

# **LECTURE 2**

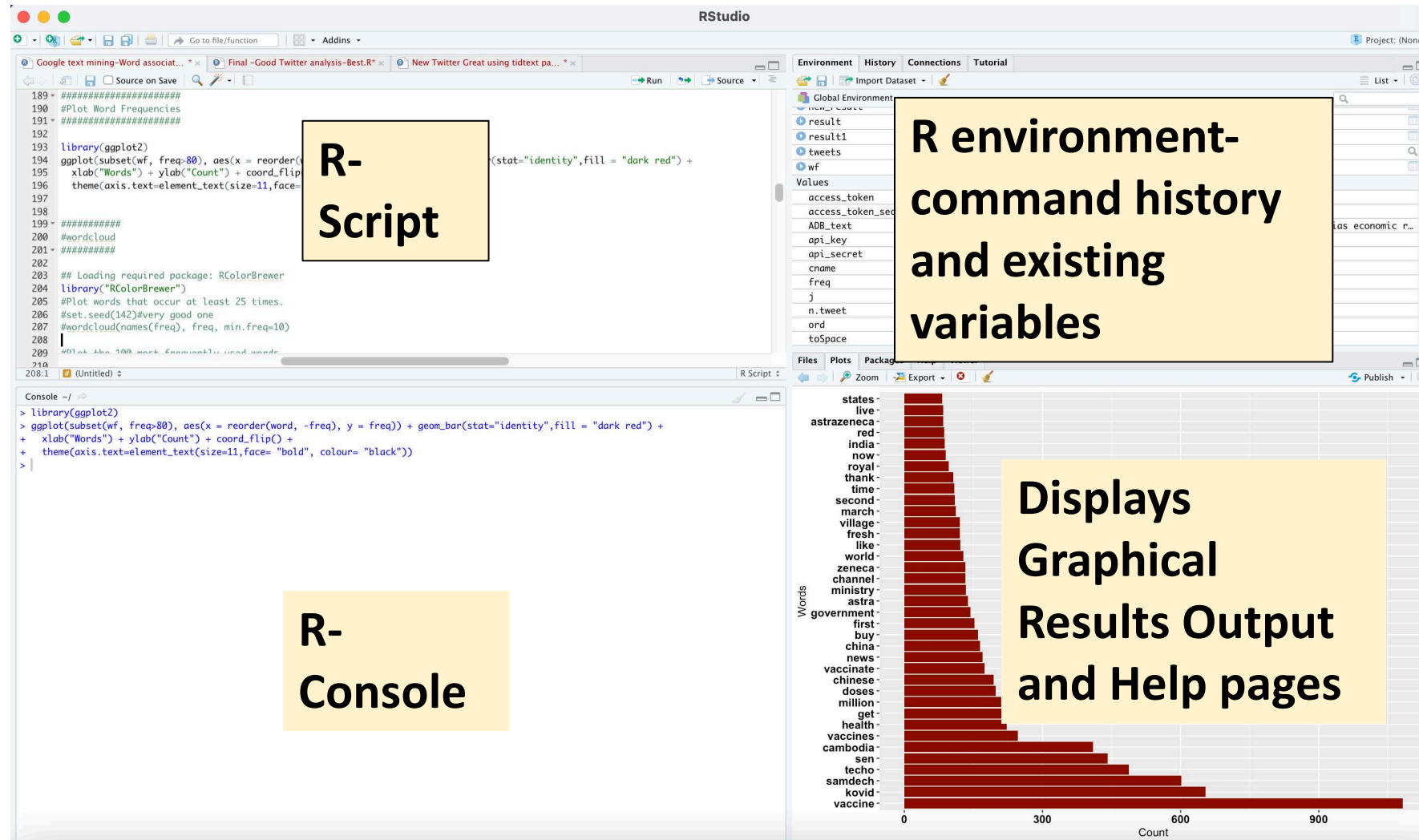
# **FUNDAMENTALS OF R**

Dr. Abbas Maarooof  
aimaarooof@gmail.com  
AITRS-2 June 2021

# **1. Getting started**

# 1.1. Install R and Rstudio

1. You have already installed R and Rstudio at your computer
2. Click on the Rstudio Icon you should see this layout



**Bottom left:** console window (also called command window). Here you can type simple commands after the “>” prompt and R will then execute your command. This is the most important window, because this is where R actually does stuff.

**Top left:** editor window (also called script window). Collections of commands (scripts) can be edited and saved. When you don't get this window, you can open it with File ! New ! R script Just typing a command in the editor window is not enough, it has to get into the command window before R executes the command. If you want to run a line from the script window (or the whole script), you can click Run or press CTRL+ENTER to send it to the command window.

**Top right:** workspace / history window. In the workspace window you can see which data and values R has in its memory. You can view and edit the values by clicking on them. The history window shows what has been typed before.

**Bottom right:** files / plots / packages /help window. Here you can open les, view plots (also previous plots), install and load packages or use the help function.

You can change the size of the windows by drag- ging the grey bars between the windows.

# 1.2. Working directory

Your **working directory** is the folder on your computer in which you are currently working. When you ask R to open a certain file, it will look in the working directory for this file, and when you tell R to save a data file or figure, it will save it in the working directory.

Before you start working, please set your working directory to where all your data and script files are or should be stored.

Type in the command window: `setwd("directoryname")`. For example:

```
> setwd("~/Desktop/AITRS Course")
```

Make sure that the slashes are forward slashes and that you don't forget the apostrophes. R is case sensitive, so make sure you write capitals where necessary.

Within RStudio you can also go to session / Set working directory.

# 1.3. Libraries

R can do many statistical and data analyses. They are organized in so-called *packages* or *libraries*. With the standard installation, most common packages are installed. To get a list of all installed packages, go to the packages window or type `library()` in the console window.

There are many more packages available on the R website. If you want to install and use a pack-age (for example, the package called “geometry”) you should:

1. Install the package: click install packages in the Tools window and type geometry or type `install.packages("ggplot2")` in the command window.
2. Load the package: check box in front of geometry or type `library("ggplot2")` in the command window.

## **2. Introduction to examples of R commands**

# The RStudio Interface

1. You type R code into the bottom line of the Rstudio console pane and then click Enter to run it. *The code you type is called command . The line you type it into is called the command line.*
2. When you type a command at the prompt and hit Enter, your computer executes the command and shows you the results. Then Rstudio displays a fresh prompt for the next command. For example, if you type `1+1` and hit Enter, RStudio will display:



# Basic Syntax

## R Command Prompt

```
> myString <- "Hello, World!"  
> print ( myString)  
[1] "Hello, World!"
```

## R Script File

```
# My first program in R Programming  
myString <- "Hello, World!"  
print ( myString)
```

## Comments

```
# My first program in R Programming
```

## Example (1)

```
> 1+1  
[1] 2  
>
```

it Means the first value

The colon operator (:) return every integer between two integers. It is easy way to create a sequence of numbers.

## Example (2)

```
> 100:130
```

```
[1] 100 101 102 103 104 105 106 107 108 109 110 111 112 113  
114 115 116 117 118 119 120 121
```

```
[23] 122 123 124 125 126 127 128 129 130
```

```
>
```

## Example (3)

If you type an incomplete command and press Enter R will display **a+** prompt, which means it is waiting for you to type the rest of your command. Either finish the command or hit Escape to start over:

```
> 5 -  
+  
+ 1  
[1] 4  
>
```

## Example (4)

If you type a command that R doesn't recognize, R will return an error message. You can try different command at the next prompt:

```
> 3 % 5
```

```
Error: unexpected input in "3 % 5"
```

```
>
```

## Example (5)

Once you get the hang of the command line, you can easily do anything in R that you would do with a calculator. For example, you could do some basic arithmetic:

```
> 2*3
```

```
[1] 6
```

```
> 4-1
```

```
[1] 3
```

```
> 6/ (4-1)
```

```
[1] 2
```

```
>
```

## Example (6)

R treats the hashtag character, # , in a special way; R will not run anything follows a hashtag on a line. This makes hashtags very useful for adding comments and annotations to your code. The hashtag is known as the *commenting symbol in R*.

We will use a single hashtag # to display our own comments and a double hashtag, ##, to display the results of code

*Ctrl+C command* is to cancel command

Now that you know how to use R, let's use it to make a *virtual die*. The `:` Operator returns its results as a vector, a one-dimensional set of numbers:

```
> 1:6  
[1] 1 2 3 4 5  
6  
>
```

Running `1:6` generated a vector of numbers for you to see, but it didn't save the vector anywhere in your computer's memory. If you want to see the numbers again you need to save them somewhere. You can do that by create an *R object* .

# Arithmetic with R

In its most basic form, R can be used as a simple calculator.  
Consider the following arithmetic operators:

Addition: +

Subtraction: -

Multiplication: \*

Division: /

Exponentiation: ^

Modulo: %%



The last two might need some explaining:

The  $\wedge$  operator raises the number to its left to the power of the number to its right: **for example  $3^2$  is 9.**

The modulo returns the remainder of the division of the number to the left by the number on its right, **for example 5 modulo 2 or  $5 \% 2$  is 1.**

For example, the expression "5 mod 2" would evaluate to 1 because 5 divided by 2 leaves a quotient of 2 and a remainder of 1

## Exercise (1): Try to do these

script.R

```
1  # An addition
2  5 + 5
3
4  # A subtraction
5  5 - 5
6
7  # A multiplication
8  3 * 5
9
10 # A division
11 (5 + 5) / 2
12
13 # Exponentiation
14 2^5
15
16 # Modulo
17 28 %% 6
```

**Hint (1):** R makes use of the # sign to add comments, so that you and others can understand what the R code is about. Just like Twitter! Comments are not run as R code, so they will not influence your result.

**Hint (2):** You can also execute R commands straight in the console. This is a good way to experiment with R code, as your submission is not checked for correctness.

# 1. Calculator

R can be used as a calculator. You can just type your equation in the command window after the “>”:

```
> 10^2 + 36  
[1] 136  
>
```

## Exercise:

Compute the difference between 2014 and the year you started at this university and divide this by the difference between 2014 and the year you were born. Multiply this with 100 to get the percentage of your life you have spent at this university. Use brackets if you need them.

**Hint:** If you use brackets and forget to add the closing bracket, the “>” on the command line changes into a \+”. The “+” can also mean that R is still busy with some heavy computation. If you want R to quit what it was doing and give back the “>”, press ESC

## 2.2. Workspace

You can also give numbers a name. By doing so, they become so-called variables which can be used later. For example, you can type in the command window:

```
> a = 4
```

You can see that a appears in the workspace window, which means that R now remembers what a is. You can also ask R what a is (just type a ENTER in the command window):

```
> a  
[1] 4
```

or do calculations with a:

```
> a * 5
```

```
[1] 20
```

If you specify a again, it will forget what value it had before. You can also assign a new value to a using the old one.

```
> a = a + 10
```

```
> a
```

```
[1] 14
```

To remove all variables from R's memory, type

```
> rm(list=ls())
```

or click “clear all: in the workspace window. You can see that RStudio then empties the workspace window. If you only want to remove the variable a, you can type rm(a).

# Variable assignment

A basic concept in (statistical) programming is called a **variable**.

A variable allows you to store a value (e.g. 4) or an object (e.g. a function description) in R. You can then later use this variable's name to easily access the value or the object that is stored within this variable.

