

1.Introduction to Data Science

“Data Science is an interdisciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from noisy, structured and unstructured data, and apply knowledge from data across a broad range of application domains. Data science is related to data mining, machine learning and big data.”

➤ Tools required for building A Data Science Project

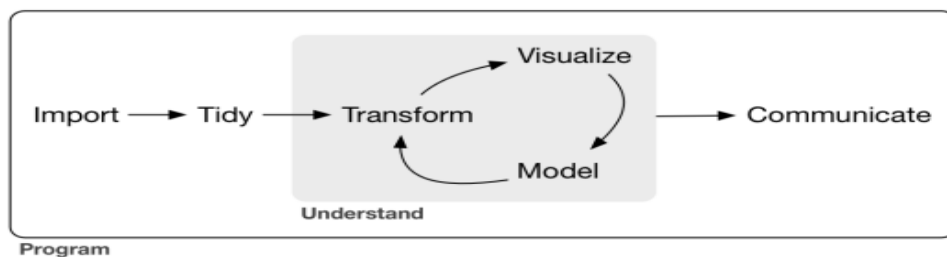


Figure 1: Tools required for building a Data Science Project

- **Importing data into R:** This typically means taking data stored in a file, database, or web API, and loading it into a data frame in R.
- **After importing data, tidy it:** Tidying your data means storing it in a consistent form that matches the semantics of the dataset with the way it is stored. In brief, when your data is tidy, each column is a variable, and each row is an observation.
- **Transformation:** Transformation includes narrowing in on observations of interest creating new variables that are functions of existing variables, and calculating a set of summary statistics. Together, tidying and transforming are called wrangling.
- **Engines of knowledge generation: visualization and modelling:** Visualization is a fundamentally human activity. A good visualization will show you things that you did not expect, or raise new questions about the data. A good visualization might also hint that you're asking the wrong question, or you need to collect different data. Models are complementary tools to visualization. Once you have made your questions sufficiently precise, you can use a model to answer them. Models are a fundamentally mathematical or computational tool, so they generally scale well.

- **Communication:** It doesn't matter how well your models and visualization have led you to understand the data unless you can also communicate your results to others.

2. Introduction to R Programming

The R Foundation describes R as “*a language and environment for statistical computing and graphics.*”

➤ Features of R in data science:

- **R is open-source software:** R is free and adaptable because it's an open-source software. R's open interfaces allow it to integrate with other applications and systems.
- **R is a programming language:** As a programming language, R provides objects, operators and functions that allow users to explore, model and visualize data.
- **R is used for data analysis:** R in data science is used to handle, store and analyze data. It can be used for data analysis and statistical modeling.
- **R is an environment for statistical analysis.** R has various statistical and graphical capabilities. The R Foundation notes that it can be used for classification, clustering, statistical tests and linear and nonlinear modeling.
- **R is a community:** R Project contributors include individuals who have suggested improvements, noted bugs and created add-on packages. While there are more than 20 official contributors, the R community extends to those using the open-source software on their own.

➤ Real-Time Applications of R

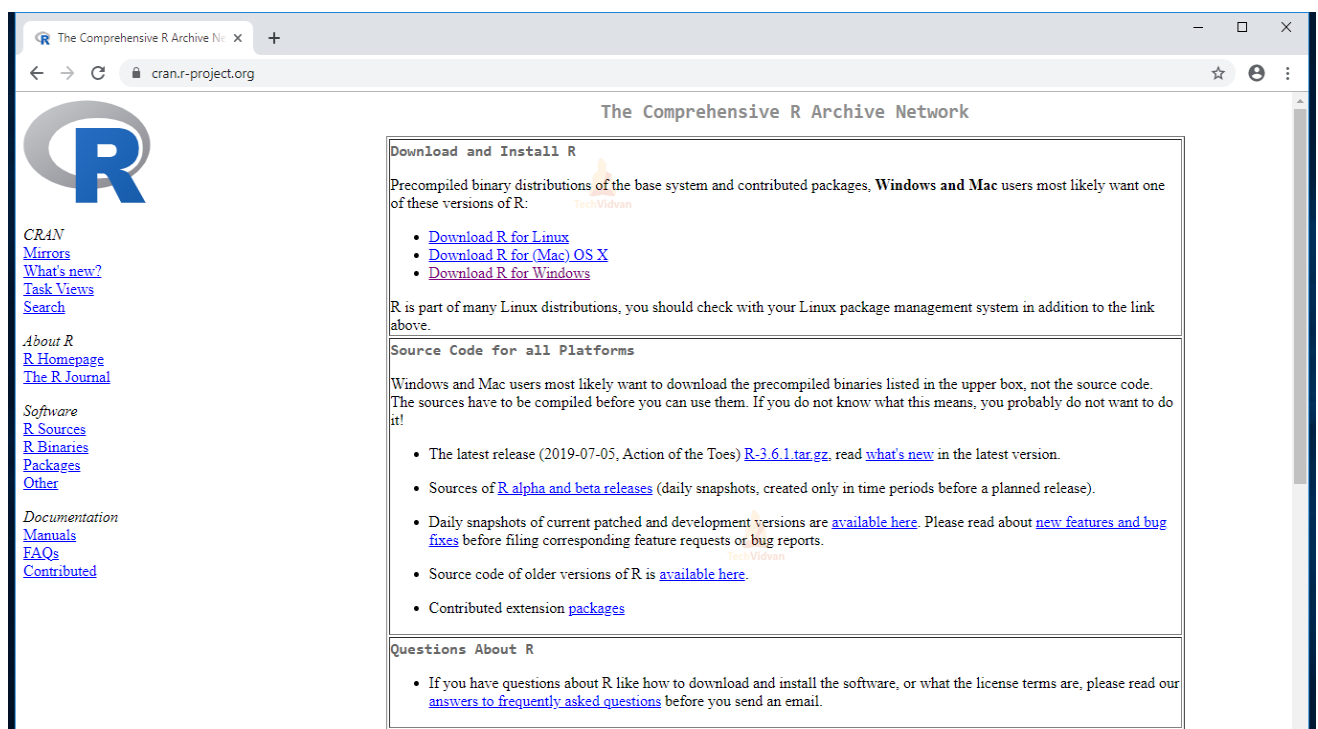
R for data science is used in industries such as banking, telecommunications and media. Few examples of data visualization in R through real-life projects are as follows:

- **T-Mobile:** The international communications company uses R to classify customer service texts so it can properly direct customers to an agent, Revolutions reports. T-Mobile even shared an open-source version of their messaging classification application programming interface on GitHub.
- **Twitter:** R can be used to perform text analysis of tweets. Text analytics and scraping of Twitter data is possible through the `twitteR` package.

- **Google Analytics:** R can be combined with Google Analytics data to complete statistical analysis and create clear data visualizations, according to Google Developers. Installing the RGoogleAnalytics package will enable these insights.
- **The Financial Times:** The Financial Times embraced R to create a data visualizations in its article, “Is Russia-Saudi Arabia the worst World Cup game ever?,” Revolutions reports. The visualization mapped every World Cup match since 1998 and was created using R and the ggplot2.
- **BBC:** Similarly, Revolutions explains how BBC uses data visualization in R to create graphics for its publications. BBC developed an R package and R cookbook to standardize their data visualization graphic creation process. Its cookbook is based on the bbplot package. BBC offers a six-week training for its data journalists to learn this process.

3. Installation of R on windows

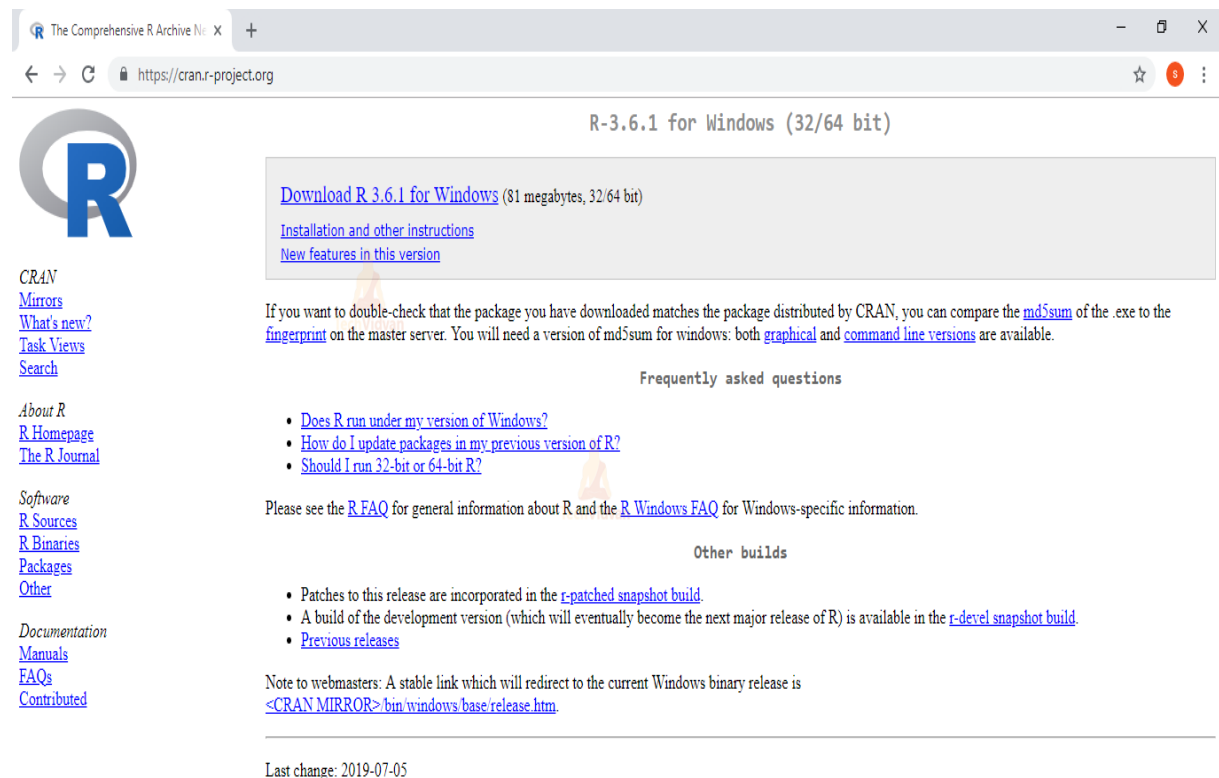
Step – 1: Go to CRAN R project website.



Step – 2: Click on the **Download R for Windows** link.

Step – 3: Click on the **base** subdirectory link or **install R for the first time** link.

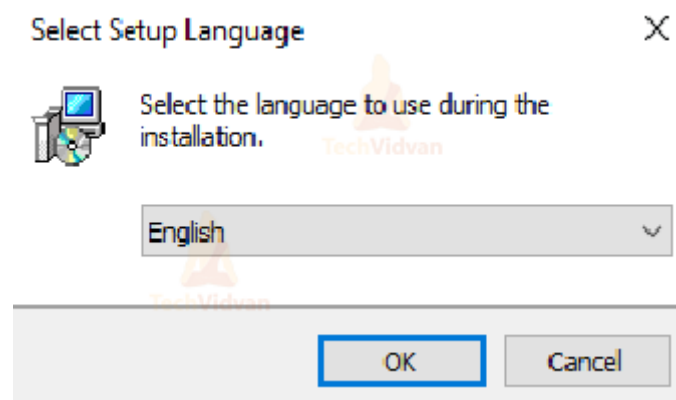
Step – 4: Click Download R X.X.X for Windows (X.X.X stand for the latest version of R. eg: 3.6.1) and save the executable .exe file.



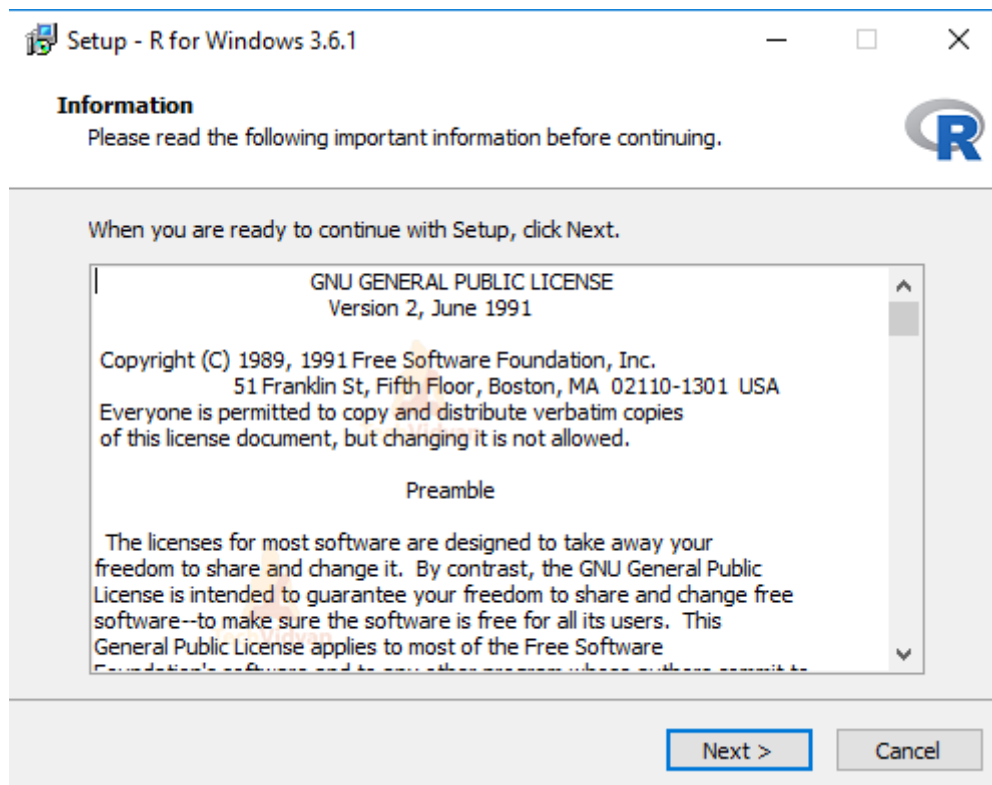
The screenshot shows the CRAN website for downloading R 3.6.1 for Windows (32/64 bit). The page includes the CRAN logo, navigation links (CRAN, Mirrors, What's new?, Task Views, Search, About R, R Homepage, The R Journal, Software, R Sources, R Binaries, Packages, Other, Documentation, Manuals, FAQs, Contributed), and a section for downloading R 3.6.1 for Windows (81 megabytes, 32/64 bit). The download link is highlighted. Below the download link, there are links for installation and other instructions, and new features in this version. A section for frequently asked questions is also present, with links to 'Does R run under my version of Windows?', 'How do I update packages in my previous version of R?', and 'Should I run 32-bit or 64-bit R?'. A note to webmasters is at the bottom, stating that a stable link will redirect to the current Windows binary release is <CRAN MIRROR> bin/windows/base/release.htm. The last change is noted as 2019-07-05.

