants, called VeryAnts. There are three different locations: A, B, and C. You want to know if the average ant sales over the past year are significantly different between the three locations. At first, it seems that you could perform T-tests between each pair of stores.

You know that the p-value is the probability that you incorrectly reject the null hypothesis on each t-test. The more t-tests you perform, the more likely that you are to get a false positive, a Type I error. Recall the

probability of the intersection of two independent events from your stats course.

For a p-value of 0.05, if the null hypothesis is true, then the probability of obtaining a significant result is 1 – 0.05 = 0.95. When you run another t-test, the probability of still getting a correct result is 0.95 * 0.95, or 0.9025. That means your probability of making an error is now close to 10%! This error probability only gets bigger with the more t-tests you do.

If you run 3 Two Sample T-Test

```
print(error_prob <- (1-(0.95**3)))
```

```
## [1] 0.142625
```

If you run 4 Two Sample T-Test

```
print(error_prob <- (1-(0.95**4)))
```

```
## [1] 0.1854938
```

Recall that you started with the objective of reducing the probability of Type 1 error to 0.05%, with 4 samples the probability goes all the way up to 18.5%

ANOVA

In the last exercise, you saw that the probability of making a Type I error got dangerously high as you performed more t-tests.

When comparing more than two numerical datasets, the best way to preserve a Type I error probability of 0.05 is to use ANOVA. ANOVA (Analysis of Variance) tests the null hypothesis that all of the datasets you are considering have the same mean. If you reject the null hypothesis with ANOVA, you're saying that at least one of the sets has a different mean; however, it does not tell you which datasets are different.

You can use the stats package function aov() to perform ANOVA on multiple datasets. aov() takes the different datasets combined into a data frame as an argument.

For example, if you were comparing scores on a video game between math majors, writing majors, and psychology majors, you could format the data in a data frame df_scores as follows:

group	score
math major	88
math major	81
writing major	92
writing major	80
psychology major	94
psychology major	83

```
#results <- aov(score ~ group, data = df_scores)
#summary(results)
```

Note: score ~ group indicates the relationship you want to analyze (i.e. how each group, or major, relates to score on the video game) To retrieve the p-value from the results of calling aov(), use the summary() function. The null hypothesis, in this case, is that all three populations have the same mean score on this video game. If you reject this null hypothesis (if the p-value is less than 0.05), you can say you are reasonably confident that a pair of datasets is significantly different. After using only ANOVA, however, you can't make any conclusions on which two populations have a significant difference.

In general:

```
#results <- aov(values ~ group, data = df)
#summary(results)
```

Assumptions of Numerical Hypothesis Tests

Before you use numerical hypothesis tests, you need to be sure that the following things are true:

1. The samples should each be normally distributed...ish

Data analysts in the real world often still perform hypothesis tests on datasets that aren't exactly normally distributed. What is more important is to recognize if there is some reason to believe that a normal distribution is especially unlikely. If your dataset is definitively not normal, the numerical hypothesis tests won't work as intended. This may sound familiar, if not, better check Central Limit Theorem

For example, imagine you have three datasets, each representing a day of traffic data in three different cities. Each dataset is independent, as traffic in one city should not impact traffic in another city. However, it is unlikely that each dataset is normally distributed. In fact, each dataset probably has two distinct peaks, one at the morning rush hour and one during the evening rush hour. In this scenario, using a numerical hypothesis test would be inappropriate.

2. The population standard deviations of the groups should be equal

For ANOVA and Two Sample T-Tests, using datasets with standard deviations that are significantly different from each other will often obscure the differences in group means.

To check for similarity between the standard deviations, it is normally sufficient to divide the two standard deviations and see if the ratio is "close enough" to 1. "Close enough" may differ in different contexts, but generally staying within 10% should suffice.

3. The samples must be independent

When comparing two or more datasets, the values in one distribution should not affect the values in another distribution. In other words, knowing more about one distribution should not give you any information about any other distribution. This may also sound familiar, if not, better check Independent and identically distributed random variables

Here are some examples where it would seem the samples are not independent:

- The number of goals scored per soccer player before, during, and after undergoing a rigorous training regimen
- A group of patients' blood pressure levels before, during, and after the administration of a drug

It is important to understand your datasets before you begin conducting hypothesis tests on them so that you know you are choosing the right test.

Review

- Samples are subsets of an entire population, and the sample mean can be used to approximate the population mean
- The null hypothesis is an assumption that there is no difference between the populations you are comparing in a hypothesis test
- Type I Errors occur when a hypothesis test finds a correlation between things that are not related, and Type II Errors occur when a hypothesis test fails to find a correlation between things that are actually related
- P-Values indicate the probability that, assuming the null hypothesis is true, such differences in the samples you are comparing would exist
- The Significance Level is a threshold p-value for which all p-values below it will result in rejecting the null hypothesis
- One Sample T-Tests indicate whether a dataset belongs to a distribution with a given mean

- Two Sample T-Tests indicate whether there is a significant difference between two datasets
- ANOVA (Analysis of Variance) allows you to detect if there is a significant difference between one of multiple datasets

Thank you.

If you have made it so far (or mayble you just skipped until this section to see what's about), I hope you have now a better understanding of the basic functions of R. Academics and statisticians have developed R over two decades. R has now one of the richest ecosystems to perform data analysis. There are around 12000 packages available in CRAN (open-source repository). It is possible to find a library for whatever the analysis you want to perform. The rich variety of library makes R the first choice for statistical analysis, especially for specialized analytical work.

The cutting-edge difference between R and the other statistical products is the output. R has fantastic tools to communicate the results. Rstudio comes with the library knitr. This very own html document has been made in R using knitr library.

There are many more things to learn about this language that may help you in your activities as an student, economist, quant or any other science. Share this with others and all your feedback is welcome. **Thank you. Sincerely Diego López Tamayo**