You can assign a value 4 to a variable **my\_var** with the command

```
my_var <- 4
```

## Exercise (2):

complete the code in the editor such that it assigns the value 42 to the variable x in the editor. Notice that when you ask R to print x, the value 42 appears.

script.R

```
1 # Assign the value 42 to x
2 x <- 42
3
4 # Print out the value of the variable x
5 x
```

**Hint:** the sign <- means equal to

## Exercise (3):

Suppose you have a fruit basket with five apples. As a data analyst in training, you want to store the number of apples in a variable with the name

**my\_apples**

## Answer:

- Type the following code in the editor: `my_apples <- 5`. This will assign the value 5 to `my_apples`.
- Type: `my_apples` below the second comment. This will print out the value of `my_apples`.

## Exercise (5):

Every tasty fruit basket needs oranges, so you decide to add six oranges. As a data analyst, your reflex is to immediately create the variable `my_oranges` and assign the value 6 to it. Next, you want to calculate how many pieces of fruit you have in total. Since you have given meaningful names to these values, you can now code this in a clear way:



## Answer:

- Assign to `my_oranges` the value 6.
- Add the variables `my_apples` and `my_oranges` and have R simply print the result.
- Assign the result of adding `my_apples` and `my_oranges` to a new variable `my_fruit`.

# Objects

# Objects

What is an object? Jus a name that you can use to call up **stored data**

**For example:** You can save data into an object like a or b, wherever encounters the object, it will replace it with the data saved inside such as:

```
> a<- 1  
> a  
[1] 1  
> a+2  
[1] 3  
>
```

So, for another example, the following code would create an object named `die` that contains the numbers one through six. To see what is stored in an object, just type the object's name by itself:

```
> die <- 1:6  
> die  
[1] 1 2 3 4 5 6  
>
```

When you create an object, the object will appear in the environment pane of RStudio, as shown in Figure 2 . This pane will show you all of the objects you've created since opening RStudio

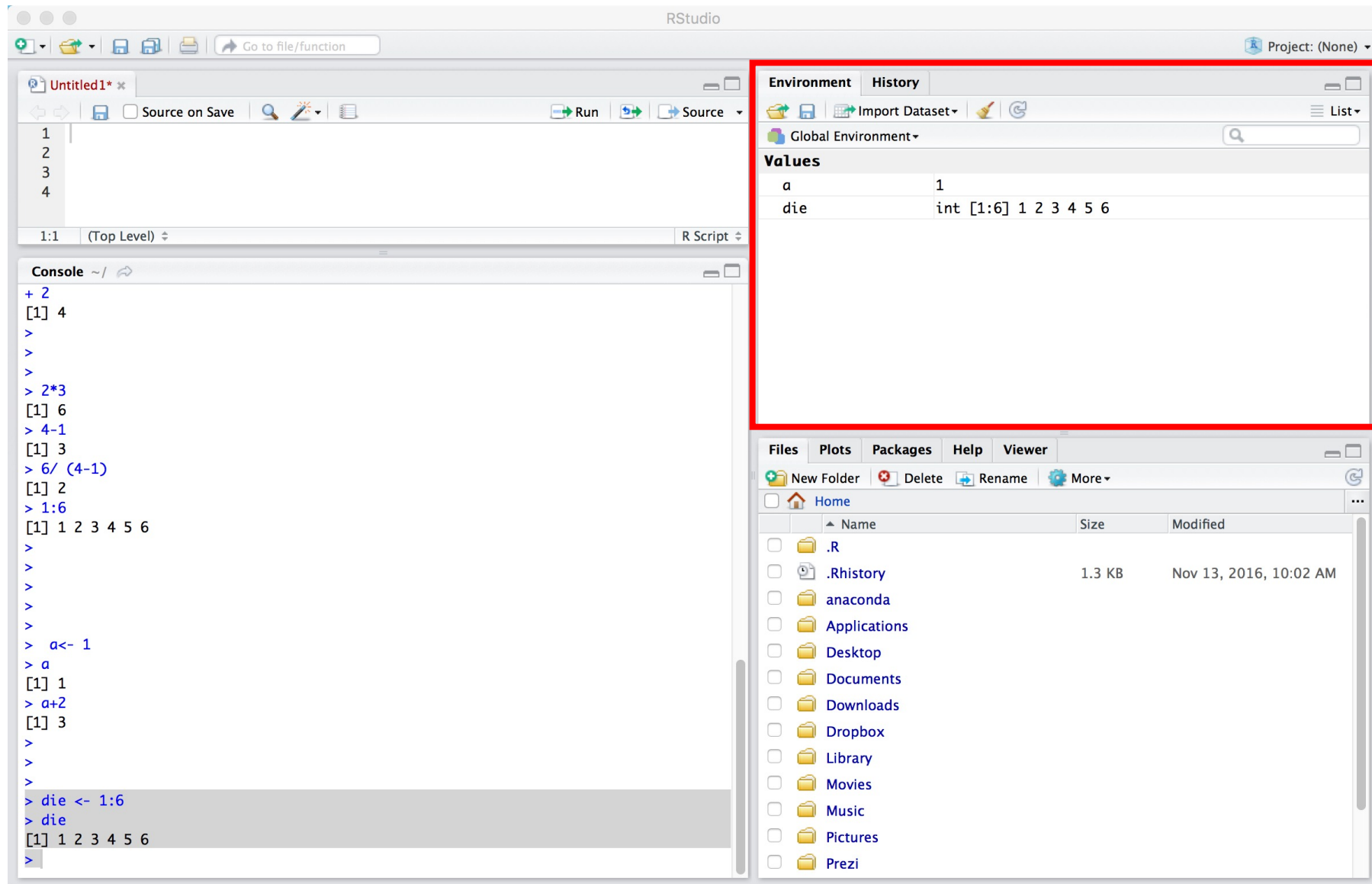


Figure 2: The RStudio environment pane keeps track of the R objects you create

You can name an object anything you want , but there are a few rules:

1. A name cannot start with a number.
2. A name cannot use some special, like ^, !,\$,@,+,-,/,or\*:

Good names	Names that cause errors
a	1trial
b	\$
F00	^mean
my_var	2nd
.day	!bad

R also understands capitalization (or is case-sensitive), so `name` and `Names` will refer to different objects:

```
> Name <- 1  
> name <- 0  
> name+1  
[1] 1  
>
```

Finally, R will overwrite any previous information stored in an object without you for permission. So it is a good to not use names that are already taken:

```
> my_number <- 1  
> my_number  
[1] 1  
> my_number <- 999  
> my_number  
[1] 999  
>
```

You can see which object names you have already used with the function *ls*:

```
> ls ()  
[1] "Name"      "a"         "die"       "mmy_number"  
"my_number" "name"  
>
```

You now have a virtual die stored in your computer's memory. you can access it whenever you like by typing the word *die*. R will replace an object with its contents whenever the object's name appear in a command

You can do all sort of math with die word, let us take a look at how to do that:

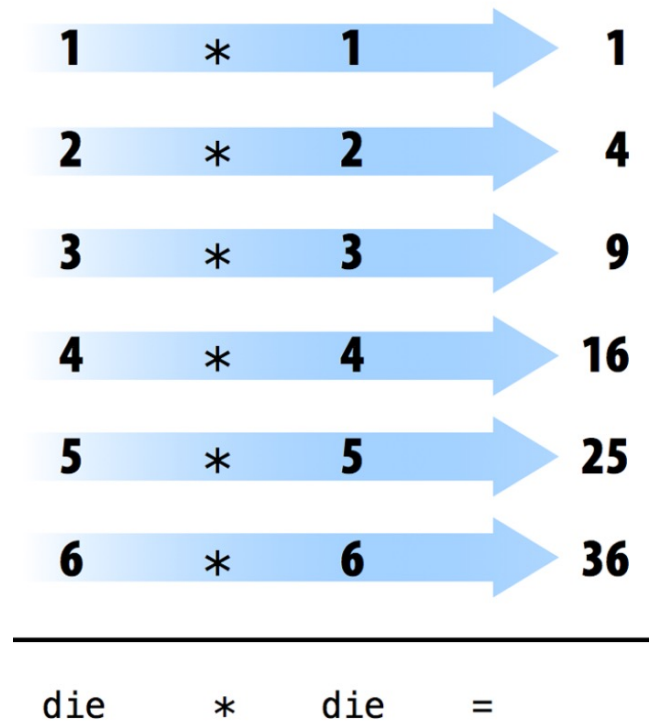


```
> die -1  
[1] 0 1 2 3 4 5  
> die /2  
[1] 0.5 1.0 1.5 2.0 2.5 3.0  
> die*die  
[1] 1 4 9 16 25 36  
>
```

R uses element-wise execution. When you manipulate a set of numbers, R will apply the same operation to each element in the set. So for example, when you run `die - 1`, R subtracts one from each element of `die`.

## For Example

when you run `die * die`, R lines up the two die vectors and then multiplies the first element of vector 1 by the first element of vector 2. It then multiplies the second element of vector 1 by the second element of vector 2, and so on, until every element has been multiplied. The result will be a new vector the same length as the first two, as shown in Figure 3.



1	*	1	1
2	*	2	4
3	*	3	9
4	*	4	16
5	*	5	25
6	*	6	36

---

die \* die =

Figure 3. When R performs element-wise execution, it matches up vectors and then manipulates each pair of elements independently

If you give R two vectors of unequal lengths, R will repeat the shorter vector until it is as long as the longer vector, and then do the maths as show in Figure 4

```
> 1:2
[1] 1 2
> 1:4
[1] 1 2 3 4
> die
[1] 1 2 3 4 5 6
>
>
>
> die + 1:2
[1] 2 4 4 6 6 8
> die + 1:4
[1] 2 4 6 8 6 8
```

Warning message:

In die + 1:4 :