Step – 5: Run the .exe file and follow the installation instructions.

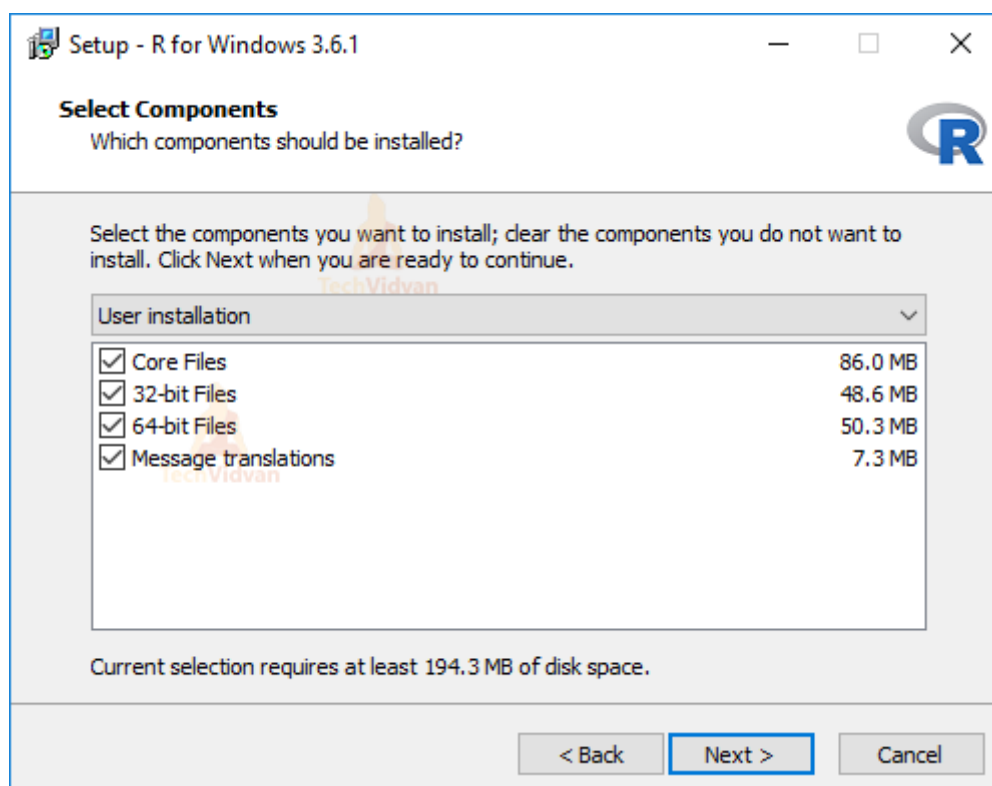
5.a. Select the desired language and then click Next.



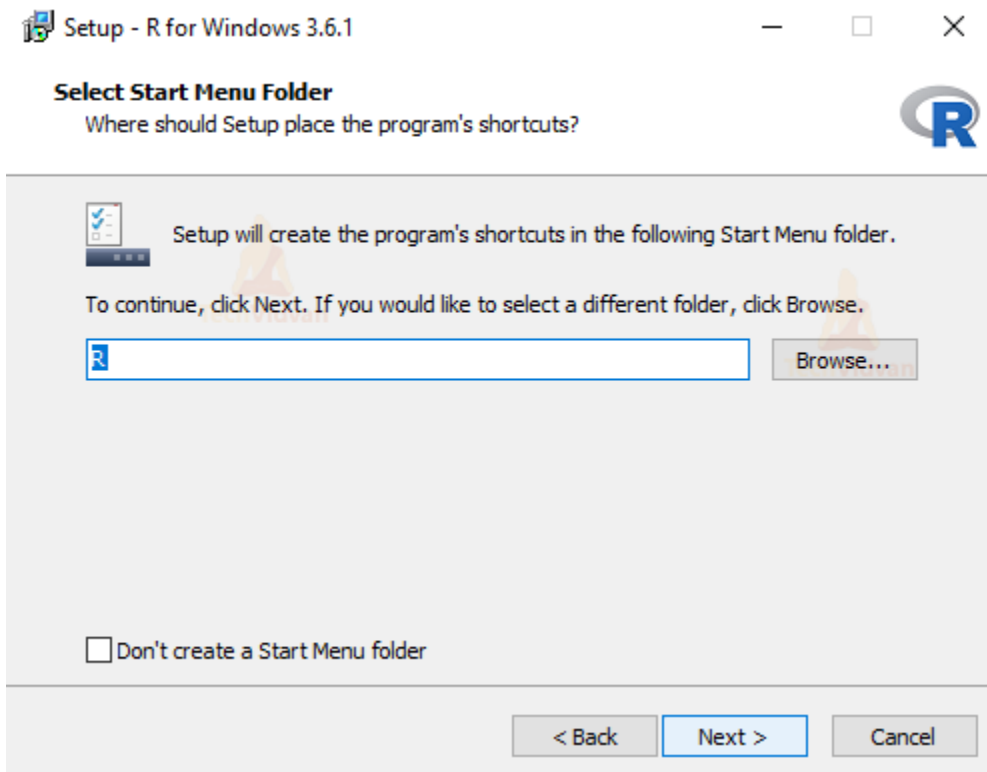
5.b. Read the license agreement and click Next.



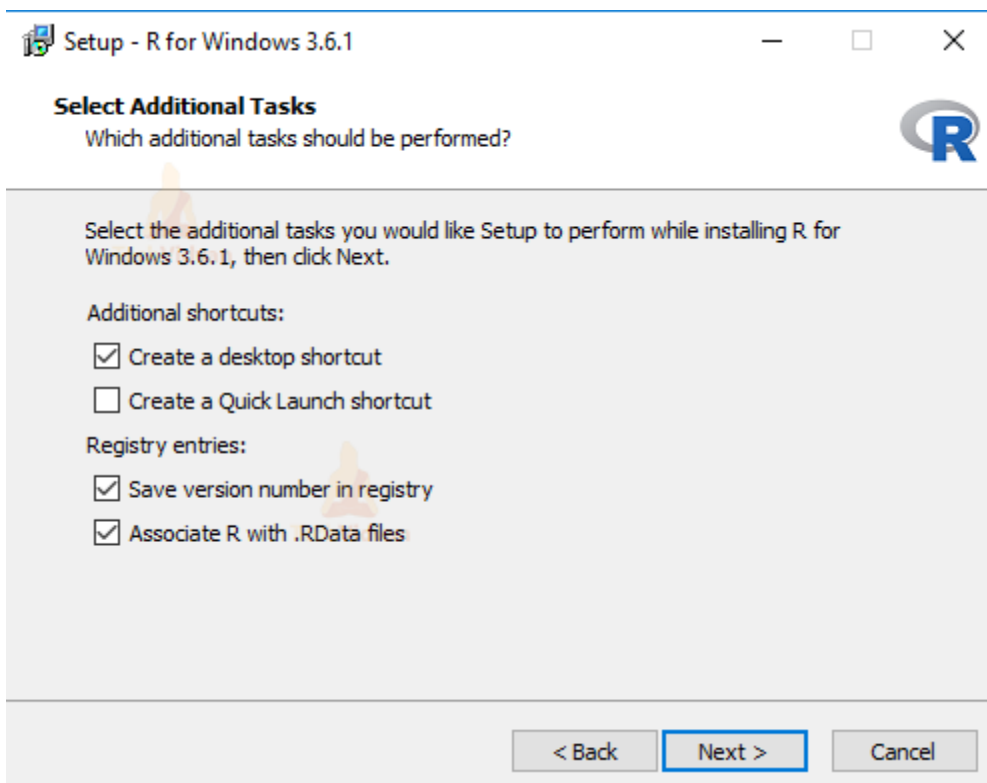
5.c. Select the components you wish to install (it is recommended to install all the components). Click **Next**.



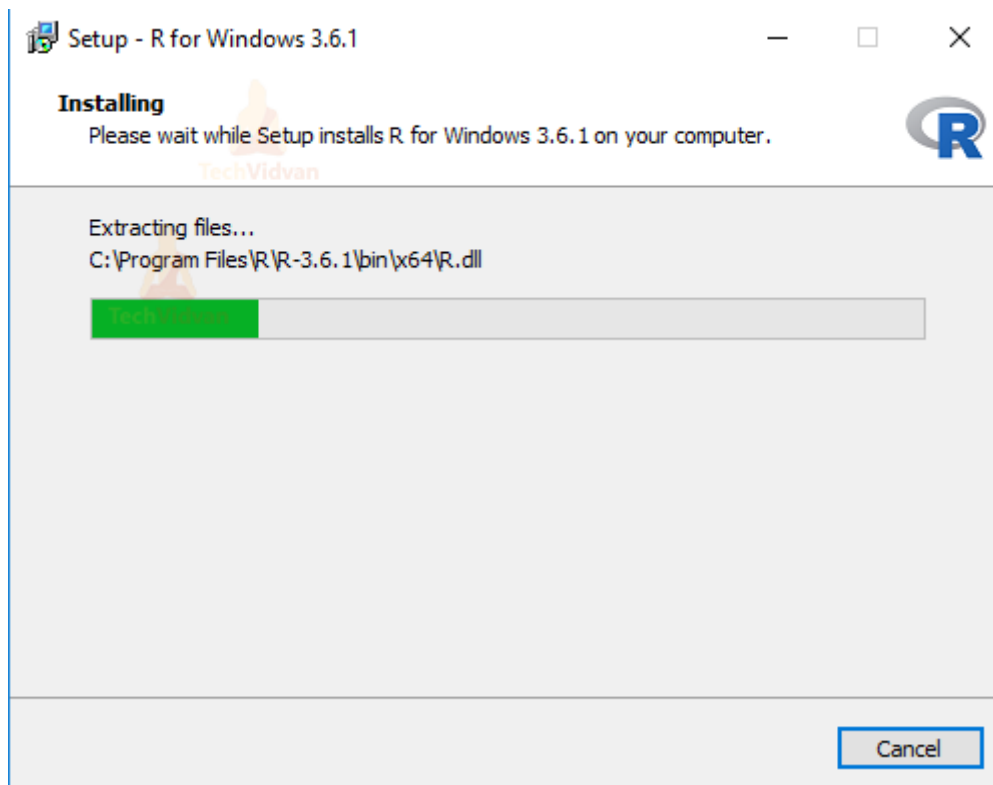
5.d. Enter/browse the folder/path you wish to install R into and then confirm by clicking **Next**.



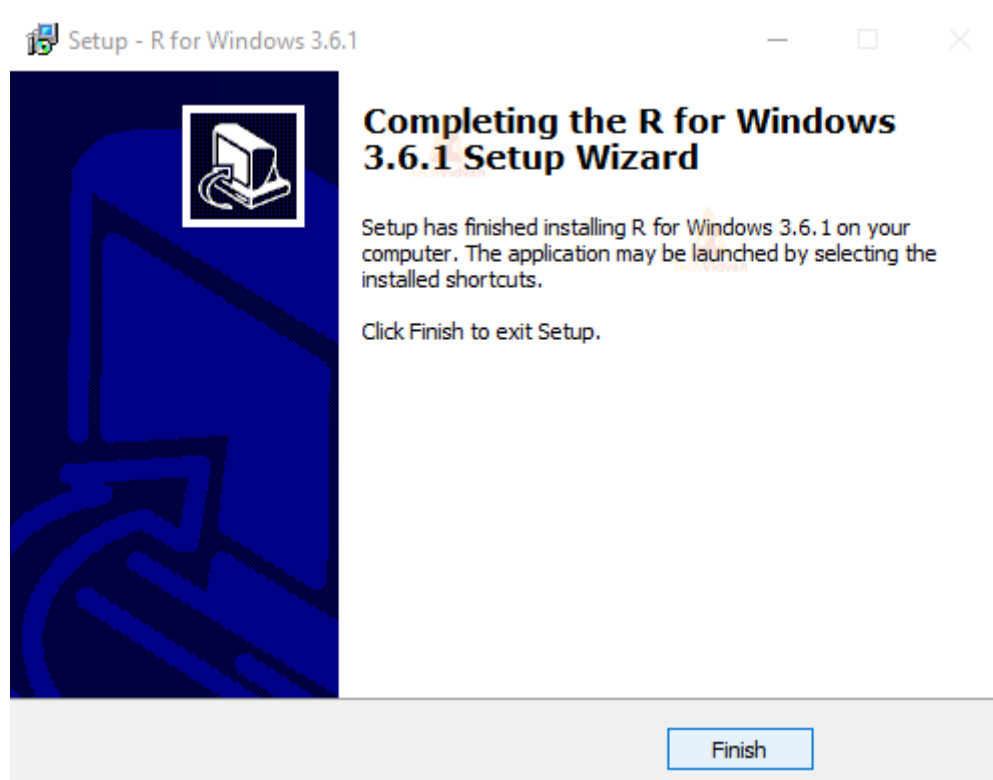
5.e. Select additional tasks like creating desktop shortcuts etc. then click **Next**.



5.f. Wait for the installation process to complete.



5.g. Click on **Finish** to complete the installation.



4. Installation of RStudio on Windows

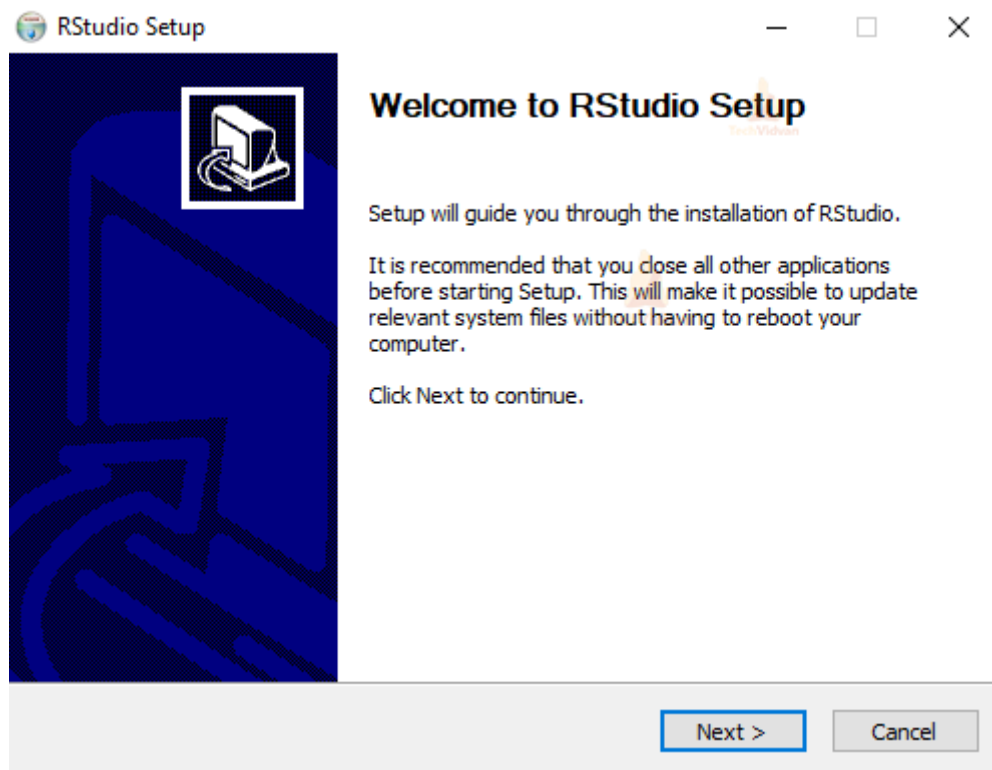
Step – 1: With R-base installed, let's move on to installing RStudio. To begin, go to download RStudio and click on the download button for **RStudio desktop**.

	RStudio Desktop Open Source License Free	RStudio Desktop Commercial License \$995/year	RStudio Server Open Source License Free	RStudio Server Pro Commercial License \$4,975/year (5 Named Users)
Integrated Tools for R	✓	✓	✓	✓
Priority Support		✓		✓
Access via Web Browser			✓	✓
Enterprise Security				✓
Project Sharing				✓
Manage Multiple R Sessions & Versions				✓

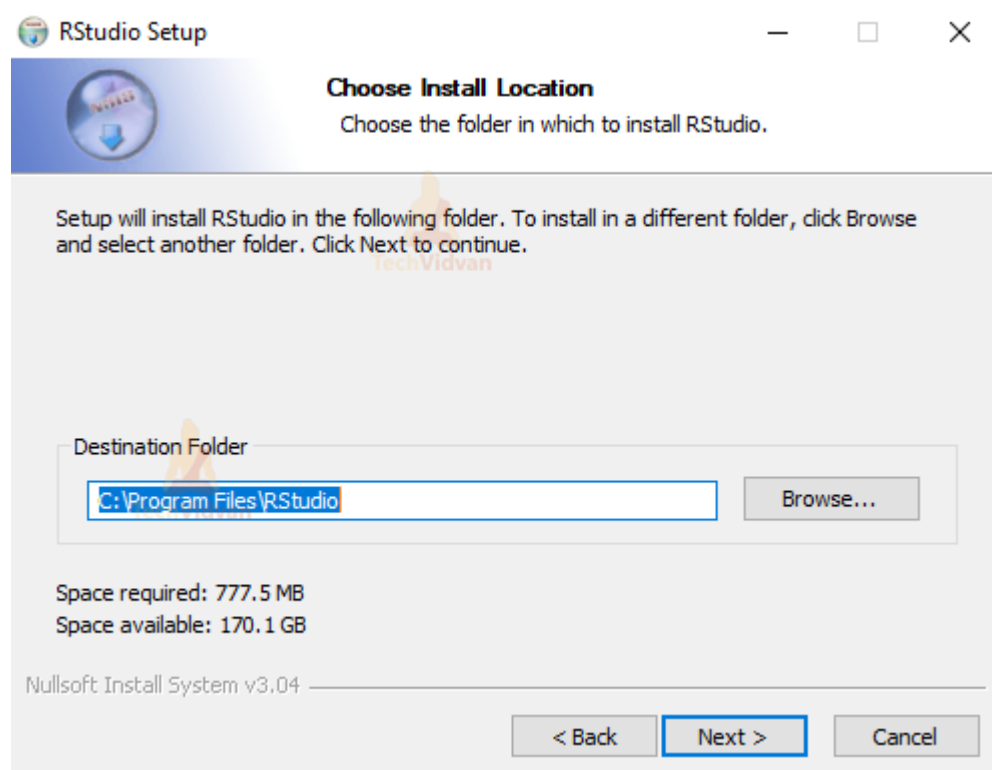
Step – 2: Click on the link for the windows version of RStudio and save the .exe file.

Step – 3: Run the .exe and follow the installation instructions.

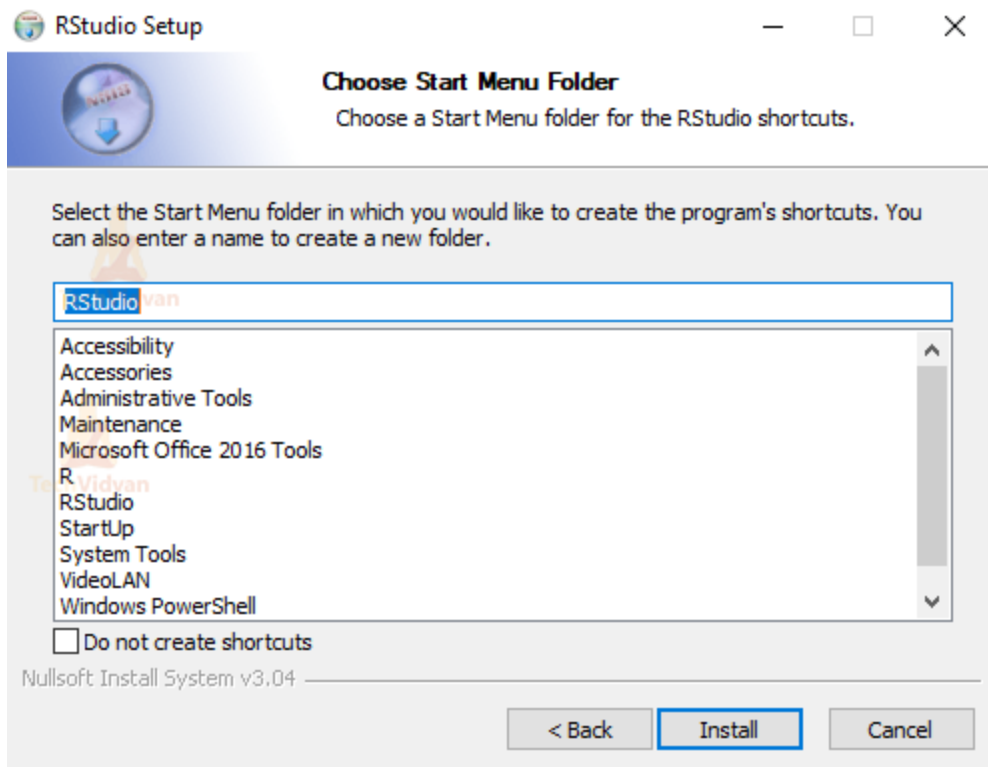
3.a. Click **Next** on the welcome window.



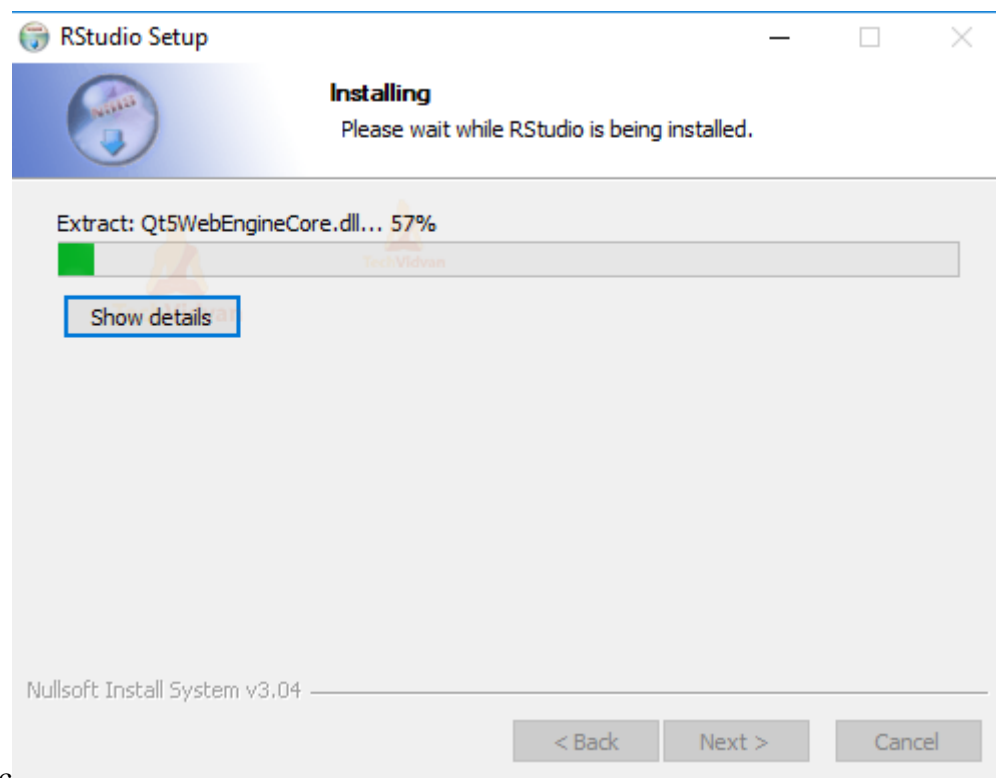
3.b. Enter/browse the path to the installation folder and click **Next** to proceed.



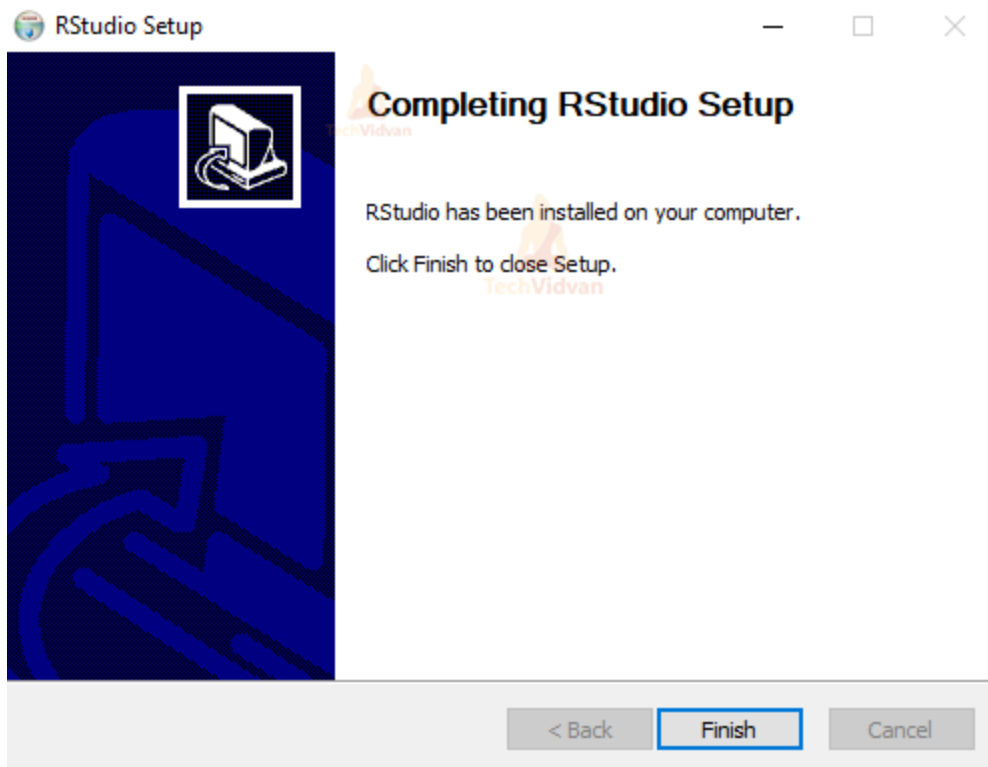
3.c. Select the folder for the start menu shortcut or click on do not create shortcuts and then click **Next**.