longer object length is not a multiple of shorter object length

>

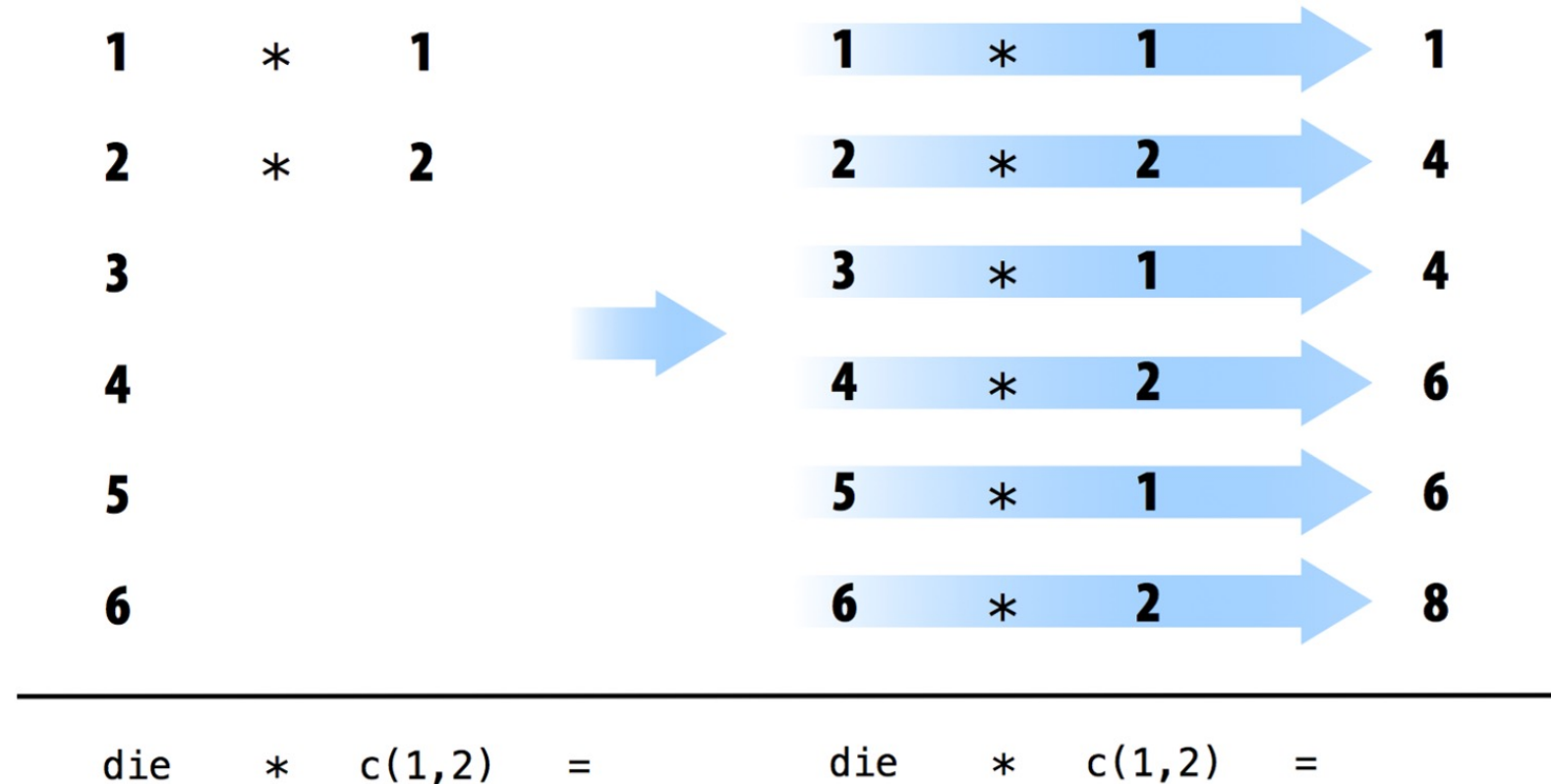


Figure 4. R will repeat a short vector to do element-wise operations with two vectors of uneven lengths.

Now that you can do math with your die object, let's look at how you could “roll” it.

Rolling your die will require something more sophisticated than basic arithmetic; you'll need to randomly select one of the die's values. And for that, you will need a *function*.

# Scripts

## Scripts

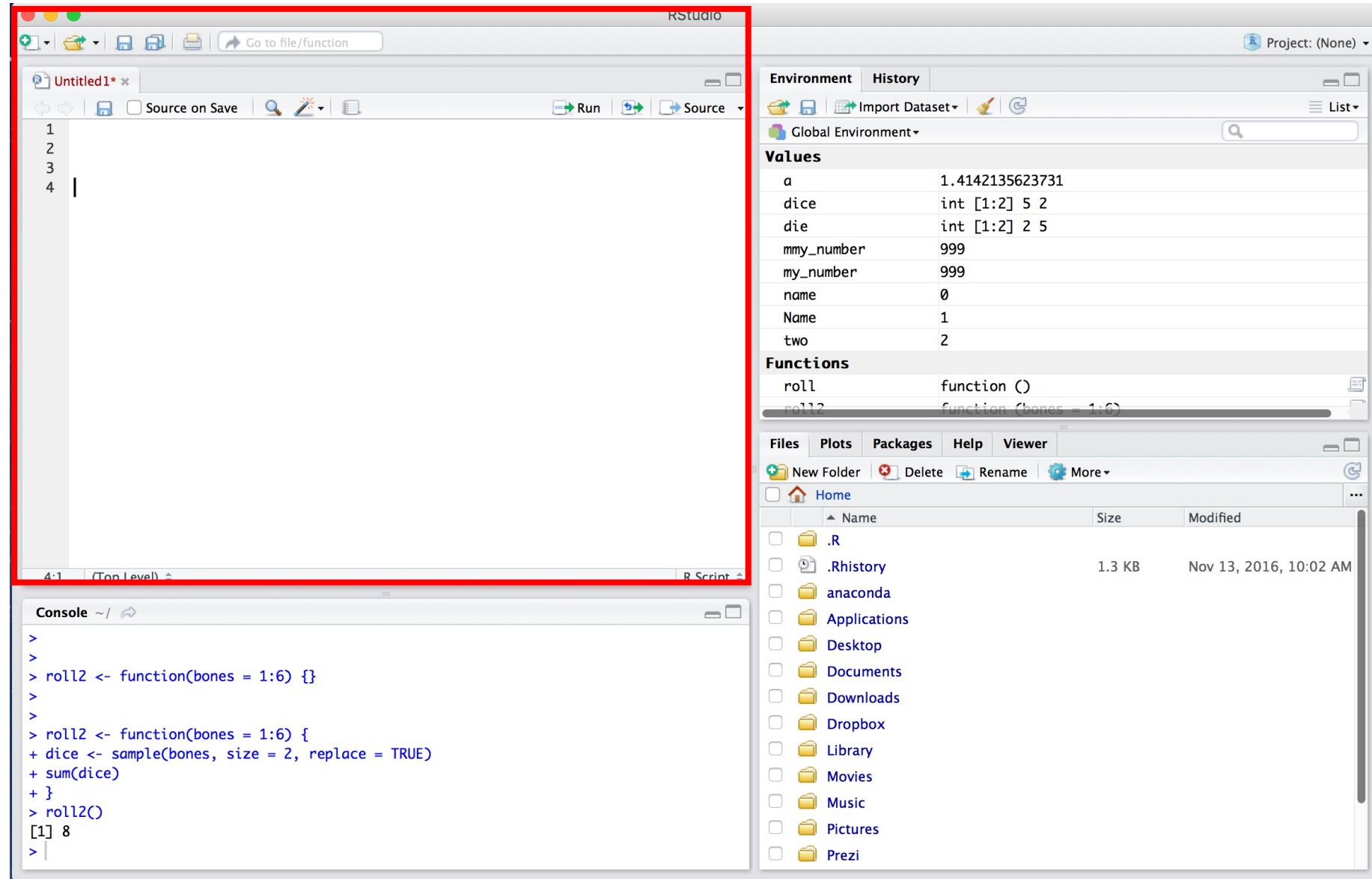
What if you want to edit `roll2` again? You could go back and retype each line of code in `roll2`, but it would be so much easier if you had a draft of the code to start from. You can create a draft of your code as you go by using an ***R script***.

An R script is just a plain text file that you save R code in. You can open an R script in RStudio by going to File > New File > R script in the menu bar. RStudio will then open a fresh script above your console pane, as shown in Figure 8.

# New Script



Figure 4. When you open an R Script (File > New File > R Script in the menu bar), RStudio creates a fourth pane above the console where you can write and edit your code.



RStudio comes with many built-in features that make it easy to work with scripts. First, you can automatically execute a line of code in a script by clicking the Run button, as shown in Figure 9.

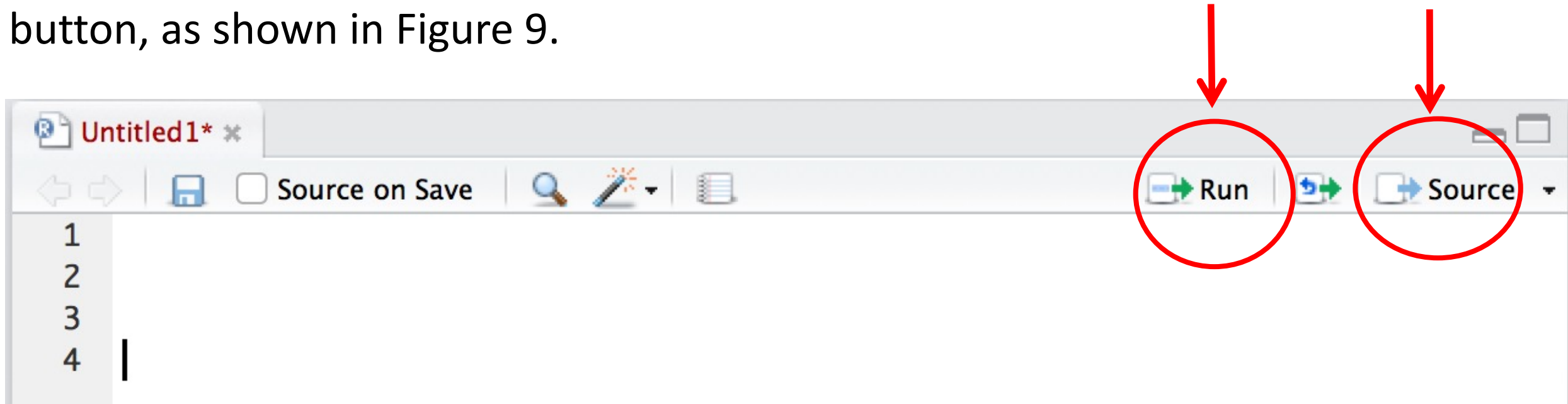


Figure 5. You can run a highlighted portion of code in your script if you click the Run button at the top of the scripts pane. You can run the entire script by clicking the Source button.



R will run whichever line of code your cursor is on. **If you have a whole section highlighted, R will run the highlighted code.** **Alternatively, you can run the entire script by clicking the Source button.** Don't like clicking buttons? You can use Control + Return as a shortcut for the Run button. On Macs, that would be Command + Return.

### **Extract function:**

RStudio comes with a tool that can help you build functions. To use it, highlight the lines of code in your R script that you want to turn into a function. **Then click Code > Extract Function in the menu bar.** RStudio will ask you for a function name to use and then wrap your code in a function call. It will scan the code for undefined variables and use these as arguments. You may want to double-check RStudio's work. It assumes that your code is correct, so if it does something surprising, you may have a problem in your code.

# R Objects

In this lecture, you will learn how to use R to store data sets in your computer's memory and how to retrieve and manipulate data once it's there.

# R Objects

In this lecture, you'll use R to assemble a deck of 52 playing cards.

you'll build the equivalent of an Excel spreadsheet from scratch. When you are finished, your deck of cards will look something like this:

face	suit	value
king	spades	13
queen	spades	12
jack	spades	11
ten	spades	10
nine	spades	9
eight	spades	8
...		

We'll start with the very basics. **The simplest type of object in R is an atomic vector. They are very simple, and they do show up everywhere.** If you look closely enough, you'll see that most structures in R are built from atomic vectors

# Atomic Vectors

An atomic vector is just a simple vector of data. In fact, you've already made an atomic vector, your **die object**.

You can make an atomic vector by grouping some values of data together with `c`:

```
> die <- c(1,2,3,4,5,6)
> die
[1] 1 2 3 4 5 6
> is.vector(die)
[1] TRUE
>
```

`is.vector` tests whether an object is an atomic vector. It returns `TRUE` if the object is an atomic vector and `FALSE` otherwise.

You can also make an atomic vector with just one value. R saves single values as an atomic vector of length 1:

```
> five <- 5  
> five  
[1] 5  
> is.vector(five)  
[1] TRUE  
> length(five)  
[1] 1  
> length(die)  
[1] 6
```

## **length**

length returns the length of an atomic vector.

To create your card deck, you will need to use different types of atomic vectors to save different types of information (text and numbers).

You can do this by using some simple conventions when you enter your data. For example, you can create an integer vector by including a capital L with your input. You can create a character vector by surrounding your input in quotation marks:

```
> int <- 1L  
> text <- "ace"
```

If you'd like to make atomic vectors that have more than one element in them, you can combine an element with the c function

```
> int <- c(1L, 5L)  
> text <- c("ace", "hearts")
```

You may wonder why R uses multiple types of vectors. R will do math with atomic vectors that contain numbers, but not with atomic vectors that contain character strings:



**Get ready to say hello to the six  
types of atomic vectors in R.**

**R has five basic or 'atomic' classes of objects.**

**Wait, what is an object ?**

Everything you see or create in R is an object.

**A vector, matrix, data frame, even a variable is an object.** R treats it that way. So, R has **6** ***basic classes of objects***. This includes:

1. **Doubles**
2. **Integers.**
3. Text (or string) values are called **Characters**.
4. Boolean values (TRUE or FALSE) are called **Logicals**.
5. **Complex**
6. **Raw.**

Each atomic vector stores its values as a one-dimensional vector, and each atomic vector can only store one type of data. You can save different types of data in R by using different types of atomic vectors.

Note the quotation marks is indicate as a character for example "some text".

## Exercise (6):

**Hint (1):** R is case sensitive!

Change the value of the:

script.R

```
1 # Change my_numeric to be 42
2 my_numeric <- 42.5
3
4 # Change my_character to be "universe"
5 my_character <- "some text"
6
7 # Change my_logical to be FALSE
8 my_logical <- TRUE
```

my\_numeric variable to 42.

my\_character variable to "universe". Note that the quotation marks indicate that "universe" is a character.

my\_logical variable to FALSE.

# What's that data type?

When you added `5 + "six"`, you got an error due to a mismatch in data types!

You can avoid such embarrassing situations by checking the data type of a variable beforehand. You can do this with the **`class()`** function, as the code on the right shows.

## Exercise (7):

Complete the code in the editor and also print out the classes of `my_character` and `my_logical`.

script.R

```
1 # Declare variables of different types
2 my_numeric <- 42
3 my_character <- "universe"
4 my_logical <- FALSE
5
6 # Check class of my_numeric
7 class(my_numeric)
8
9 # Check class of my_character
10
11
12 # Check class of my_logical
13
```

# Doubles

A double vector stores regular numbers. The numbers can be positive or negative, large or small, and have digits to the right of the decimal place or not.

```
> die <- c(1,2,3,4,5,6)
> die
[1] 1 2 3 4 5 6
>
```

you can also ask R what type of object an object is with *typeof*.

```
> typeof(die)
[1] "double"
>
```

# Integers

Integer vectors store integers, numbers that can be written without a **decimal component**. As a data scientist, you won't use the integer type very often because you can save integers as a double object.

You can specifically create an integer in R by typing a number followed by an **uppercase L**. For example:

```
> int <- c(-1L, 2L, 4L)
> int
[1] -1  2  4
> typeof(int)
[1] "integer"
>
```

Note that R won't save a number as an integer unless you include the L.

Why would you save your data as an integer instead of a double?

**Important hint:** You can avoid floating-point errors by avoiding decimals and only using integers as shown below:

```
> sqrt(2)^2-2  
[1] 4.440892e-16
```



# Characters

A character vector stores small pieces of text. You can create a character vector in R by typing a character or string of characters surrounded by quotes:

```
> text <- c("Hello", "World")
> text
[1] "Hello" "World"
> typeof(text)
[1] "character"
> typeof("Hello")
[1] "character"
>
```

## Exercise

Can you spot the difference between a character string and a number? Here's a test:

Which of these are character strings and which are numbers? 1, "1", "one".

The individual elements of a character vector are known as strings.

Anything surrounded by quotes in R will be treated as a character string—no matter what appears between the quotes

# Logicals

Logical vectors store TRUEs and FALSEs, R's form of Boolean data. Logicals are very helpful for doing things like comparisons:

```
> 3 > 4  
[1] FALSE  
>
```

Any time you type TRUE or FALSE in capital letters (without quotation marks), R will treat your input as logical data. R also assumes that T and F are shorthand for TRUE and FALSE:

```
> logic <- c(TRUE, FALSE, TRUE)  
> logic  
[1] TRUE FALSE TRUE  
> typeof(logic)  
[1] "logical"  
> typeof(F)  
[1] "logical"  
>
```

# Complex and Raw

Doubles, integers, characters, and logicals are the most common types of atomic vectors in R, but R also recognizes two more types: complex and raw

Complex vectors store complex numbers. To create a complex vector, add an imaginary term to a number with i:

```
> comp <- c(1+1i, 1+2i, 1+3i)
```

```
> comp
```

```
[1] 1+1i 1+2i 1+3i
```

```
>
```

```
> typeof(comp)
```

```
[1] "complex"
```

```
>
```

You can make an empty raw vector of length n with `raw(n)`. See the help page of `raw` for more options when working with this type of data:

```
> raw(3)
[1] 00 00 00
> typeof(raw(3))
[1] "raw"
>
```

## Exercise

Create an atomic vector that stores just the face names of the cards in a royal flush, for example, the ace of spades, king of spades, queen of spades, jack of spades, and ten of spades. The face name of the ace of spades would be “ace,” and “spades” is the suit.

**Which type of vector will you use to save the names?**

**A character vector** is the most appropriate type of atomic vector in which to save card names. You can create one with the `c` function if you surround each name with quotation marks:

```
> hand <- c("ace", "king", "queen", "jack", "ten")
> hand
[1] "ace"  "king" "queen" "jack" "ten"
> typeof(hand)
[1] "character"
>
```

This creates a one-dimensional group of card names.

# Data Types

Generally, while doing programming in any programming language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that, when you create a variable you reserve some space in memory.

You may like to store information of various data types like character, wide character, integer, floating point, double floating point, Boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

### **There are many types of R-objects**

1. Vectors
2. Lists
3. Matrices
4. Arrays
5. Factors
6. Data Frames



# 1. Vectors

**Vector:** As mentioned above, a vector contains object of same class. But, you can mix objects of different classes too. When objects of different classes are mixed in a list, coercion occurs. This effect causes the objects of different types to 'convert' into one class. For example:

```
> qt <- c("Time", 24, "October", TRUE, 3.33) #character  
> ab <- c(TRUE, 24) #numeric  
> cd <- c(2.5, "May") #character
```

To check the class of any object, use `class("vector name")` function.

```
> class(qt)  
"character"
```

To convert the class of a vector, you can use `as.` command.

```
> bar <- 0:5  
> class(bar)  
> "integer"  
> as.numeric(bar)  
> class(bar)  
> "numeric"  
> as.character(bar)  
> class(bar)  
> "character"
```

Similarly, you can change the class of any vector. But, you should pay attention here. If you try to convert a “character” vector to “numeric” , NAs will be introduced. Hence, you should be careful to use this command.

# Create a vector

Do you still remember what you have learned in the first section? Assign the value "Go!" to the variable `vegas`.

Remember: **R is case sensitive!**

script.R

```
1 # Define the variable vegas
2 vegas <- "Go!"
3 vegas
4
5 |
```

Vectors are **one-dimension arrays** that can hold numeric data, character data, or logical data. In other words, a vector is a simple tool to store data. For example, you can store your daily gains and losses in the trading.

In R, you create a vector with the combine function **c()**. You place the vector elements separated by a comma between the parentheses. **For example:**

```
numeric_vector <- c(1, 2, 3)
character_vector <- c("a", "b", "c")
```

**Hint:** Once you have created these vectors in R, you can use them to do calculations.

**Exercise (1):** Complete the code such that `boolean_vector` contains the three elements: `TRUE`, `FALSE` and `TRUE` (in that order).

script.R

```
1 numeric_vector <- c(1, 10, 49)
2 character_vector <- c("a", "b", "c")
3
4 # Complete the code for boolean_vector
5 boolean_vector <-|
```

script.R

```
1 numeric_vector <- c(1, 10, 49)
2 character_vector <- c("a", "b", "c")
3
4 # Complete the code for boolean_vector
5 boolean_vector <- c(TRUE, FALSE, TRUE)
```

# Naming a vector

As a data analyst, it is important to have a clear view on the data that you are using. Understanding what each element refers to is therefore essential.

In the previous exercise, we created a vector with your winnings over the week. Each vector element refers to a day of the week but it is hard to tell which element belongs to which day. It would be nice if you could show that in the vector itself.

You can give a name to the elements of a vector with the **names()** function. Have a look at this example:

```
some_vector <- c("John Doe", "poker player")  
names(some_vector) <- c("Name", "Profession")
```

This code first creates a vector `some_vector` and then gives the two elements a name. The first element is assigned the name `Name`, while the second element is labeled `Profession`. Printing the contents to the console yields following output:

Name	Profession
"John Doe"	"poker player"

# Example:

In R programming, the very basic data types are the R-objects called **vectors** which hold elements of different classes

When you want to create vector with more than one element, you should use **c()** function which means to combine the elements into a vector.

```
# Create a vector.
```

```
apple <- c('red','green',"yellow")
```

```
print(apple)
```

```
# Get the class of the vector.
```

```
print(class(apple))
```

When we execute the above code, it produces the following result –

```
[1] "red" "green" "yellow"
```

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



## **2. List**

# Creating a list

Let us create our first list! To construct a list you use the function `list()`:

```
my_list <- list(comp1, comp2 ...)
```

The arguments to the `list` function are the list components. Remember, these components can be matrices, vectors, other lists, ...

# Example1: Lists

A list is an R-object which can contain many different types of elements inside it like vectors, functions and even another list inside it.

# Create a list.

```
list1 <- list(c(2,5,3),21.3, sin)
```

# Print the list.

```
print(list1)
```

When we execute the above code, it produces the following result –

```
[[1]]
```

```
[1] 2 5 3
```

```
[[2]]
```

```
[1] 21.3
```

```
[[3]] function (x) .Primitive("sin")
```

**Exercise (2):** Construct a list, named `my_list`, that contains the variables `my_vector`, `my_matrix` and `my_df` as list components.

```
1. # Vector with numerics from 1 up to 10
2. my_vector <- 1:10
3.
4. # Matrix with numerics from 1 up to 9
5. my_matrix <- matrix(1:9, ncol = 3)
6.
7. # First 10 elements of the built-in data frame mtcars
8. my_df <- mtcars[1:10,]
9.
10. # Construct list with these different elements:
11. my_list <-
```

script.R

```
1 # Vector with numerics from 1 up to 10
2 my_vector <- 1:10
3
4 # Matrix with numerics from 1 up to 9
5 my_matrix <- matrix(1:9, ncol = 3)
6
7 # First 10 elements of the built-in data frame mtcars
8 my_df <- mtcars[1:10,]
9
10 # Construct list with these different elements:
11 my_list <- list(my_vector, my_matrix, my_df)
```

# Creating a named list

Well done, you're on a roll!

Just like on your to-do list, you want to avoid not knowing or remembering what the components of your list stand for. That is why you should give names to them:

```
my_list <- list(name1 = your_comp1,  
               name2 = your_comp2)
```

This creates a list with components that are named name1, name2, and so on. If you want to name your lists after you've created them, you can use the `names()` function as you did with vectors. The following commands are fully equivalent to the assignment above:

```
my_list <- list(your_comp1, your_comp2)  
names(my_list) <- c("name1", "name2")
```

### Exercise (3):

- Change the code of the previous exercise (see editor) by adding names to the components. Use for my\_vector the name vec, for my\_matrix the name mat and for my\_df the name df.
- Print out my\_list so you can inspect the output.

```
1. # Vector with numerics from 1 up to 10
2. my_vector <- 1:10
3.
4. # Matrix with numerics from 1 up to 9
5. my_matrix <- matrix(1:9, ncol = 3)
6.
7. # First 10 elements of the built-in data frame mtcars
8. my_df <- mtcars[1:10,]
9.
10. # Adapt list() call to give the components names
11. my_list <- list(my_vector, my_matrix, my_df)
12.
13. # Print out my_list
```

script.R

```
1 # Vector with numerics from 1 up to 10
2 my_vector <- 1:10
3
4 # Matrix with numerics from 1 up to 9
5 my_matrix <- matrix(1:9, ncol = 3)
6
7 # First 10 elements of the built-in data frame mtcars
8 my_df <- mtcars[1:10,]
9
10 # Adapt list() call to give the components names
11 my_list <- list(vec=my_vector, mat=my_matrix, df=my_df)
12
13 # Print out my_list
14 my_list
```

# Selecting elements from a list

our list will often be built out of numerous elements and components. Therefore, getting a single element, multiple elements, or a component out of it is not always straightforward

One way to select a component is using the numbered position of that component. For example, to "grab" the first component of `shining_list` you type

```
shining_list[[1]]
```

A quick way to check this out is typing it in the console. Important to remember: to select elements from vectors, you use single square brackets: `[ ]`. Don't mix them up!