3.d. Wait for the installation process to complete.



3.e. Click **Finish** to end the installation.



➤ “R” Objects

An object refers to anything that can be assigned to a variable. Each object has two attributes:

1. length: number of elements in the object
2. mode: denotes type of the object’s data (numeric, character, complex or logical)

5. “R” Data Structures

Data structures are the objects that are manipulated regularly in R. They are used to store data in an organized fashion to make data manipulation and other data operations more efficient. It reduces the complexities of space and time in various tasks.

➤ Vectors

Vector is one of the basic data structures in R. It is homogenous, which means that it only contains elements of the same data type. Data types can be numeric, integer, character, complex, or logical.

Vectors are created by using the `c()` function. Coercion takes place in a vector, from bottom to top, if the elements passed are of different data types, from logical to integer to double to character.

The `typeof()` function is used to check the data type of the vector, and the `class()` function is used to check the class of the vector.

```
Vec1 <- c(44, 25, 64, 96, 30)
Vec2 <- c(1, FALSE, 9.8, "hello world")
typeof(Vec1)
typeof(Vec2)
```

Output:

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

To delete a vector, you simply have to do the following:

```
Vec1 <- NULL
Vec2 <- NULL
```

➤ Methods to Access Vector Elements

Vectors can be accessed in the following ways:

- Elements of a vector can be accessed by using their respective indexes. `[]` brackets are used to specify indexes of the elements to be accessed.

For example:

```
x <- c("Jan", "Feb", "March", "Apr", "May", "June", "July")
y <- x[c(3,2,7)]
print(y)
```

Output:

```
[1] "March" "Feb"   "July"
```

- Logical indexing, negative indexing, and 0/1 can also be used to access the elements of a vector.

For example:

```
x <- c("Jan","Feb","March","Apr","May","June","July")
y <- x[c(TRUE,FALSE,TRUE,FALSE,FALSE,TRUE,TRUE)]
z <- x[c(-3,-7)]
c <- x[c(0,0,0,1,0,0,1)]

print(y)
print(z)
print(c)

Output:

[1] "Jan" "March" "June" "July"(All TRUE values are printed)

[1] "Jan" "Feb" "Apr" "May" "June"(All corresponding values for negative indexes are
dropped)

[1] "Jan" "Jan"(All corresponding values are printed)
```

➤ Vector Arithmetic

You can perform addition, subtraction, multiplication, and division on the vectors having the same number of elements in the following ways:

```
v1 <- c(4,6,7,31,45)
v2 <- c(54,1,10,86,14,57)

add.v <- v1+v2
print(add.v)

sub.v <- v1-v2
print(sub.v)

multi.v <- v1*v2
print(multi.v)

divi.v <- v1/v2
print(divi.v)
```

Output:

```
[1] 58  7 17 117 59 66
[1] -50  5 -3 -55 31 -48
[1] 216  6 70 2666 630 513
[1] 0.07407407 6.00000000 0.70000000 0.36046512 3.21428571 0.15789474
```

➤ **Recycling Vector Elements**

If arithmetic operations are performed on vectors having unequal lengths, then a vector's elements, which are shorter in number as compared to the elements of other vectors, are recycled. For example:

```
v1 <- c(8,7,6,5,0,1)
v2 <- c(7,15)
add.v <- v1+v2
(v2 becomes c(7,15,7,15,7,15))
print(add.v)
sub.v <- v1-v2
print(sub.v)
```

Output:

```
[1] 15 22 13 20  7 16
[1]  1 -8 -1 -10 -7 -14
```

➤ **Sorting a Vector**

You can sort the elements of a vector by using the `sort()` function in the following way:

```
v <- c(4,78,-45,6,89,678)
sort.v <- sort(v)
print(sort.v)

#Sort the elements in the reverse order
```

```
revsort.v <- sort(v, decreasing = TRUE)
print(revsort.v)

#Sorting character vectors
v <- c("Jan","Feb","March","April")
sort.v <- sort(v)
print(sort.v)

#Sorting character vectors in reverse order
revsort.v <- sort(v, decreasing = TRUE)
print(revsort.v)
```

Output:

```
[1] -45  4  6 78 89 678
[1] 678 89 78  6  4 -45
[1] "April" "Feb" "Jan"  "March"
[1] "March" "Jan"  "Feb"  "April"
```

➤ Lists

A list is a non-homogeneous data structure, which implies that it can contain elements of different data types. It accepts numbers, characters, lists, and even matrices and functions inside it. It is created by using the `list()` function.

For example:

```
list1<- list("Sam", "Green", c(8,2,67), TRUE, 51.99, 11.78,FALSE)
print(list1)
```

Output:

```
[[1]]
[1] "Sam"

[[2]]
[1] "Green"

[[3]]
```

```
[1] 8 2 67
[[4]]
[1] TRUE
[[5]]
[1] 51.99
[[6]]
[1] 11.78
[7]]
[1] FALSE
```

➤ Accessing the Elements of a List

The elements of a list can be accessed by using the indices of those elements.

For example:

```
list2 <- list(matrix(c(3,9,5,1,-2,8), nrow = 2), c("Jan","Feb","Mar"), list(3,4,5))
print(list2[1])
print(list2[2])
print(list2[3])
```

Output:

```
[[1]]
[1,] [2,] [3,]      (First element of the list)
[1,]  3  5 -2
[2,]  9  1  8
[[1]]
[1] "Jan" "Feb" "Mar"      (Second element of the list)
[1,]  3  5 -2
[[1]]
[[1]][[1]]
[1] 3
```



```
[[1]][[2]]          (Third element of the list)
```

```
[1] 4
```

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

```
[1] 5
```

➤ Adding and Deleting the Elements of a List

You can add and delete elements only at the end of a list.

For example:

```
list2 <- list(matrix(c(3,9,5,1,-2,8), nrow = 2), c("Jan","Feb","Mar"), list(3,4,5))  
list2[4] <- c("HELLO")  
print(list2[4])
```

Output:

```
[[1]]
```

```
[1] "Hello"
```

Similarly,

```
list2[4] <- NULL  
print(list2[4])
```

Output:

```
[[1]]
```

```
NULL
```

➤ Updating the Elements of a List

To update a value in a list, use the following syntax:

```
list2[3] <- "Element Updated"  
print(list2[3])
```

Output:

```
[[1]]
```

```
[1] "Element Updated"
```

➤ Matrices

Matrix is a two-dimensional data structure that is homogenous, meaning that it only accepts elements of the same data type. Coercion takes place if elements of different data types are passed. It is created by using the `matrix()` function.

The basic syntax to create a matrix is given below:

```
matrix(data, nrow, ncol, byrow, dimnames)
```

where,

`data` = the input element of a matrix given as a vector.

`nrow` = the number of rows to be created.

`ncol` = the number of columns to be created.

`byrow` = the row-wise arrangement of the elements instead of column-wise

`dimnames` = the names of columns or rows to be created.

For example:

```
M1 <- matrix(c(1:9), nrow = 3, ncol = 3, byrow = TRUE)
```

```
print(M1)
```

Output:

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

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

```
[2,]  4  5  6
```

```
[3,]  7  8  9
```

```
M2 <- matrix(c(1:9), nrow = 3, ncol = 3, byrow = FALSE)
```

```
print(M2)
```

Output:

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

```
[1,]  1  4  7
```

```
[2,]  2  5  8
```

```
[3,]  3  6  9
```

➤ Accessing the Elements of a Matrix

To access the elements of a matrix, row and column indices are used in the following ways:
For accessing the elements of the matrix M2 created above, use the following syntax:

```
print(M2[1,1])  
print(M2[3,3])  
print(M2[2,3])
```

Output:

```
[1] 1 (Element at first row and first column)  
[1] 9 (Element at third row and third column)  
[1] 8 (Element at second row and third column)
```

➤ Factor

Factors are used in data analysis for statistical modeling. They are used to categorize unique values in columns, such as “Male”, “Female”, “TRUE”, “FALSE”, etc., and store them as levels. They can store both strings and integers. They are useful in columns that have a limited number of unique values.

Factors can be created using the **factor()** function and they take vectors as inputs.

For example:

```
data <- c("Male","Female","Male","Child","Child","Male","Female","Female")  
print(data)  
factor.data <- factor(data)  
print(factor.data)
```

Output:

```
[1] Male Female Male Child Child Male Female Female  
Levels: Child Female Male
```

➤ Data Frame

Data frame is a two-dimensional array-like structure that also resembles a table, in which each column contains values of one variable and each row contains one set of values from each column.

A data frame has the following characteristics:

- The column names of a data frame should not be empty.
- The row names of a data frame should be unique.
- The data stored in a data frame can be a numeric, factor, or character type.
- Each column should contain the same number of data items.

➤ Creating a Data Frame

You can use the following syntax for creating a data frame in R programming:

```
empid <- c(1:4)
empname <- c("Sam","Rob","Max","John")
empdept <- c("Sales","Marketing","HR","R & D")
emp.data <- data.frame(empid,empname,empdept)
print(emp.data)
```

Output:

Sl.No.	Empid	empname	Empdept
1	1	Sam	Sales
2	2	Rob	Marketing
3	3	Max	HR
4	4	John	R&D

➤ Extracting Columns or Rows from a Data Frame

To extract a specific column from a data frame, use the following syntax:

```
result <- data.frame(emp.data$empname,emp.data$empdept)
print(result)
```

Output:

Sr. No.	emp.data.empname	emp.data.empdept
1	Sam	Sales
2	Rob	Marketing
3	Max	HR
4	John	R&D

To extract specific rows from a data frame, use the following syntax:

```
result <- emp.data[1:2,]  
print(result)
```

Output:

Sr. No.	Empid	empname	Empdept
1	1	Sam	Sales
2	2	Rob	Marketing

The following code extracts the first and third rows with second and third columns respectively.

```
result <- emp.data[c(1,2),c(2,3)]  
print(result)
```

Output:

Sr. No.	Empname	Empdept
1	Sam	Sales
2	Max	HR

➤ Adding a Column to a Data Frame

To add a salary column to the above data frame, you can use the following syntax:

```
emp.data$salary <- c(20000,30000,40000,27000)
n <- emp.data
print(n)
```

Sr. No.	empid	empname	empdept	Salary
1	1	Sam	Sales	20000
2	2	Rob	Marketing	30000
3	3	Max	HR	40000
4	4	John	R & D	27000

➤ Adding a Row to a Data Frame

To add a new row(s) to an existing data frame, you need to create a new data frame that contains the new row(s), and then merge it with the existing data frame using the `rbind()` function.

➤ Creating a New Data Frame

```
emp.newdata <- data.frame(
empid = c(5:7),
empname = c("Frank","Tony","Eric"),
empdept = c("IT","Operations","Finance"),
salary = c(32000,51000,45000)
)
```

➤ Merging the New Data Frame with the Existing Data Frame

```
emp.finaldata <- rbind(emp.data,emp.newdata)
print(emp.finaldata)
```

Output:

Sr. No.	Empid	Empname	empdept	Salary
1	1	Sam	Sales	20000
2	2	Rob	Marketing	30000
3	3	Max	HR	40000
4	4	John	R & D	27000
5	5	Frank	IT	32000
6	6	Tony	Operations	51000
7	7	Eric	Finance	45000

➤ **Arrays**

An Array is a Multi-Dimensional data Structure. Arrays refer to the type of data structure that is used to store homogeneous elements. This leads to a collection of items that are stored at contiguous memory locations. This memory location is denoted by the array name. The position of an element can be calculated simply by adding an offset to its base value.

Syntax: `array(c(elements),dimension=c(rows,cols,number_of_dimensions))`

For example:

```
> vec1<-c(1,2,3,4,5,6)
```

```
> vec2<-c(7,8,9,10,11,12)
```

```
> a1<-array(c(vec1,vec2),dim=c(2,3,2))
```

```
> print(a1)
```

Output:

```
, , 1          #Dimension1
```

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

```
[1,] 1 3 5
```

```
[2,] 2 4 6
```

```
,,2      #Dimension2
```

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

```
[1,] 7 9 11
```

```
[2,] 8 10 12
```

6. CONTROL STRUCTURES

➤ **if Condition in R**

This task is carried out only if this condition is returned as TRUE. R makes it even easier:

You can drop the word *then* and specify your choice in an if statement.

Syntax:

```
if (test_expression) {
```

```
statement
```

```
}
```

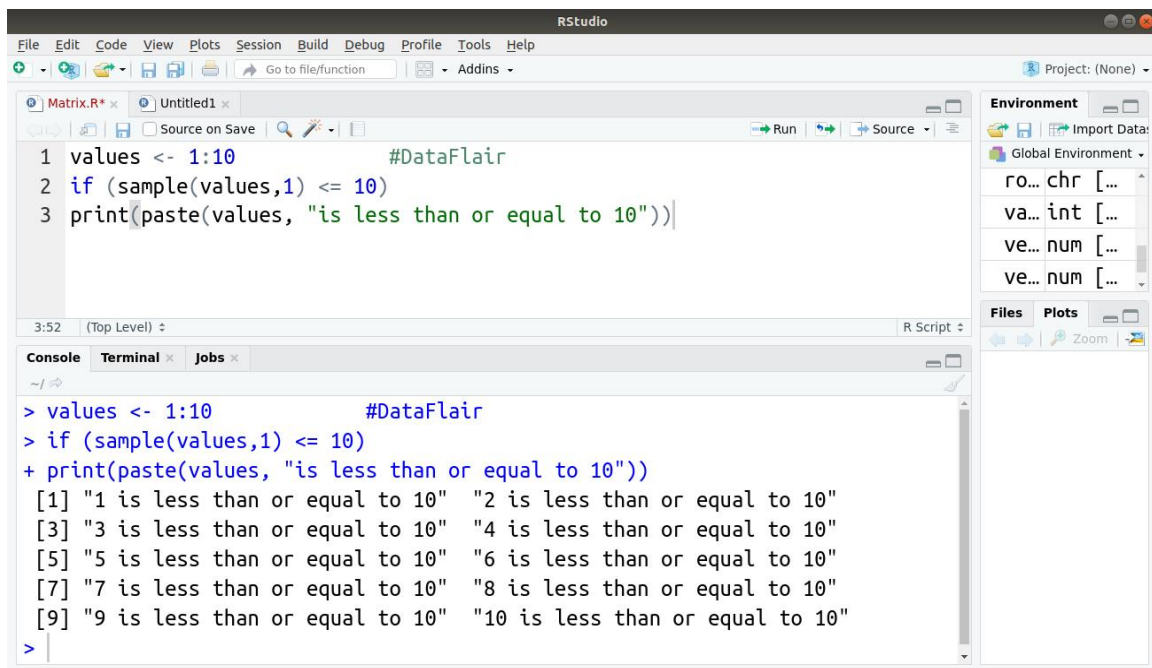
Example:

```
values <- 1:10
```

```
if (sample(values,1) <= 10)
```

```
print(paste(values, "is less than or equal to 10"))
```

Output:



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1 values <- 1:10 #DataFlair
2 if (sample(values,1) <= 10)
3 print(paste(values, "is less than or equal to 10"))
```

The console shows the execution of this code:

```
> values <- 1:10 #DataFlair
> if (sample(values,1) <= 10)
+ print(paste(values, "is less than or equal to 10"))
[1] "1 is less than or equal to 10" "2 is less than or equal to 10"
[3] "3 is less than or equal to 10" "4 is less than or equal to 10"
[5] "5 is less than or equal to 10" "6 is less than or equal to 10"
[7] "7 is less than or equal to 10" "8 is less than or equal to 10"
[9] "9 is less than or equal to 10" "10 is less than or equal to 10"
>
```

➤ if-else Condition in R

An *if...else* statement contains the same elements as an *if* statement with some extra elements:

- The keyword *else*, placed after the first code block.
- The second block of code, contained within braces, that has to be carried out, only if the result of the condition in the *if()* statement is *FALSE*.

Syntax:

```
if (test_expression) {
```

```
statement
```

```
} else {
```

```
statement
```

```
}
```

Example:

```
val1 = 10 #Creating our first variable val1
```

```
val2 = 5 #Creating second variable val2
```

```
if (val1 > val2){ #Executing Conditional Statement based on the comparison
```

```
print("Value 1 is greater than Value 2")
```

```

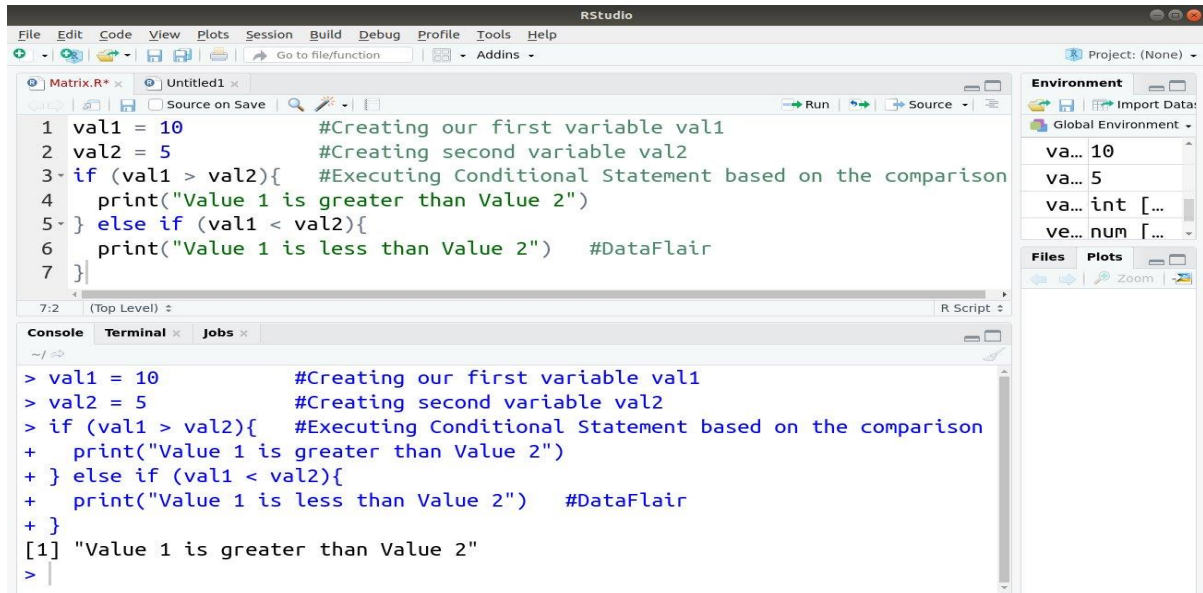
} else if (val1 < val2){

print("Value 1 is less than Value 2")

}

```

Output:



The screenshot shows the RStudio interface. The script editor contains the following R code:

```

1 val1 = 10          #Creating our first variable val1
2 val2 = 5           #Creating second variable val2
3 if (val1 > val2){   #Executing Conditional Statement based on the comparison
4   print("Value 1 is greater than Value 2")
5 } else if (val1 < val2){
6   print("Value 1 is less than Value 2") #DataFlair
7 }

```

The console shows the execution output:

```

> val1 = 10          #Creating our first variable val1
> val2 = 5           #Creating second variable val2
> if (val1 > val2){   #Executing Conditional Statement based on the comparison
+   print("Value 1 is greater than Value 2")
+ } else if (val1 < val2){
+   print("Value 1 is less than Value 2") #DataFlair
+ }
[1] "Value 1 is greater than Value 2"
>

```

The Environment pane on the right shows the global environment with variables `val1` (value 10) and `val2` (value 5).

➤ For loop in R

A loop is a sequence of instructions that is repeated until a certain condition is reached. `for`, `while` and `repeat`, with the additional clauses `break` and `next` are used to construct loops.

Example:

These control structures in R, made of the rectangular box ‘init’ and the diamond. It is executed a known number of times. *for* is a block that is contained within curly braces.

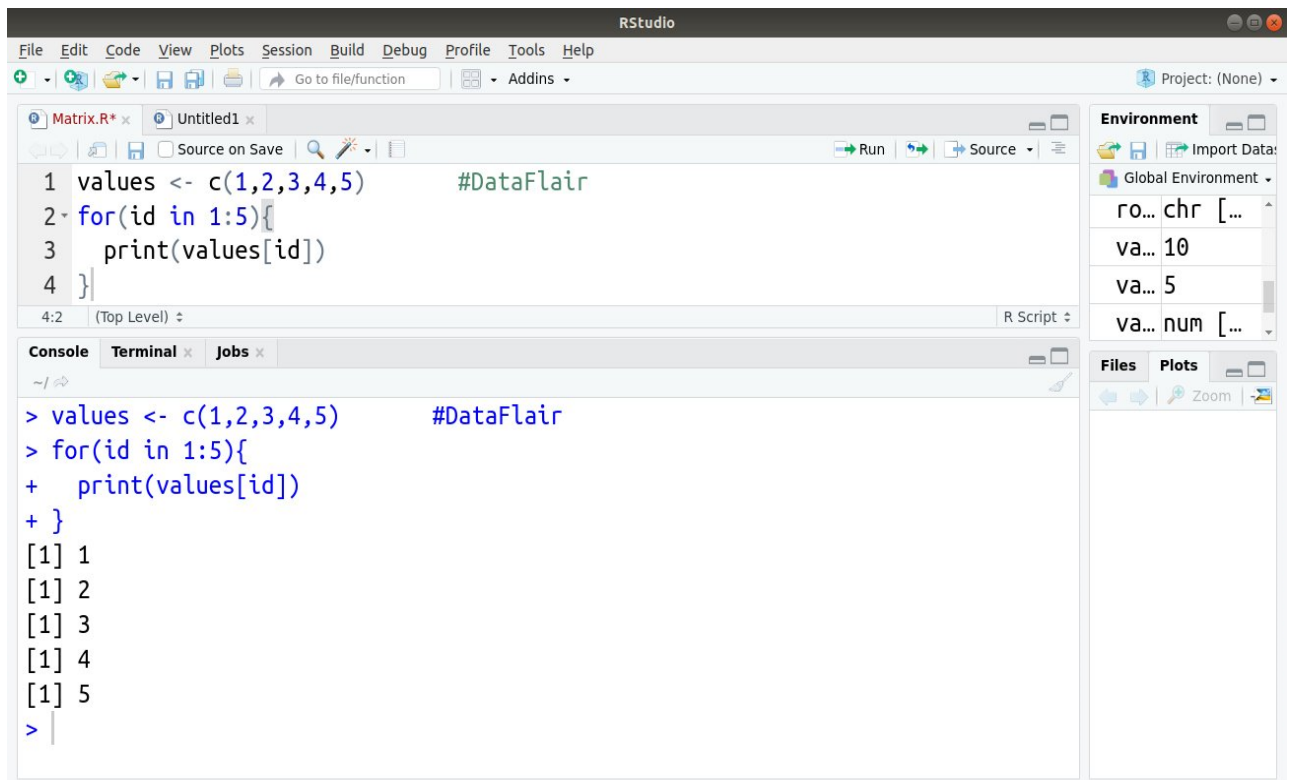
```
values <- c(1,2,3,4,5)
```

```

for(id in 1:5){
print(values[id])
}

```

Output:



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1 values <- c(1,2,3,4,5)      #DataFlair
2 for(id in 1:5){
3   print(values[id])
4 }
```

The console shows the execution output:

```
> values <- c(1,2,3,4,5)      #DataFlair
> for(id in 1:5){
+   print(values[id])
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
>
```

The Environment pane on the right shows the Global Environment with variables: ro... chr [...], va... 10, va... 5, and va... num [...].

➤ while Loop in R

The format is **while(cond) expr**, where **cond** is the condition to test and **expr** is an expression.

Example:

```
val = 2.987
```

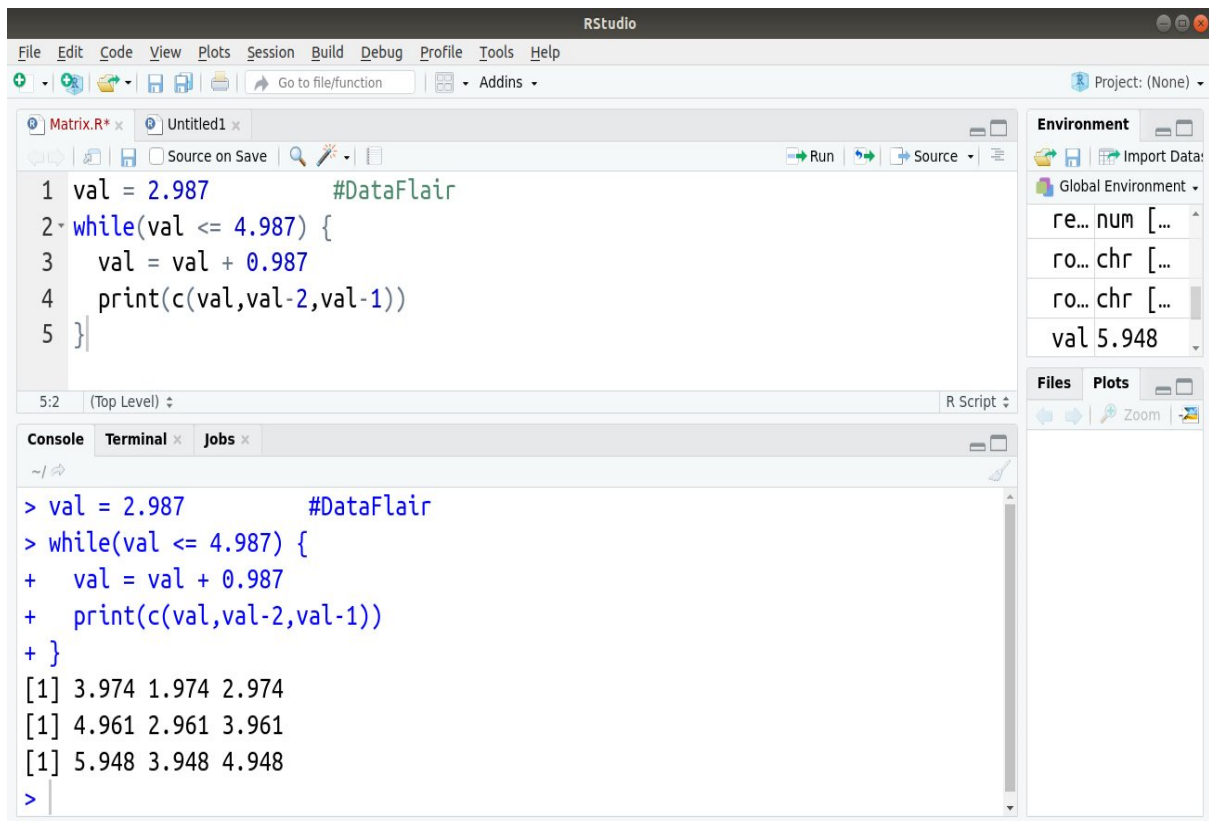
```
while(val <= 4.987) {
```

```
val = val + 0.987
```

```
print(c(val,val-2,val-1))
```

```
}
```

Output:



The screenshot shows the RStudio interface. The script editor contains the following code:

```
1 val = 2.987          #DataFlair
2 while(val <= 4.987) {
3   val = val + 0.987
4   print(c(val,val-2,val-1))
5 }
```

The console shows the output of the script:

```
> val = 2.987          #DataFlair
> while(val <= 4.987) {
+   val = val + 0.987
+   print(c(val,val-2,val-1))
+ }
[1] 3.974 1.974 2.974
[1] 4.961 2.961 3.961
[1] 5.948 3.948 4.948
>
```

The Environment pane on the right shows the following objects:

Object	Class	Value
re...	num	[...]
ro...	chr	[...]
ro...	chr	[...]
val	dbl	5.948

7. 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.