You can also refer to the names of the components, with `[[" "]]` or with the `$` sign. Both will select the data frame representing the reviews:

```
shining_list[["reviews"]]
```

```
shining_list$reviews
```

Besides selecting components, you often need to select specific elements out of these components. For example, with `shining_list[[2]][1]` you select from the second component, actors (`shining_list[[2]]`), the first element (`[1]`). When you type this in the console, you will see the answer is Jack Nicholson.

# 3. Matrices



# What's a matrix?

In R, a matrix is a collection of elements of the same data type (numeric, character, or logical) arranged into a fixed number of rows and columns. Since you are only working with rows and columns, a matrix is called **two-dimensional**.

You can construct a matrix in R with the **matrix()** function. Consider the following example:

```
matrix(1:9, byrow = TRUE, nrow = 3)
```

In the **matrix()** function:

The first argument is the collection of elements that R will arrange into the rows and columns of the matrix. Here, we use 1:9 which is a shortcut for c(1, 2, 3, 4, 5, 6, 7, 8, 9). The **argument byrow** indicates that the matrix is filled by the rows. If we want the matrix to be filled by the columns, we just place **byrow = FALSE**.

The third argument **nrow** indicates that the matrix should have three rows.

**Exercise (1):** Construct a matrix with 3 rows containing the numbers 1 up to 9, filled row-wise.

# 3. Matrices

A matrix is a two-dimensional rectangular data set. It can be created using a vector input to the matrix function.

```
# Create a matrix.
```

```
M = matrix( c('a','a','b','c','b','a'), nrow = 2, ncol = 3, byrow = TRUE)  
print(M)
```

When we execute the above code, it produces the following result –

```
[,1] [,2] [,3]  
[1,] "a" "a" "b"  
[2,] "c" "b" "a"
```

## 4. Arrays

While matrices are confined to two dimensions, arrays can be of any number of dimensions. The array function takes a dim attribute which creates the required number of dimension. In the below example we create an array with two elements which are 3x3 matrices each.

# Create an array.

```
a <- array(c('green','yellow'),dim = c(3,3,2))  
print(a)
```

When we execute the above code, it produces the following result

```
[,1] [,2] [,3]  
[1,] "green" "yellow" "green"  
[2,] "yellow" "green" "yellow"  
[3,] "green" "yellow" "green"
```

# 3. Factors

Very often, data falls into a limited number of categories. For example, humans are either male or female. In R, categorical data is stored in factors. Given the importance of these factors in data analysis, you should start learning how to create, subset and compare them now!

# What's a factor and why would you use it?

There are two types of categorical variables: a **nominal categorical variable** and an **ordinal categorical variable**.

**A nominal variable** is a categorical variable without an implied order. This means that it is impossible to say that 'one is worth more than the other'. For example, think of the categorical variable **animals\_vector** with the categories **"Elephant"**, **"Giraffe"**, **"Donkey"** and **"Horse"**. Here, it is impossible to say that one stands above or below the other. (Note that some of you might disagree ;-)).

In contrast, **ordinal variables** do have a natural ordering. Consider for example the categorical variable **temperature\_vector** with the categories: **"Low"**, **"Medium"** and **"High"**. Here it is obvious that "Medium" stands above "Low", and "High" stands above "Medium".

The term factor refers to a statistical data type used to store categorical variables. The difference between a categorical variable and a continuous variable is that a categorical variable can belong to a **limited number of categories**. A continuous variable, on the other hand, can correspond to an infinite number of values.

It is important that R knows whether it is dealing with a continuous or a categorical variable, as the statistical models you will develop in the future treat both types differently. (You will see later why this is the case.)

A good example of a categorical variable is the variable '**Gender**'. A human individual can either be "Male" or "Female", making abstraction of inter-sexes. So here "Male" and "Female" are, in a simplified sense, the two values of the categorical variable "Gender", and every observation can be assigned to either the value "Male" or "Female".

**Exercise (1):** Assign to variable theory the value "factors for categorical variables".

**Answer**

```
1 # Assign to the variable theory what this chapter is about!  
2 theory <- "factors for categorical variables"
```

## you use it?

To create factors in R, you make use of the function `factor()`. First thing that you have to do is create a vector that contains all the observations that belong to a limited number of categories. For example, `gender_vector` contains the sex of 5 different individuals:

```
gender_vector <- c("Male","Female","Female","Male","Male")
```

It is clear that there are two categories, or in R-terms 'factor levels', at work here: "Male" and "Female".

The function `factor()` will encode the vector as a factor:

```
factor_gender_vector <- factor(gender_vector)
```



## Exercise (2):

- Convert the character vector `gender_vector` to a factor with `factor()` and assign the result to `factor_gender_vector`
- Print out `factor_gender_vector` and assert that R prints out the factor levels below the actual values.

```
1 # Gender vector
2 gender_vector <- c("Male", "Female", "Female", "Male", "Male")
3
4 # Convert gender_vector to a factor
5 factor_gender_vector <-
6
7 # Print out factor_gender_vector
```

script.R

```
1 # Gender vector
2 gender_vector <- c("Male", "Female", "Female", "Male", "Male")
3
4 # Convert gender_vector to a factor
5 factor_gender_vector <- factor(gender_vector)
6
7 # Print out factor_gender_vector
8 factor_gender_vector
```

**Exercise (3):** Click 'Submit Answer' to check how R constructs and prints nominal and ordinal variables. Do not worry if you do not understand all the code just yet, we will get to that.

```
1. # Animals
2. animals_vector <- c("Elephant", "Giraffe", "Donkey", "Horse")
3. factor_animals_vector <- factor(animals_vector)
4. factor_animals_vector
5.
6. # Temperature
7. temperature_vector <- c("High", "Low", "High", "Low", "Medium")
8. factor_temperature_vector <- factor(temperature_vector, order = TRUE, levels = c("Low", "Medium", "High"))
9. factor_temperature_vector
```

# Factor levels

**When you first get a data set, you will often notice that it contains factors with specific factor levels.** However, sometimes you will want to change the names of these levels for clarity or other reasons. R allows you to do this with the function `levels()`:

```
levels(factor_vector) <- c("name1", "name2",...)
```

A good illustration is the raw data that is provided to you by a survey. A standard question for every questionnaire is the gender of the respondent. You remember from the previous question that this is a factor and when performing the questionnaire on the streets its levels are often coded as "M" and "F".

```
survey_vector <- c("M", "F", "F", "M", "M")
```

Next, when you want to start your data analysis, your main concern is to keep a nice overview of all the variables and what they mean. At that point, you will often want to change the factor levels to "Male" and "Female" instead of "M" and "F" to make your life easier.

Watch out: the order with which you assign the levels is important. If you type `levels(factor_survey_vector)`, you'll see that it **outputs [1] "F" "M"**. If you don't specify the levels of the factor when creating the vector, R will automatically assign them alphabetically. To correctly map "F" to "Female" and "M" to "Male", the levels should be set to `c("Female", "Male")`, in this order.

## Exercise (4):

- Check out the code that builds a factor vector from `survey_vector`. You should use `factor_survey_vector` in the next instruction.
- Change the factor levels of `factor_survey_vector` to `c("Female", "Male")`. Mind the order of the vector elements here.

```
1. # Code to build factor_survey_vector
2. survey_vector <- c("M", "F", "F", "M", "M")
3. factor_survey_vector <- factor(survey_vector)
4.
5. # Specify the levels of factor_survey_vector
6. levels(factor_survey_vector) <-
7.
8. factor_survey_vector
```

script.R

```
1 # Code to build factor_survey_vector
2 survey_vector <- c("M", "F", "F", "M", "M")
3 factor_survey_vector <- factor(survey_vector)
4
5 # Specify the levels of factor_survey_vector
6 levels(factor_survey_vector) <- c("Female", "Male")
7
8 factor_survey_vector
9 |
```

# Summarizing a factor

After finishing this Part, one of your favorite functions in R will be **summary()**. This will give you a quick overview of the contents of a variable:

**summary(my\_var)**

Going back to our survey, you would like to know how many "Male" responses you have in your study, and how many "Female" responses. The `summary()` function gives you the answer to this question.

**Exercise (5):** Ask a summary() of the survey\_vector and factor\_survey\_vector. Interpret the results of both vectors. Are they both equally useful in this case?

```
1. # Build factor_survey_vector with clean levels
2. survey_vector <- c("M", "F", "F", "M", "M")
3. factor_survey_vector <- factor(survey_vector)
4. levels(factor_survey_vector) <- c("Female", "Male")
5. factor_survey_vector
6.
7. # Generate summary for survey_vector
8.
9.
10. # Generate summary for factor_survey_vector
```

script.R

```
1 # Build factor_survey_vector with clean levels
2 survey_vector <- c("M", "F", "F", "M", "M")
3 factor_survey_vector <- factor(survey_vector)
4 levels(factor_survey_vector) <- c("Female", "Male")
5 factor_survey_vector
6
7 # Generate summary for survey_vector
8 summary(survey_vector)
9
10 # Generate summary for factor_survey_vector
11 summary(factor_survey_vector)
```

# More example about Factors

Factors are the r-objects which are created using a vector. It stores the vector along with the distinct values of the elements in the vector as labels. The labels are always character irrespective of whether it is numeric or character or Boolean etc. in the input vector. They are useful in statistical modelling.

Factors are created using the **factor()** function. The **nlevels** functions gives the count of levels.

```
# Create a vector.  
apple_colors <- c('green','green','yellow','red','red','red','green')  
  
# Create a factor object.  
factor_apple <- factor(apple_colors)  
  
# Print the factor.  
print(factor_apple)  
print(nlevels(factor_apple))
```

When we execute the above code, it produces the following result

```
[1] green green yellow red red red green  
Levels: green red yellow  
[1] 3
```

# 4.Data frames

Most data sets you will be working with will be stored as data frames. By the end of this lecture focused on R basics, you will be able to create a data frame, select interesting parts of a data frame and order a data frame according to certain variables.



# What's a data frame?

**Data Frame:** This is the most commonly used member of data types family. It is used to store **tabular data**. It is different from matrix. In a matrix, every element must have same class. But, in a data frame, you can put list of vectors containing different classes. This means, every column of a data frame acts like a list. Every time you will read data in R, it will be stored in the form of a data frame. Hence, it is important to understand the majorly used commands on data frame:

# What's a data frame?

## Simple example:

```
> df <- data.frame(name = c("ash", "jane", "paul", "mark"),  
score = c(67, 56, 87, 91))
```

```
> df
```

```
name score
```

```
1 ash 67
```

```
2 jane 56
```

```
3 paul 87
```

```
4 mark 91
```

```
> dim(df)
```

```
[1] 4 2
```

# What's a data frame?

All the elements that you put in a matrix should be of the same type. Back then, your data set on Star Wars only contained numeric elements.

When doing a market research survey, however, you often have questions such as:

- 'Are you married?' or 'yes/no' questions (logical)
- 'How old are you?' (numeric)
- 'What is your opinion on this product?' or other 'open-ended' questions (character)
- ...