➤ Function Definition

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
}
```

➤ Built-in Function

Simple examples of in-built functions are **seq()**, **mean()**, **max()**, **sum(x)** and **paste(...)** etc. They are directly called by user written programs.

```
# 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 –

```
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44
```

```
[1] 53.5
```

```
[1] 1526
```

➤ User-defined Function

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function is created and used.

```
# Create a function to print squares of numbers in sequence.
```

```
new.function <- function(a) {
```

```
  for(i in 1:a) {
```

```
    b <- i^2
```

```
    print(b)
```

```
  }
```

```
}
```

Calling a Function

```
# Call the function new.function supplying 6 as an argument.
```

```
new.function(6)
```

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

```
[1] 1
```

```
[1] 4
```

```
[1] 9
```

```
[1] 16
```

```
[1] 25
```

```
[1] 36
```

8. Install a New Package

There are two ways to add new R packages. One is installing directly from the CRAN directory and another is downloading the package to your local system and installing it manually.

➤ Install directly from CRAN

The following command gets the packages directly from CRAN webpage and installs the package in the R environment. You may be prompted to choose a nearest mirror. Choose the one appropriate to your location.

```
install.packages("Package Name")
```

```
# Install the package named "dplyr".  
install.packages("dplyr")
```

➤ Load Package to Library

Before a package can be used in the code, it must be loaded to the current R environment. You also need to load a package that is already installed previously but not available in the current environment.

A package is loaded using the following command –

```
library("package Name")
```

Example:

```
library(dplyr)
```

9. Package for “Data Science”- The Tidyverse

. An R *package* is a collection of functions, data, and documentation that extends the capabilities of base R. The packages in the tidyverse share a common philosophy of data and R programming, and are designed to work together naturally. You can install the complete tidyverse with a single line of code:

```
install.packages("tidyverse")
```

Once you have installed a package, you can load it with the library() function:

```
library(tidyverse)
```

```
#> Loading tidyverse: ggplot2
```

```
#> Loading tidyverse: tibble
```

```
#> Loading tidyverse: tidyr
```

```
#> Loading tidyverse: readr
```

```
#> Loading tidyverse: purrr
```

```
#> Loading tidyverse: dplyr
```

```
#> Conflicts with tidy packages -----
```

```
#> filter(): dplyr, stats
```

```
#> lag(): dplyr, stats
```

This tells you that tidyverse is loading the **ggplot2**, **tibble**, **tidyr**, **readr**, **purrr**, and **dplyr** packages

10. DATA HANDLING

➤ Working with CSV Files

Importing CSV Files

Comma separated values or CSV files can be imported and read in using `read.csv()` function.

Syntax:

```
read.csv(filename, header = FALSE, sep = "")
```

Parameters:

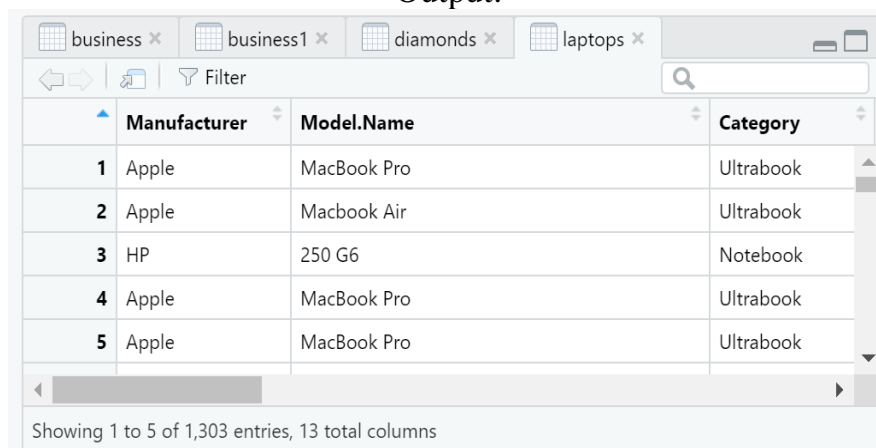
header represents if the file contains header row or not **sep** represents the delimiter value used in file

Example:

```
>laptops<-read.csv("C:/Users/jaideep/Downloads/DataSets-master/DataSets-master/laptops.csv")
```

```
>View(laptops)
```

Output:



	Manufacturer	Model.Name	Category
1	Apple	MacBook Pro	Ultrabook
2	Apple	Macbook Air	Ultrabook
3	HP	250 G6	Notebook
4	Apple	MacBook Pro	Ultrabook
5	Apple	MacBook Pro	Ultrabook

Selecting specific columns from dataset

```
>library(dplyr)
```

```
> laptops %>% select(1,2)->laptops1_2
```

```
> View(laptops1_2)
```

Output: first and Second Column selection from laptops.csv file

	Manufacturer	Model.Name
1	Apple	MacBook Pro
2	Apple	Macbook Air
3	HP	250 G6
4	Apple	MacBook Pro
5	Apple	MacBook Pro
6	Acer	Aspire 3

Showing 1 to 6 of 1,303 entries, 2 total columns

Selecting a range of columns

Taking into consideration the laptops.csv file, a range of columns is selected by the following code

```
> laptops %>% select(3:6)->laptops3_6
```

```
> View(laptops3_6)
```

Output:

	Category	Screen.Size	Screen	CPU
1	Ultrabook	13.3"	IPS Panel Retina Display 2560x1600	Intel Core i5 2.3GHz
2	Ultrabook	13.3"	1440x900	Intel Core i5 1.8GHz
3	Notebook	15.6"	Full HD 1920x1080	Intel Core i5 7200U 2.5GHz
4	Ultrabook	15.4"	IPS Panel Retina Display 2880x1800	Intel Core i7 2.7GHz
5	Ultrabook	13.3"	IPS Panel Retina Display 2560x1600	Intel Core i5 3.1GHz
6	Notebook	15.6"	1366x768	AMD A9-Series 9420 3GHz
7	Ultrabook	15.4"	IPS Panel Retina Display 2880x1800	Intel Core i7 2.2GHz
8	Ultrabook	13.3"	1440x900	Intel Core i5 1.8GHz
9	Ultrabook	14.0"	Full HD 1920x1080	Intel Core i7 8550U 1.8GHz
10	Ultrabook	14.0"	IPS Panel Full HD 1920x1080	Intel Core i5 8250U 1.6GHz
11	Notebook	15.6"	1366x768	Intel Core i5 7200U 2.5GHz
12	Notebook	15.6"	Full HD 1920x1080	Intel Core i3 6006U 2GHz
13	Ultrabook	15.4"	IPS Panel Retina Display 2880x1800	Intel Core i7 2.8GHz
14	Notebook	15.6"	Full HD 1920x1080	Intel Core i3 6006U 2GHz
15	Ultrabook	12.0"	IPS Panel Retina Display 2304x1440	Intel Core M m3 1.2GHz
16	Ultrabook	13.3"	IPS Panel Retina Display 2560x1600	Intel Core i5 2.3GHz
17	Notebook	15.6"	Full HD 1920x1080	Intel Core i7 7500U 2.7GHz

Showing 1 to 17 of 17 entries, 4 total columns

Selecting columns with column names

```
> laptops %>% select("Manufacturer")->lap
```

```
> View(lap)
```


Showing 1 to 6 of 1,303 entries, 1 total columns

	Manufacturer
1	Apple
2	Apple
3	HP
4	Apple
5	Apple
6	Acer

`filter()`

`laptops %>% filter(Manufacturer=="Dell")->dell_laptop`

Showing 1 to 6 of 297 entries, 13 total columns

	Manufacturer	Model.Name	Category	Screen.Size	Screen
1	Dell	Inspiron 3567	Notebook	15.6"	Full HD 1920x1080
2	Dell	Inspiron 3567	Notebook	15.6"	Full HD 1920x1080
3	Dell	XPS 13	Ultrabook	13.3"	IPS Panel Full HD
4	Dell	Inspiron 5379	2 in 1 Convertible	13.3"	Full HD / Touchscreen
5	Dell	Inspiron 3567	Notebook	15.6"	1366x768

`> laptops %>% filter(Manufacturer=="Dell" & Category=="Ultrabook")->dell_laptop`

`> View(dell_laptop)`

	Manufacturer	Model.Name	Category	Screen.Size	Screen
1	Dell	XPS 13	Ultrabook	13.3"	IPS Panel Full HD
2	Dell	Latitude 5590	Ultrabook	15.6"	Full HD 1920x1080
3	Dell	XPS 13	Ultrabook	13.3"	Touchscreen / Quad HD
4	Dell	Vostro 5471	Ultrabook	14.0"	Full HD 1920x1080
5	Dell	Inspiron 5370	Ultrabook	13.3"	IPS Panel Full HD
6	Dell	Latitude 5590	Ultrabook	15.6"	IPS Panel Full HD
7	Dell	XPS 13	Ultrabook	13.3"	IPS Panel Full HD
8	Dell	XPS 13	Ultrabook	13.3"	Quad HD+ / Touchscreen
9	Dell	XPS 13	Ultrabook	13.3"	IPS Panel Full HD
10	Dell	XPS 13	Ultrabook	13.3"	Full HD 1920x1080
11	Dell	XPS 13	Ultrabook	13.3"	IPS Panel 4K Ultra
12	Dell	XPS 13	Ultrabook	13.3"	Full HD 1920x1080

Writing to CSV files

A matrix or data-frame object can be redirected and written to csv file using `write.csv()` function.

Syntax: `write.csv(x, file)`

Parameter:

file specifies the file name used for writing

Example:

```
> x <- c(1, 3, 4, 5, 10)
```

```
> y <- c(2, 4, 6, 8, 10)
```

```
> z <- c(10, 12, 14, 16, 18)
```

```
> data <- cbind(x, y, z)
```

```
> write.csv(data, file="abc.csv")
```

```
> read.csv("abc.csv")
```

Output:

```
> read.csv("abc.csv")
  X  x  y  z
1 1  1  2 10
2 2  3  4 12
3 3  4  6 14
4 4  5  8 16
5 5 10 10 18
> |
```

➤ Working with XML Files

XML is a file format which shares both the file format and the data on the World Wide Web, intranets, and elsewhere using standard ASCII text. It stands for Extensible Markup Language (XML). Similar to HTML it contains markup tags. But unlike HTML where the markup tag describes structure of the page, in xml the markup tags describe the meaning of the data contained into the file.

You can read a xml file in R using the "XML" package. This package can be installed using following command.

```
install.packages("XML")
```

Input Data

Create a XML file by copying the below data into a text editor like notepad. Save the file with a **.xml** extension and choosing the file type as **all files(*.*)**.

```
<RECORDS>
  <EMPLOYEE>
```

```
<ID>1</ID>
<NAME>Rick</NAME>
<SALARY>623.3</SALARY>
<STARTDATE>1/1/2012</STARTDATE>
<DEPT>IT</DEPT>
</EMPLOYEE>

<EMPLOYEE>
  <ID>2</ID>
  <NAME>Dan</NAME>
  <SALARY>515.2</SALARY>
  <STARTDATE>9/23/2013</STARTDATE>
  <DEPT>Operations</DEPT>
</EMPLOYEE>

<EMPLOYEE>
  <ID>3</ID>
  <NAME>Michelle</NAME>
  <SALARY>611</SALARY>
  <STARTDATE>11/15/2014</STARTDATE>
  <DEPT>IT</DEPT>
</EMPLOYEE>

<EMPLOYEE>
  <ID>4</ID>
  <NAME>Ryan</NAME>
  <SALARY>729</SALARY>
  <STARTDATE>5/11/2014</STARTDATE>
  <DEPT>HR</DEPT>
</EMPLOYEE>

<EMPLOYEE>
  <ID>5</ID>
  <NAME>Gary</NAME>
  <SALARY>843.25</SALARY>
  <STARTDATE>3/27/2015</STARTDATE>
  <DEPT>Finance</DEPT>
</EMPLOYEE>

<EMPLOYEE>
  <ID>6</ID>
  <NAME>Nina</NAME>
  <SALARY>578</SALARY>
  <STARTDATE>5/21/2013</STARTDATE>
  <DEPT>IT</DEPT>
</EMPLOYEE>

<EMPLOYEE>
  <ID>7</ID>
  <NAME>Simon</NAME>
  <SALARY>632.8</SALARY>
  <STARTDATE>7/30/2013</STARTDATE>
```

```
<DEPT>Operations</DEPT>
</EMPLOYEE>

<EMPLOYEE>
  <ID>8</ID>
  <NAME>Guru</NAME>
  <SALARY>722.5</SALARY>
  <STARTDATE>6/17/2014</STARTDATE>
  <DEPT>Finance</DEPT>
</EMPLOYEE>

</RECORDS>
```

Reading XML File

The xml file is read by R using the function **xmlParse()**. It is stored as a list in R.

```
# Load the package required to read XML files.
library("XML")

# Also load the other required package.
library("methods")

# Give the input file name to the function.
result <- xmlParse(file = "input.xml")

# Print the result.
print(result)
```

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

```
1
Rick
623.3
1/1/2012
IT

2
Dan
515.2
9/23/2013
Operations

3
Michelle
611
11/15/2014
IT

4
Ryan
729
5/11/2014
```

HR

5

Gary

843.25

3/27/2015

Finance

6

Nina

578

5/21/2013

IT

7

Simon

632.8

7/30/2013

Operations

8

Guru

722.5

6/17/2014

Finance

➤ Working with JSON Files in R Programming

JSON stands for **JavaScript Object Notation**. These files contain the data in human readable format, i.e. as text. Like any other file, one can read as well as write into the JSON files. In order to work with JSON files in R, one needs to install the “**rjson**” package. The most common tasks done using JSON files under rjson packages are as follows:

- Install and load the rjson package in R console
- Create a JSON file
- Reading data from JSON file
- Write into JSON file
- Converting the JSON data into Dataframes
- Working with URLs

Install and load the rjson package

One can install the rjson from the R console using the **install.packages()** command in the following way:

install.packages("rjson")

After installing rjson package one has to load the package using the **library()** function as follows:

library("rjson")

Creating a JSON file

To create a JSON file, one can do the following steps:

Copy the data given below into a notepad file or any text editor file. One can also create his own data as per the given format.

```
{  
  "ID":["1","2","3","4","5"],  
  "Name":["Mithuna","Tanushree","Parnasha","Arjun","Pankaj"],  
  "Salary":["722.5","815.2","1611","2829","843.25"],  
  "StartDate":["6/17/2014","1/1/2012","11/15/2014","9/23/2013","5/21/2013"],  
  "Dept":["IT","IT","HR","Operations","Finance"]  
}
```

Choose “**all types**” as the file type and save the file with **.json** extension.(Example: example.json)

One must make sure that the information or data is contained within a pair of curly braces { } .

Reading a JSON file

In R, reading a JSON file is quite a simple task. One can extract and read the data of a JSON file very efficiently using the **fromJSON()** function. The **fromJSON()** function takes the JSON file and returns the extracted data from the JSON file in the list format by default.

Example:

Suppose the above data is stored in a file named **example.json** in the E drive. To read the file we must write the following code.

```
# Read a JSON file  
  
# Load the package required to read JSON files.  
  
library("rjson")  
  
# Give the input file name to the function.  
  
result <- fromJSON(file = "E:\\example.json")  
  
# Print the result.  
  
print(result)
```

Output:

\$ID

```
[1] "1" "2" "3" "4" "5"
```

\$Name

```
[1] "Mithuna" "Tanushree" "Parnasha" "Arjun" "Pankaj"
```

\$Salary

```
[1] "722.5" "815.2" "1611" "2829" "843.25"
```

\$StartDate

```
[1] "6/17/2014" "1/1/2012" "11/15/2014" "9/23/2013" "5/21/2013"
```

\$Dept

```
[1] "IT" "IT" "HR" "Operations" "Finance"
```

Writing into a JSON file

One need to create a JSON Object using **toJSON()** function before he writes the data to a JSON file. To write into a JSON file use the **write()** function.

Example:

```
# Writing into JSON file.
```

```
# Load the package required to read JSON files.
```

```
library("rjson")
```

```
# creating the list
```

```
list1 <- vector(mode="list", length=2)
```

```
list1[[1]] <- c("sunflower", "guava", "hibiscus")
```

```
list1[[2]] <- c("flower", "fruit", "flower")
```

```
# creating the data for JSON file
```

```
jsonData <- toJSON(list1)
```

```
# writing into JSON file

write(jsonData, "result.json")

# Give the created file name to the function

result <- fromJSON(file = "result.json")

# Print the result

print(result)
```

Output:

```
[[1]]
[1] "sunflower" "guava"      "hibiscus"

[[2]]
[1] "flower" "fruit"  "flower"
```

Working with URLs

One can take datasets from any websites and extract the data and use them. This can be done under any of two packages, namely **RJSONIO** and **jsonlite**.

Example:

```
# working with URL

# import required library

library(RJSONIO)

# extracting data from the website

Raw <- fromJSON( "https://data.ny.gov/api/views/9a8c-vfzj/rows.json?accessType=DOWNLOAD")

# extract the data node

food_market <- Raw[['data']]
```



```
# assembling the data into data frames

Names <- sapply(food_market, function(x) x[[14]])

head(Names)
```

Output:

```
[[1]]

[1] "BRENTS EXPRESS STOP"

[[2]]

[1] "COUNTRY WAGON PRODUCE"

[[3]]

[1] "QUICKWAY FOOD STORE #74"

[[4]]

[1] "DOLLAR GENERAL # 17913"

[[5]]

[1] "DOLLAR GENERAL #8363"

[[6]]

[1] "FEDERAL MEATS"
```

➤ Databases in R Programming Language

R can be connected to many relational databases such as Oracle, MySQL, SQL Server, etc, and fetches the result as a data frame. Once the result set is fetched into data frame, it becomes very easy to visualize and manipulate them. In this article, we'll discuss MySQL as reference to connect with R, creating, dropping, inserting, updating, and querying the table using R Language.

RMySQL Package

It is a built-in package in R and Its provides connectivity between the R and MySql databases. It can be installed with the following commands:

```
install.packages("RMySQL")
```

Connecting MySQL with R Programming Language

R requires **RMySQL** package to create a connection object which takes username, password, hostname and database name while calling the function. **dbConnect()** function is used to create the connection object in R.

Syntax: *dbConnect(drv, user, password, dbname, host)*

Parameter values:

- **drv** represents Database Driver
- **user** represents username
- **password** represents password value assigned to Database server
- **dbname** represents name of the database
- **host** represents host name

Example:

```
# Install package
```

```
install.packages("RMySQL")
```

```
# Loading library
```

```
library("RMySQL")
```

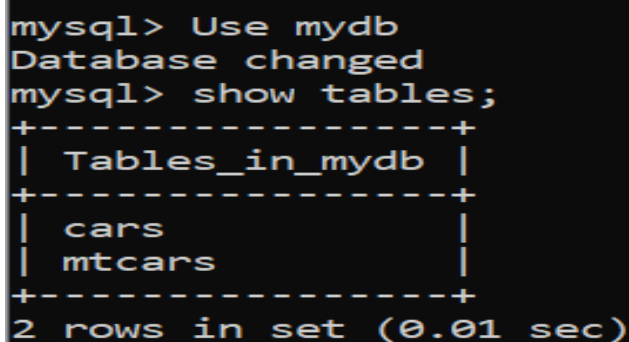
```
# Create connection
```

```
mysqlconn = dbConnect(MySQL(), user = 'root', password = 'welcome', dbname = 'mydb', host  
='localhost')
```

```
# Show tables in database
```

```
dbListTables(mysqlconn)
```

Tables present in database named “mydb”:



```
mysql> Use mydb
Database changed
mysql> show tables;
+-----+
| Tables_in_mydb |
+-----+
| cars            |
| mtcars          |
+-----+
2 rows in set (0.01 sec)
```

Create Tables in MySQL Using R

Tables in MySQL can be created using function **dbSendQuery()** in R.

```
# Create connection object
```

```
mysqlconn = dbConnect(MySQL(), user = 'root', password = 'welcome', dbname = 'mydb', host = 'localhost')
```

```
# Create new table Cars
```

```
> dbSendQuery(mysqlconn, 'CREATE TABLE Cars(Id INTEGER PRIMARY KEY, Name VARCHAR(20), Price INT)')
```

Insert into Table in MySQL Using R

Here we are going to insert a value into a table.

Example:

```
# Create connection object
```

```
mysqlconn = dbConnect(MySQL(), user = 'root', password = 'welcome', dbname = 'mydb', host = 'localhost')
```

```
# Inserting into articles table
```

```
> dbSendQuery(mysqlconn, "INSERT INTO Cars VALUES(1,'Audi',52642)")
```

Output:

```
<MySQLResult:1702063201,0,2>
```

Database content:

```
mysql> select * from Cars
-> ;
+----+-----+-----+
| Id | Name | Price |
+----+-----+-----+
|  1 | Audi | 52642 |
+----+-----+-----+
1 row in set (0.00 sec)
```

Selecting Data from MySQL table using R

```
> query = "SELECT * FROM Cars";  
> rs = dbSendQuery(mysqlconn, query);  
> df = fetch(rs, -1);  
> df  
  Id Name Price  
1  1 Audi 52642
```

Working with Excel Files

Reading Excel Files in R Programming Language

First, install **readxl** package in R to load excel files.

```
> library(readxl)
```

```
> Data1 <- read_excel("C:/Users/jaideep/Documents/practiceR/excel.xlsx")
```

```
> head(Data1)
```

OUTPUT:

```
# A tibble: 6 × 3  
  a      b      c  
  <chr> <chr> <chr>  
1 a      b      c  
2 a      b      c  
3 a      b      c  
4 a      b      c  
5 a      b      c  
6 a      b      c  
> |
```

Data Visualization using R

➤ R – graphs and Charts

ggplot2 is a plotting package that provides helpful commands to create complex plots from data in a data frame. It provides a more programmatic interface for specifying what variables to plot, how they are displayed, and general visual properties.

To build a ggplot, we will use the following basic template that can be used for different types of plots:

```
ggplot(data = <DATA>, mapping = aes(<MAPPINGS>)) + <GEOM_FUNCTION>()
```

- define a data set.
- define an aesthetic mapping (using the aesthetic (aes) function), by selecting the variables to be plotted and specifying how to present them in the graph, e.g., as x/y positions or characteristics such as size, shape, color, etc.

```
ggplot(data = surveys_complete, mapping = aes(x = weight, y = hindfoot_length))
```

- add ‘geoms’ – graphical representations of the data in the plot (points, lines, bars). **ggplot2** offers many different geoms; we will use some common ones today, including:
 - `geom_point()` for scatter plots, dot plots, etc.
 - `geom_boxplot()` for, well, boxplots!
 - `geom_line()` for trend lines, time series, etc.

Example:

Consider diamonds dataset for this example. Use Command **View(diamonds)** to view dataset and `?diamond` to get complete information about diamonds dataset.

```
>library(tidyverse)
```

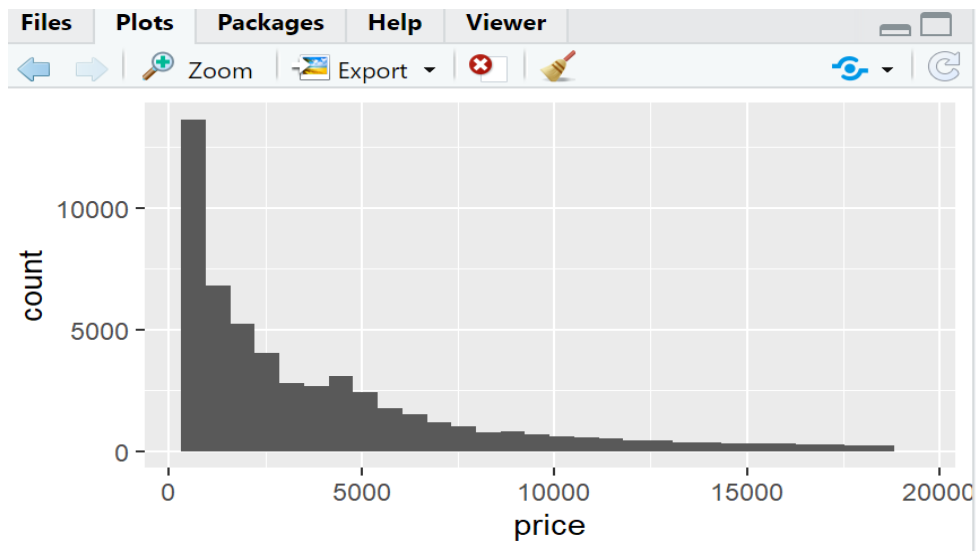
```
> library(ggplot2)
```

```
> ggplot(data=diamonds)
```

```
> ggplot(data=diamonds, aes(x=price))
```

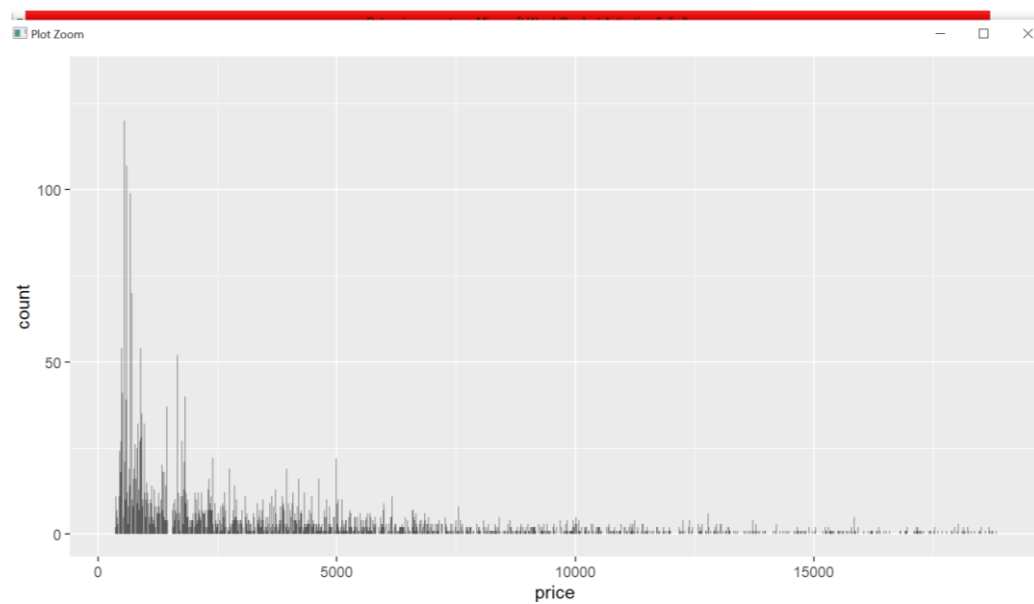
```
> ggplot(data=diamonds,aes(x=price))+geom_histogram()
```