The output, namely the respondents' answers to the questions formulated above, is a data set of different data types. You will often find yourself working with data sets that contain different data types instead of only one.

**A data frame** has the variables of a data set as columns and the observations as rows. This will be a familiar concept for those coming from different statistical software packages such as SAS or SPSS.

**Exercise (6):** The data from the built-in example data frame **mtcars** will be printed to the console.

1. # Print out built-in R data frame
2. mtcars

### Console Output

```
> mtcars
      mpg  cyl  disp  hp drat   wt  qsec vs  am  gear  carb
Mazda RX4    21.0   6 160.0 110 3.90 2.620 16.46 0  1    4    4
Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02 0  1    4    4
Datsun 710    22.8   4 108.0  93 3.85 2.320 18.61 1  1    4    1
Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44 1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02 0  0    3    2
Valiant      18.1   6 225.0 105 2.76 3.460 20.22 1  0    3    1
Duster 360    14.3   8 360.0 245 3.21 3.570 15.84 0  0    3    4
Merc 240D     24.4   4 146.7  62 3.69 3.190 20.00 1  0    4    2
Merc 230      22.8   4 140.8  95 3.92 3.150 22.90 1  0    4    2
Merc 280      19.2   6 167.6 123 3.92 3.440 18.30 1  0    4    4
Merc 280C     17.8   6 167.6 123 3.92 3.440 18.90 1  0    4    4
Merc 450SE    16.4   8 275.8 180 3.07 4.070 17.40 0  0    3    3
Merc 450SL    17.3   8 275.8 180 3.07 3.730 17.60 0  0    3    3
Merc 450SLC   15.2   8 275.8 180 3.07 3.780 18.00 0  0    3    3
Cadillac Fleetwood 10.4   8 472.0 205 2.93 5.250 17.98 0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82 0  0    3    4
Chrysler Imperial 14.7   8 440.0 230 3.23 5.345 17.42 0  0    3    4
Fiat 128      32.4   4  78.7  66 4.08 2.200 19.47 1  1    4    1
Honda Civic   30.4   4  75.7  52 4.93 1.615 18.52 1  1    4    2
Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90 1  1    4    1
Toyota Corona 21.5   4 120.1  97 3.70 2.465 20.01 1  0    3    1
Dodge Challenger 15.5   8 318.0 150 2.76 3.520 16.87 0  0    3    2
AMC Javelin   15.2   8 304.0 150 3.15 3.435 17.30 0  0    3    2
Camaro Z28    13.3   8 350.0 245 3.73 3.840 15.41 0  0    3    4
Pontiac Firebird 19.2   8 400.0 175 3.08 3.845 17.05 0  0    3    2
Fiat X1-9     27.3   4  79.0  66 4.08 1.935 18.90 1  1    4    1
Porsche 914-2 26.0   4 120.3  91 4.43 2.140 16.70 0  1    5    2
Lotus Europa  30.4   4  95.1 113 3.77 1.513 16.90 1  1    5    2
Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.50 0  1    5    4
Ferrari Dino  19.7   6 145.0 175 3.62 2.770 15.50 0  1    5    6
Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.60 0  1    5    8
Volvo 142E    21.4   4 121.0 109 4.11 2.780 18.60 1  1    4    2
>
```

## Quick, have a look at your data set

Wow, that is a lot of cars!

Working with large data sets is not uncommon in data analysis. When you work with (extremely) large data sets and data frames, your first task as a data analyst is to develop a clear understanding of its structure and main elements. Therefore, it is often useful to show only a small part of the entire data set. So how to do this in R? Well, the function `head()` enables you to show the first observations of a data frame. Similarly, the function `tail()` prints out the last observations in your data set.

Both `head()` and `tail()` print a top line called the '`header`', which contains the names of the different variables in your data set.

**Exercise (1):** Call `head()` on the `mtcars` data set to have a look at the header and the first observations.

1. `# Call head() on mtcars`
2. `head(mtcars)`

## Have a look at the structure

Another method that is often used to get a rapid overview of your data is the function `str()`. The function `str()` shows you the structure of your data set. For a data frame it tells you:

- The total number of observations (e.g. 32 car types)
- The total number of variables (e.g. 11 car features)
- A full list of the variables names (e.g. `mpg`, `cyl` ... )
- The data type of each variable (e.g. `num`)
- The first observations

Applying the `str()` function will often be the first thing that you do when receiving a new data set or data frame. It is a great way to get more insight in your data set before diving into the real analysis.

**Exercise (2):** Investigate the structure of `mtcars`. Make sure that you see the same numbers, variables and data types as mentioned above.

```
# Investigate the structure of mtcars  
str(mtcars)
```

# Creating a data frame

Since using built-in data sets is not even half the fun of creating your own data sets, the rest of this section is based on your personally developed data set. Put your jet pack on because it is time for some space exploration!

As a first goal, **you want to construct a data frame that describes the main characteristics of eight planets in our solar system**. According to your good friend Buzz, the main features of a planet are:

- The type of planet (Terrestrial or Gas Giant).
- The planet's diameter relative to the diameter of the Earth.
- The planet's rotation across the sun relative to that of the Earth.
- If the planet has rings or not (TRUE or FALSE).

After doing some high-quality research on Wikipedia, you feel confident enough to create the necessary **vectors: name, type, diameter, rotation and rings**; these vectors have already been coded up on the right. The first element in each of these vectors correspond to the first observation.

You construct a data frame with the **data.frame()** function. As arguments, you pass the vectors from before: they will become the different columns of your data frame. Because every column has the same length, the vectors you pass should also have the same length. But don't forget that it is possible (and likely) that they contain different types of data.

**Exercise (1):** Use the function `data.frame()` to construct a data frame. Pass the vectors `name`, `type`, `diameter`, `rotation` and `rings` as arguments to `data.frame()`, in this order. Call the resulting data frame `planets_df`.

```
1. # Definition of vectors
2. planets <- c("Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune")
3. type <- c("Terrestrial planet", "Terrestrial planet", "Terrestrial planet",
4.           "Terrestrial planet", "Gas giant", "Gas giant", "Gas giant", "Gas giant")
5. diameter <- c(0.382, 0.949, 1, 0.532, 11.209, 9.449, 4.007, 3.883)
6. rotation <- c(58.64, -243.02, 1, 1.03, 0.41, 0.43, -0.72, 0.67)
7. rings <- c(FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE)

8. # Create a data frame: planets_df
9. planets_df <- data.frame(planets, type, diameter, rotation, rings)
```

```
1 # Definition of vectors
2 name <- c("Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune")
3 type <- c("Terrestrial planet", "Terrestrial planet", "Terrestrial planet",
4           "Terrestrial planet", "Gas giant", "Gas giant", "Gas giant", "Gas giant")
5 diameter <- c(0.382, 0.949, 1, 0.532, 11.209, 9.449, 4.007, 3.883)
6 rotation <- c(58.64, -243.02, 1, 1.03, 0.41, 0.43, -0.72, 0.67)
7 rings <- c(FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE)
8
9 # Create a data frame from the vectors
10 planets_df <- data.frame(name, type, diameter, rotation, rings)
11
```



The `planets_df` data frame should have 8 observations and 5 variables. It has been made available in the workspace, so you can directly use it.

**Exercise (4):** Use `str()` to investigate the structure of the new `planets_df` variable.

1. `# Check the structure of planets_df`
2. `str(planets_df)`

# Selection of data frame elements

Similar to vectors and matrices, you select elements from a data frame with the help of square brackets [ ]. By using a comma, you can indicate what to select from the rows and the columns respectively. For example:

- `my_df[1,2]` selects the value at the first row and select element in `my_df`.
- `my_df[1:3,2:4]` selects rows 1, 2, 3 and columns 2, 3, 4 in `my_df`.

Sometimes you want to select all elements of a row or column. For example, `my_df[1, ]` selects all elements of the first row. Let us now apply this technique on `planets_df`!

**Exercise (4):** From `planets_df`, select the diameter of Mercury: this is the value at the first row and the third column. Simply print out the result.

From `planets_df`, select all data on Mars (the fourth row). Simply print out the result.

script.R

```
1 # The planets_df data frame from the previous exercise is pre-loaded
2
3 # Print out diameter of Mercury (row 1, column 3)
4 planets_df[1,3]
5
6 # Print out data for Mars (entire fourth row)
7 planets_df[4,]
8
```

# More example of Data Frames

Data frames are tabular data objects. Unlike a matrix in data frame each column can contain different modes of data. The first column can be numeric while the second column can be character and third column can be logical. It is a list of vectors of equal length.

Data Frames are created using the **data.frame()** function.

```
# Create the data frame.  
BMI <- data.frame(  
  gender = c("Male", "Male", "Female"),  
  height = c(152, 171.5, 165),  
  weight = c(81, 93, 78),  
  Age = c(42, 38, 26) )  
print(BMI)
```

When we execute the above code, it produces the following result

1	Male	152.0	81	42
2	Male	171.5	93	38
3	Female	165.0	78	26

# Summary

You can save data in R with five different objects, which let you store different types of values in different types of relationships, as in Figure below. Of these objects, data frames are by far the most useful for data science. Data frames store one of the most common forms of data used in data science, tabular data.

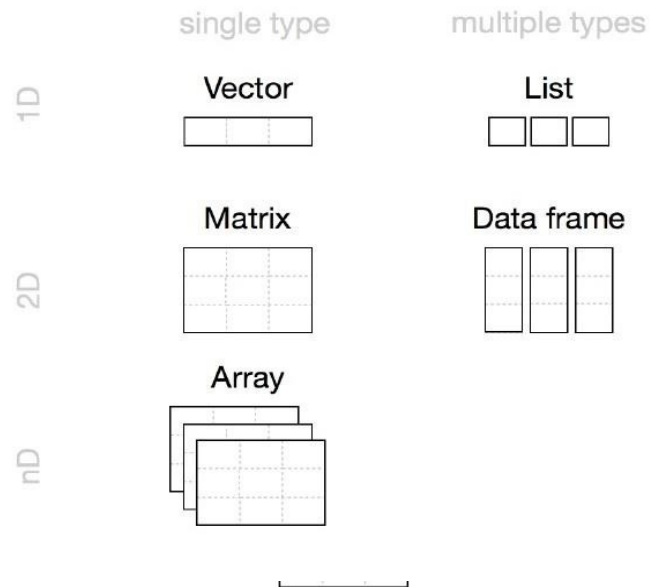


Figure 1. R's most common data structures are vectors, matrices, arrays, lists, and data frames.

# Functions

A function is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.

An R function is created by using the keyword **function**. The basic syntax of an R function definition is as follows

```
function_name <- function(arg_1, arg_2, ...) { Function body }
```

## Function Components

The different parts of a function are

- **Function Name** – This is the actual name of the function. It is stored in R environment as an object with this name.
- **Arguments** – An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.
- **Function Body** – The function body contains a collection of statements that defines what the function does.
- **Return Value** – The return value of a function is the last expression in the function body to be evaluated.

```
# Create a sequence of numbers from 32 to 44.  
print(seq(32,44))
```

```
# Find mean of numbers from 25 to 82.  
print(mean(25:82))
```

```
# Find sum of numbers from 41 to 68.  
print(sum(41:68))
```

When we execute the above code, it produces the following result

```
> # Create a sequence of numbers from 32 to 44.  
> print(seq(32,44))  
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44  
>  
> # Find mean of numbers from 25 to 82.  
> print(mean(25:82))  
[1] 53.5  
>  
> # Find sum of numbers from 41 to 68.  
> print(sum(41:68))  
[1] 1526  
>
```



Another useful operations are:

Operator or Function	Description
<code>A * B</code>	Element-wise multiplication
<code>A %% B</code>	Matrix multiplication
<code>A %o% B</code>	Outer product. $AB'$
<code>t(A)</code>	Transpose
<code>diag(x)</code>	Creates diagonal matrix with elements of x in the principal diagonal
<code>solve(A, b)</code>	Returns vector x in the equation $b = Ax$ (i.e., $A^{-1}b$ )
<code>solve(A)</code>	Inverse of A where A is a square matrix.
<code>cbind(A,B,...)</code>	Combine matrices(vectors) horizontally. Returns a matrix.
<code>rbind(A,B,...)</code>	Combine matrices(vectors) vertically. Returns a matrix.
<code>rowMeans(A)</code>	Returns vector of row means.
<code>rowSums(A)</code>	Returns vector of row sums.
<code>colMeans(A)</code>	Returns vector of column means.
<code>colSums(A)</code>	Returns vector of column means.

# Your first script

Fire up RStudio (or whatever IDE you use) and type the following two lines into the text editor. Then execute them, which can be done using a keyboard shortcut <https://support.rstudio.com/hc/en-us/articles/200711853-Keyboards-Shortcuts> — Ctrl+Enter (Windows) or Command+Enter (mac) in RStudio.

# **Applied Basic Econometrics using R:**

## **Simple Linear Regression**

# Simple Linear Regression

**Regression analysis** is a very widely used statistical tool to establish a relationship model between two variables.

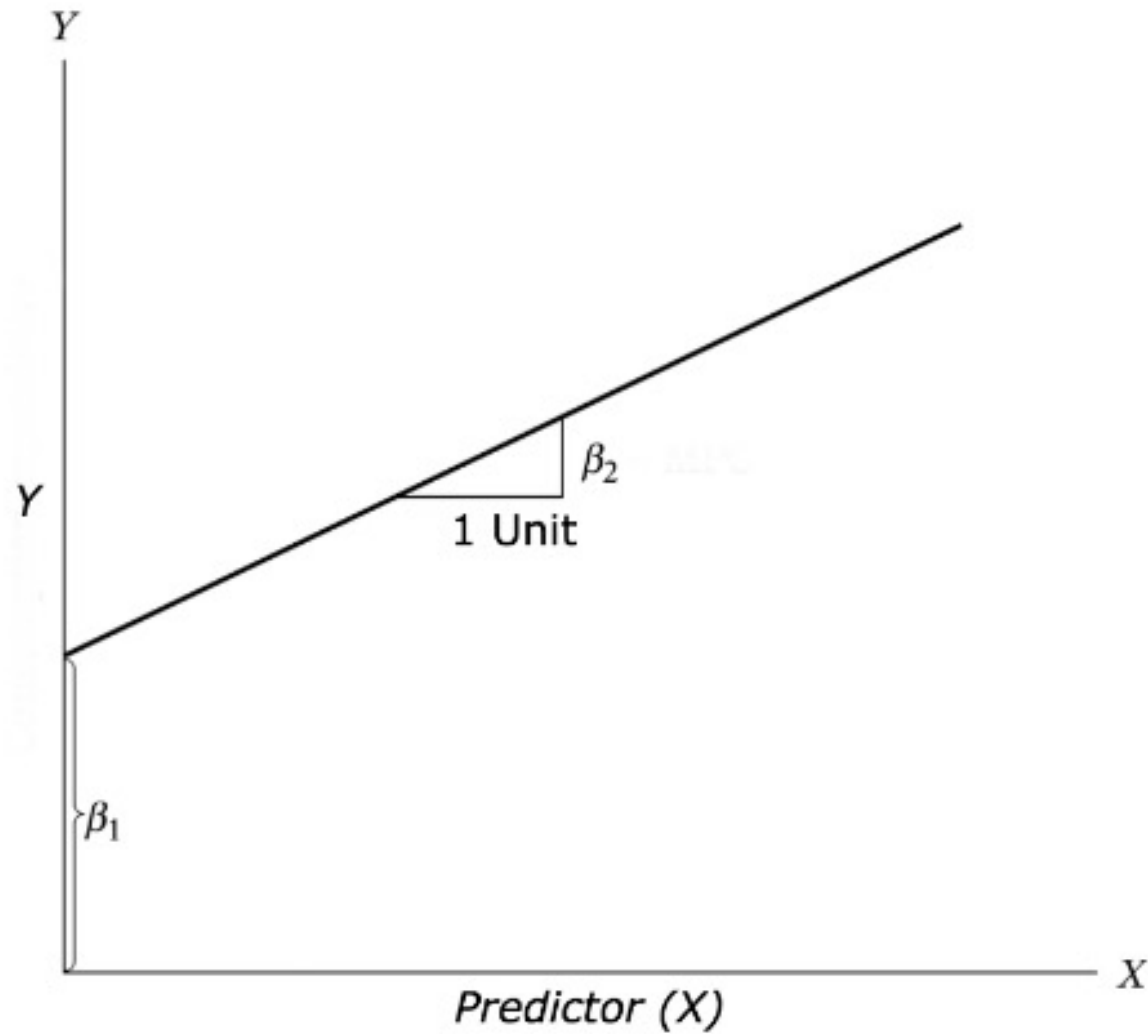
Regression is to build a function of **independent variables** (also known as predictors whose value is gathered through experiments) to predict a **dependent variable** (also called response whose value is derived from the predictor variable). For example, banks assess the risk of home-loan applicants based on their age, income, expenses, occupation, number of dependents, total credit limit, etc.

The aim of linear regression is to model a continuous variable  $Y$  as a mathematical function of one or more  $X$  variable(s), so that we can use this regression model to predict the  $Y$  when only the  $X$  is known. This mathematical equation can be generalized as follows:

$$Y = \beta_1 + \beta_2 X + \epsilon$$

where,  $\beta_1$  is the intercept and  $\beta_2$  is the slope. Collectively, they are called regression coefficients.  $\epsilon$  is the error term, the part of  $Y$  the regression model is unable to explain.

$$Y = \beta_1 + \beta_2 X + \epsilon$$



# Example Problem

For this analysis, we will use the **cars dataset** that comes with R by default. cars is a standard built-in dataset, that makes it convenient to demonstrate linear regression in a simple and easy to understand fashion.

You can access this dataset simply by typing in cars in your R console. You will find that it consists of 50 observations(rows) and 2 variables (columns) – dist and speed. Lets print out the first six observations here..

# Cars Dataset

```
> head(cars) # display the first 6
```

```
observations
```

```
  speed dist
```

```
1     4     2
```

```
2     4    10
```

```
3     7     4
```

```
4     7    22
```

```
5     8    16
```

```
6     9    10
```



# Graphical Analysis:

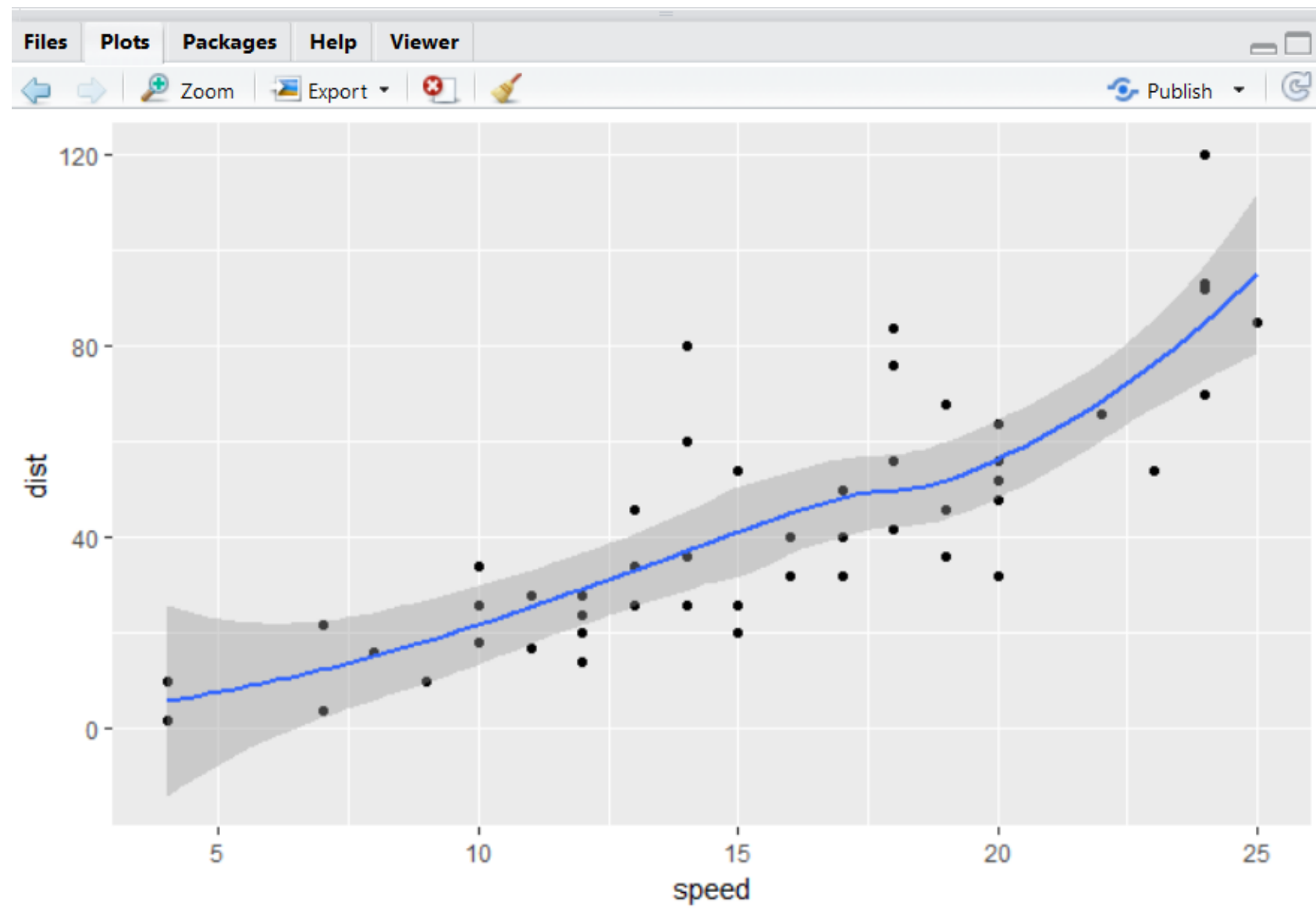
The aim of this exercise is to build a simple regression model that we can use to predict Distance (dist) by establishing a statistically significant linear relationship with Speed (speed). But before jumping in to the syntax, let's try to understand these variables graphically. Typically, for each of the independent variables (predictors), the following plots are drawn to visualize the following behavior:

- 1. Scatter plot:** Visualize the linear relationship between the predictor and response
- 2. Box plot:** To spot any outlier observations in the variable. Having outliers in your predictor can drastically affect the predictions as they can easily affect the direction/slope of the line of best fit.
- 3. Density plot:** To see the distribution of the predictor variable. Ideally, a close to normal distribution (a bell shaped curve), without being skewed to the left or right is preferred. Let us see how to make each one of them.

# Scatter Plot

Scatter plots can help visualize any linear relationships between the dependent (response) variable and independent (predictor) variables. Ideally, if you are having multiple predictor variables, a scatter plot is drawn for each one of them against the response, along with the line of best fit as seen below.

```
library(ggplot2)  
ggplot(cars, aes(x= speed , y=dist))+ geom_point()+ stat_smooth()
```

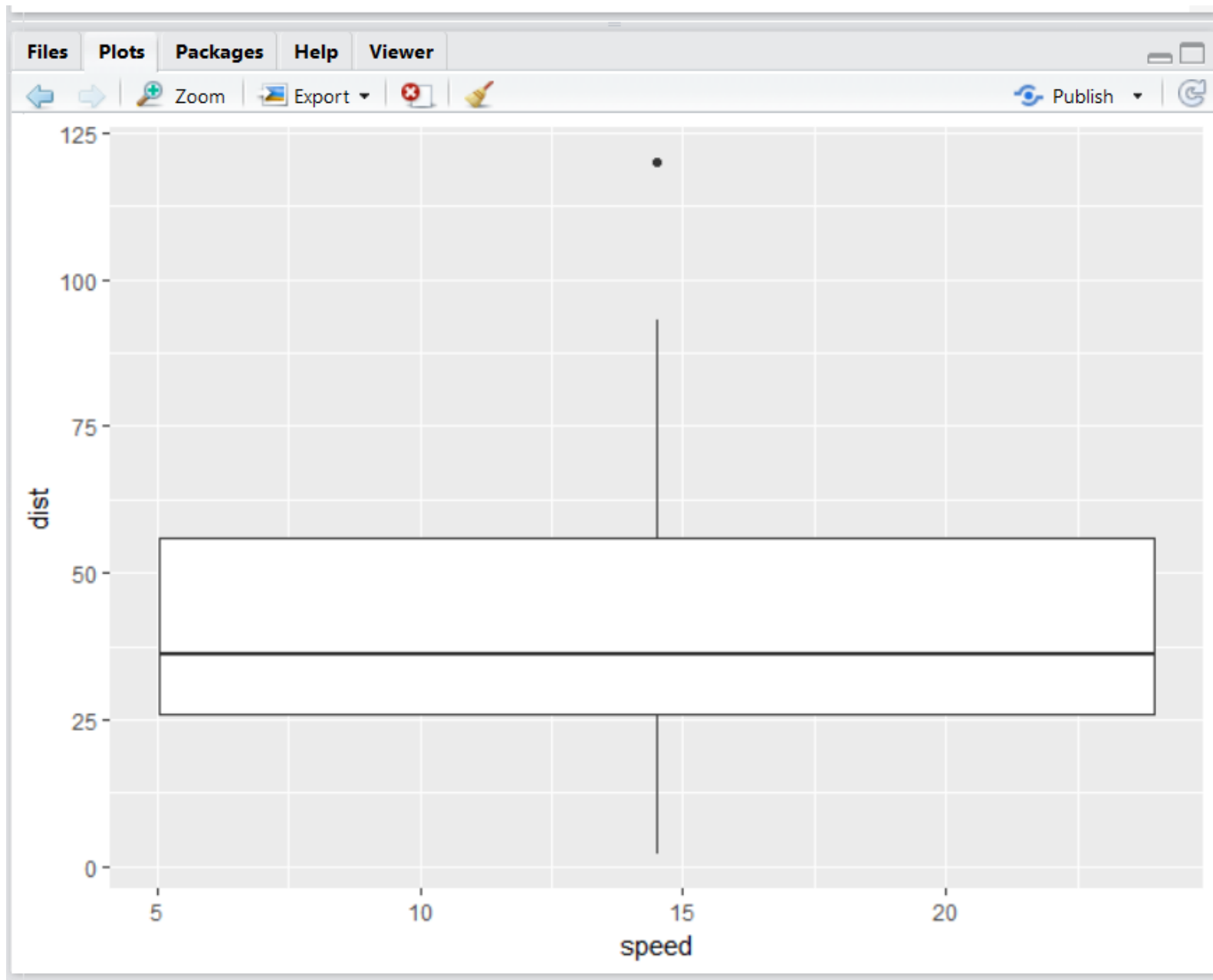


The scatter plot along with the smoothing line above suggests a linearly increasing relationship between the 'dist' and 'speed' variables. This is a good thing, because, one of the underlying assumptions in linear regression is that the relationship between the response and predictor variables is linear and additive.

# BoxPlot – Check for outliers

Generally, any datapoint that lies outside the  $1.5 * \text{IQR}$  interquartile-range ( $1.5 * \text{IQR}$ ) is considered an outlier, where, IQR is calculated as the distance between the 25th percentile and 75th percentile values for that variable.

```
library(ggplot2)  
ggplot(cars, aes(x= speed , y=dist)) + geom_boxplot()
```



**Outlier rows = 120**

# Correlation

Correlation is a statistical measure that suggests the level of linear dependence between two variables, that occur in pair – just like what we have here in speed and dist. Correlation can take values between -1 to +1. If we observe for every instance where speed increases, the distance also increases along with it, then there is a high positive correlation between them and therefore the correlation between them will be closer to 1. The opposite is true for an inverse relationship, in which case, the correlation between the variables will be close to -1. A value closer to 0 suggests a weak relationship between the variables. A low correlation ( $-0.2 < x < 0.2$ ) probably suggests that much of variation of the response variable (Y) is unexplained by the predictor (X), in which case, we should probably look for better explanatory variables. We can measure this with the correlation coefficient

```
> with(cars, cor(speed, dist))  
[1] 0.8068949  
>
```

# Build Linear Model

Regression is a special case of a linear model. The R function for fitting a linear model is **lm()** and we can use it like this. **The function coef()** returns the estimated coefficients of the fitted linear model.

The **lm()** function takes in two main arguments, namely: **1. Formula** **2. Data**. The data is typically a **data.frame** and the formula is a object of class formula. But the most common convention is to write out the formula directly in place of the argument as written below.

```
> fit = lm(dist ~ speed, data = cars)
> coef(fit)
(Intercept)      speed
-17.579095      3.932409
>
```



Now that we have built the linear model, we also have established the relationship between the predictor (independent variables) and response (dependent variables) in the form of a mathematical formula for Distance (dist) as a function for speed. For the above output, you can notice the 'Coefficients' part having two components: Intercept: -17.579, speed: 3.932 These are also called the beta coefficients. In other words,

$$\text{dist} = \text{Intercept} + (\beta * \text{speed})$$
$$\Rightarrow \text{dist} = -17.579 + 3.932 * \text{speed}$$

# Linear Regression Diagnostics

Now the linear model is built and we have a formula that we can use to predict the dist value if a corresponding speed is known. Is this enough to actually use this model? NO! Before using a regression model, you have to ensure that it is statistically significant. How do you ensure this? Lets begin by printing the summary statistics for linearMod.

```
> summary(fit)
```

```
> summary(fit)
```

Call:

```
lm(formula = dist ~ speed, data = cars)
```

Residuals:

Min	1Q	Median	3Q	Max
-29.069	-9.525	-2.272	9.215	43.201

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	-17.5791	6.7584	-2.601	0.0123 *
speed	3.9324	0.4155	9.464	1.49e-12 ***

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

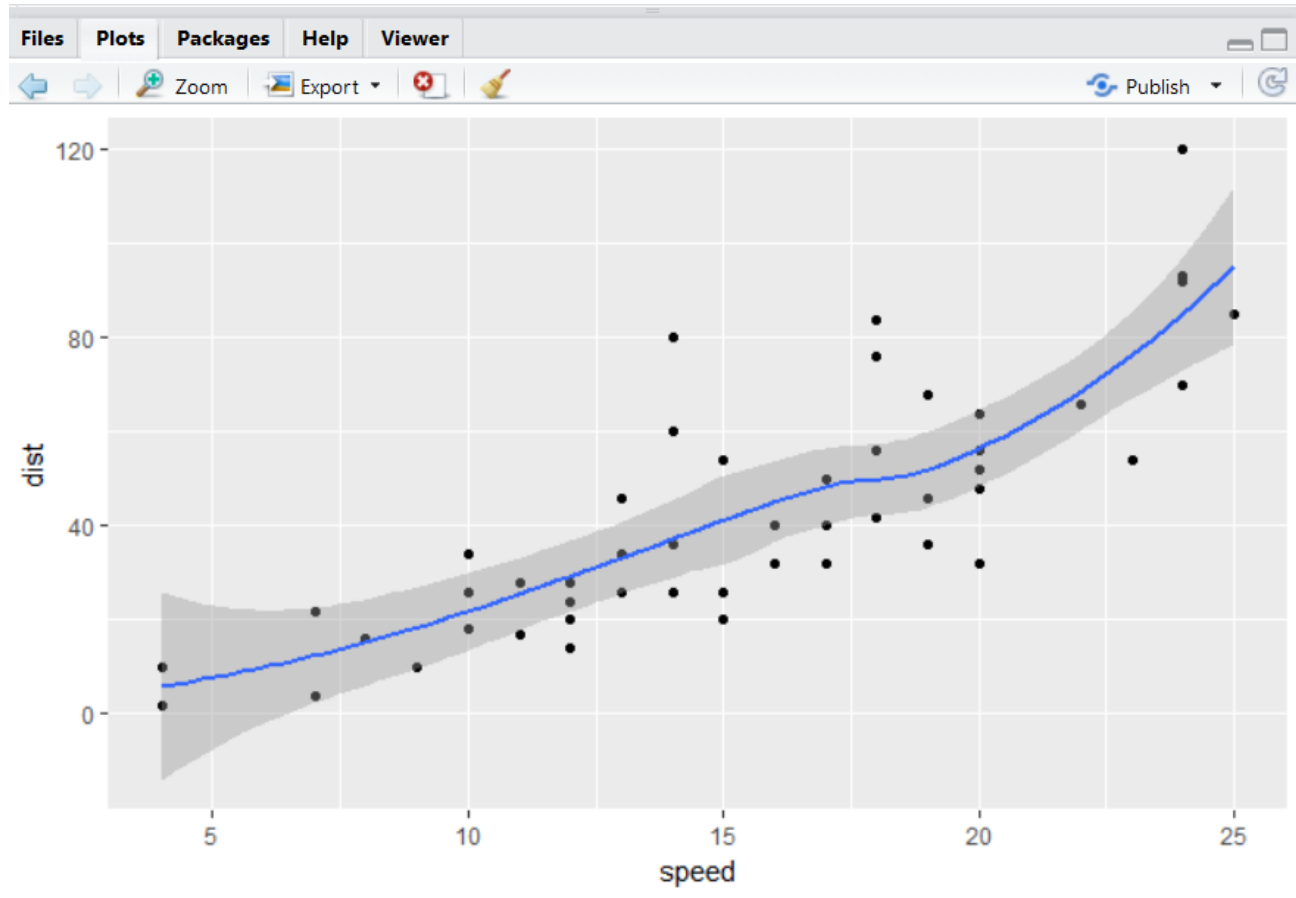
Residual standard error: 15.38 on 48 degrees of freedom

Multiple R-squared: 0.6511, Adjusted R-squared: 0.6438

F-statistic: 89.57 on 1 and 48 DF, p-value: 1.49e-12

We can add the line to the plot using `geom_smooth()`. The default behavior is to add a locally smooth regression line with a shaded gray area to represent pointwise 95 percent confidence regions for the true mean `dis` as a function of `speed`. We first show this plot and then the options to override the default and just plot the simple regression line.

```
library(ggplot2)
ggplot(cars, aes(x= speed , y=dist))+ geom_point()+ stat_smooth()
```



The scatter plot along with the smoothing line above suggests a linearly increasing relationship between the 'dist' and 'speed' variables. This is a good thing, because, one of the underlying assumptions in linear regression is that the relationship between the response and predictor variables is linear and additive.

## geom\_smooth: method="auto" and size of largest group is <1000, so using loess. Use 'method = x' to change the smoothing method.

```
ggplot(cars, aes(x=speed,y=dist)) + geom_point() + geom_smooth(method="lm",  
se=FALSE)
```