Output: Histogram



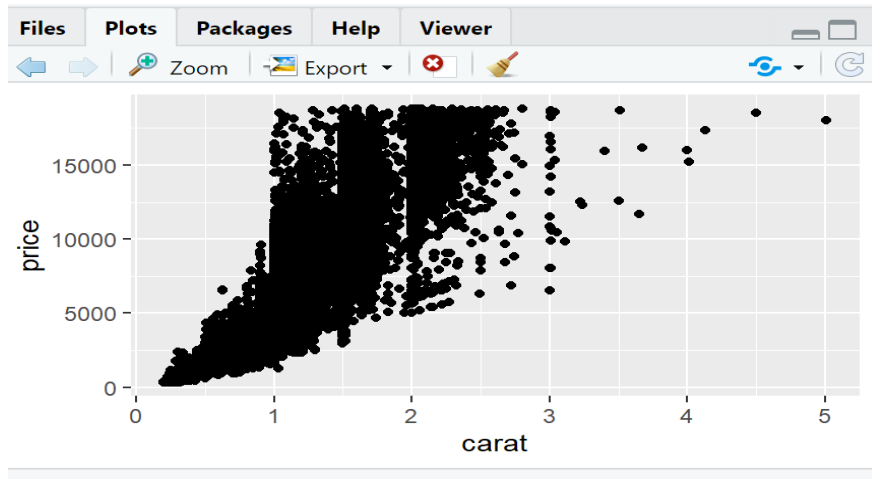
```
> ggplot(data=diamonds,aes(x=price))+geom_bar()
```

Output: Bar graph



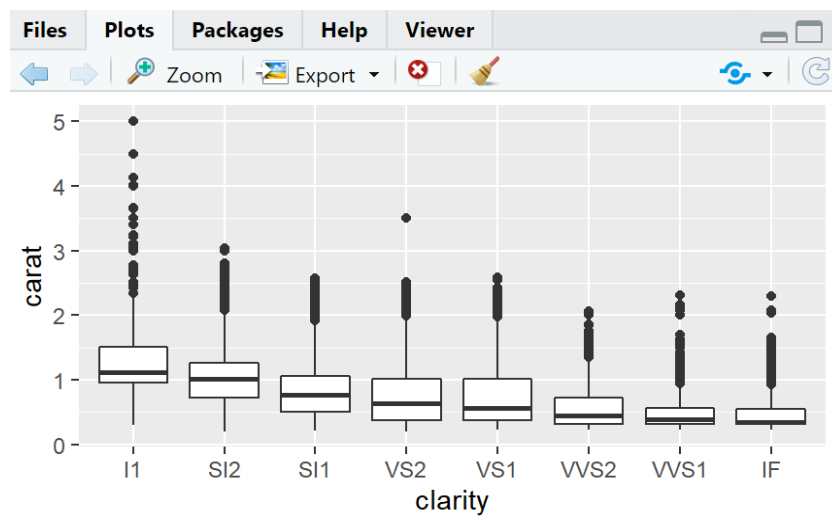
```
> ggplot(data=diamonds,aes(x=carat, y=price))+geom_point()
```

Output: Scatter plot



```
> ggplot(data=diamonds,aes(x=clarity, y=carat))+geom_boxplot()
```

Output: BoxPlot



```
> ggplot(data=diamonds,aes(x=clarity, y=carat))+geom_line()
```

Output: Line graph



11. Statistical Analysis Using R

➤ Measure of Central Tendency

Central Tendency is one of the features of descriptive statistics. Central tendency tells about how the group of data is clustered around the centre value of the distribution. Central tendency performs the following measures:

- Arithmetic Mean
- Geometric Mean
- Harmonic Mean
- Median
- Mode

Arithmetic Mean

The arithmetic mean is simply called the average of the numbers which represents the central value of the data distribution. It is calculated by adding all the values and then dividing by the total number of observations.

Formula:

$$X = \frac{1}{n} \sum_{i=1}^n x_i = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

where,

X indicates the arithmetic mean x_i indicates i^{th} value in data vector n indicates total number of observations. In R language, arithmetic mean can be calculated by `mean()` function.

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

Parameters:

x : Represents object

$trim$: Specifies number of values to be removed from each side of object before

calculating the mean. The value is between 0 to 0.5
na.rm: If TRUE then removes the NA value from *x*

Example:

```
# Defining vector
```

```
x <- c(3, 7, 5, 13, 20, 23, 39, 23, 40, 23, 14, 12, 56, 23)
```

```
# Print mean
```

```
print(mean(x))
```

Output:

```
[1] 21.5
```

Geometric Mean

The geometric mean is a type of mean that is computed by multiplying all the data values and thus, shows the central tendency for given data distribution.

Formula:

$$X = \left(\prod_{i=1}^n x_i \right)^{\frac{1}{n}} = \sqrt[n]{x_1 x_2 \cdots x_n}$$

where,

X indicates geometric mean *x_i* indicates *ith* value in data vector *n* indicates total number of observations **prod()** and **length()** function helps in finding the geometric mean for given set of numbers as there is no direct function for geometric mean.

Syntax:

prod(x)^(1/length(x))

where,

prod() function returns the product of all values present in vector *x*

length() function returns the length of vector *x*

Example:

```
# Defining vector
```

```
x <- c(1, 5, 9, 19, 25)
```

```
# Print Geometric Mean
```

```
print(prod(x)^(1 / length(x)))
```

Output:

```
[1] 7.344821
```

Harmonic Mean

Harmonic mean is another type of mean used as another measure of central tendency. It is computed as reciprocal of the arithmetic mean of reciprocals of the given set of values.

Formula:

$$X = \frac{N}{\sum_{i=1}^N \frac{1}{x_i}}$$

where,

X indicates harmonic mean x_i indicates i^{th} value in data vector

n indicates total number of observations

Example:

Modifying the code to find the harmonic mean of given set of values.

```
# Defining vector
```

```
x <- c(1, 5, 8, 10)
```

```
# Print Harmonic Mean
```

```
print(1 / mean(1 / x))
```

Output:

```
[1] 2.807018
```

Median

Median in statistics is another measure of central tendency which represents the middlemost value of a given set of values.

In R language, median can be calculated by **median()** function.

Syntax: `median(x, na.rm = FALSE)`

Parameters:

x: It is the data vector

na.rm: If TRUE then removes the NA value from x

Example:

```
# Defining vector

x <- c(3, 7, 5, 13, 20, 23, 39, 23, 40, 23, 14, 12, 56, 23)

# Print Median

median(x)
```

Output:

```
[1] 21.5
```

Mode

The mode of a given set of values is the value that is repeated most in the set. There can exist multiple mode values in case if there are two or more values with matching maximum frequency.

Example 1: Single-mode value

In R language, there is no function to calculate mode. So, modifying the code to find out the mode for a given set of values.

```
# Defining vector

x <- c(3, 7, 5, 13, 20, 23, 39, 23, 40, 23, 14, 12, 56, 23, 29, 56, 37, 45, 1, 25, 8)

# Generate frequency table

y <- table(x)

# Print frequency table

print(y)

# Mode of x

m <- names(y)[which(y == max(y))]

# Print mode

print(m)
```

Output:

```
x
1 3 5 7 8 12 13 14 20 23 25 29 37 39 40 45 56
1 1 1 1 1 1 1 1 1 4 1 1 1 1 1 1 2
[1] "23"
```

Example 2: Multiple Mode values

```
# Defining vector
```

```
x <- c(3, 7, 5, 13, 20, 23, 39, 23, 40, 23, 14, 12, 56, 23, 29, 56, 37, 45, 1, 25, 8, 56, 56)
```

```
# Generate frequency table
```

```
y <- table(x)
```

```
# Print frequency table
```

```
print(y)
```

```
# Mode of x
```

```
m <- names(y)[which(y == max(y))]
```

```
# Print mode
```

```
print(m)
```

Output:

```
x
1 3 5 7 8 12 13 14 20 23 25 29 37 39 40 45 56
1 1 1 1 1 1 1 1 1 4 1 1 1 1 1 1 4
[1] "23" "56"
```

➤ *Measures of Variability*

Following are some of the measures of variability that R offers to differentiate between data sets:

- Variance
- Standard Deviation
- Range

- Mean Deviation
- Interquartile Range

Variance

Variance is a measure that shows how far is each value from a particular point, preferably mean value. Mathematically, it is defined as the average of squared differences from the mean value.

Formula:

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

where,

σ^2 specifies variance of the data set

x^i specifies i^{th} value in data set

μ specifies the mean of data set

n specifies total number of observations

In the R language, there is a standard built-in function to calculate the variance of a data set.

Syntax: `var(x)`

Parameter:

x: It is data vector

Example:

```
# Defining vector
```

```
x <- c(5, 5, 8, 12, 15, 16)
```

```
# Print variance of x
```

```
print(var(x))
```

Output:

```
[1] 23.76667
```

Standard Deviation

Standard deviation in statistics measures the spreadness of data values with respect to mean and mathematically, is calculated as square root of variance.

Formula:

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

where, σ^2 specifies standard deviation of the data set x^i specifies i^{th} value in data set μ specifies the mean of data set n specifies total number of observations

In R language, there is no standard built-in function to calculate the standard deviation of a data set. So, modifying the code to find the standard deviation of data set.

Example:

Defining vector

x <- c(5, 5, 8, 12, 15, 16)

Standard deviation

d <- sqrt(var(x))

Print standard deviation of x

print(d)

Output:

[1] 4.875107

Range

Range is the difference between maximum and minimum value of a data set. In R language, **max()** and **min()** is used to find the same, unlike **range()** function that returns the minimum and maximum value of data set.

Example:

Defining vector

```
x <- c(5, 5, 8, 12, 15, 16)
```

```
# range() function output
```

```
print(range(x))
```

```
# Using max() and min() function
```

```
# to calculate the range of data set
```

```
print(max(x)-min(x))
```

Output:

```
[1] 5 16
```

```
[1] 11
```

Mean Deviation

Mean deviation is a measure calculated by taking an average of the arithmetic mean of the absolute difference of each value from the central value. Central value can be mean, median, or mode.

Formula:

$$MD \equiv \frac{1}{n} \sum_{i=1}^n |x_i - \mu|$$

where,

x^i specifies i^{th} value in data set

μ specifies the mean of data set

n specifies total number of observations

In R language, there is no standard built-in function to calculate mean deviation. So, modifying the code to find mean deviation of the data set.

Example:

```
# Defining vector
```

```
x <- c(5, 5, 8, 12, 15, 16)
```

```
# Mean deviation
```

```
md <- sum(abs(x-mean(x)))/length(x)
```

```
# Print mean deviation
```

```
print(md)
```

Output:

```
[1] 4.166667
```

Interquartile Range

Interquartile Range is based on splitting a data set into parts called as quartiles. There are 3 quartile values (Q1, Q2, Q3) that divide the whole data set into 4 equal parts. Q2 specifies the median of the whole data set.

Mathematically, the interquartile range is depicted as:

$$IQR = Q3 - Q1$$

where,

Q3 specifies the median of n largest values

Q1 specifies the median of n smallest values

In R language, there is built-in function to calculate the interquartile range of data set.

Syntax: *IQR(x)*

Parameter:

x: *It specifies the data set*

Example:

```
# Defining vector
```

```
x <- c(5, 5, 8, 12, 15, 16)
```

```
# Print Interquartile range
```

```
print(IQR(x))
```

Output:

```
[1] 8.5
```

13. Hypothesis Testing

A hypothesis is made by the researchers about the data collected for any experiment or data set. A hypothesis is an assumption made by the researchers that are not mandatory true. In simple words, a hypothesis is a decision taken by the researchers

based on the data of the population collected. Hypothesis Testing in R Programming is a process of testing the hypothesis made by the researcher or to validate the hypothesis. To perform hypothesis testing, a random sample of data from the population is taken and testing is performed. Based on the results of testing, the hypothesis is either selected or rejected. This concept is known as Statistical Inference. the four-step process of hypothesis testing are One sample T-Testing, Two-sample T-Testing, Directional Hypothesis, one sample T-test, two sample T-test and correlation test in R programming.

Four Step Process of Hypothesis Testing

There are 4 major steps in hypothesis testing:

- **State the hypothesis-** This step is started by stating null and alternative hypothesis which is presumed as true.
- **Formulate an analysis plan and set the criteria for decision-** In this step, significance level of test is set. The significance level is the probability of a false rejection in a hypothesis test.
- **Analyze sample data-** In this, a test statistic is used to formulate the statistical comparison between the sample mean and the mean of the population or standard deviation of the sample and standard deviation of the population.
- **Interpret decision-** The value of the test statistic is used to make the decision based on the significance level. For example, if the significance level is set to 0.1 probability, then the sample mean less than 10% will be rejected. Otherwise, the hypothesis is retained to be true.

One Sample T-Testing

One sample T-Testing approach collects a huge amount of data and tests it on random samples. To perform T-Test in R, normally distributed data is required. This test is used to test the mean of the sample with the population. For example, the height of persons living in an area is different or identical to other persons living in other areas.

Syntax: `t.test(x, mu)`

Parameters:

x: represents numeric vector of data

mu: represents true value of the mean

To know about more optional parameters of **t.test()**, try below command:

`help("t.test")`

Example:

```
# Defining sample vector
```

```
x <- rnorm(100)
```

```
# One Sample T-Test
```

```
t.test(x, mu = 5)
```

Output:

One Sample t-test

data: x

t = -49.504, df = 99, p-value < 2.2e-16

alternative hypothesis: true mean is not equal to 5

95 percent confidence interval:

-0.1910645 0.2090349

sample estimates:

mean of x

0.008985172

Two Sample T-Testing

In two sample T-Testing, the sample vectors are compared. If var.equal = TRUE, the test assumes that the variances of both the samples are equal.

Syntax: *t.test(x, y)*

Parameters:

x and y: Numeric vectors

Example:

```
# Defining sample vector
```

```
x <- rnorm(100)
```

```
y <- rnorm(100)
```

```
# Two Sample T-Test
```

```
t.test(x, y)
```

Output:

Welch Two Sample t-test

data: x and y

t = -1.0601, df = 197.86, p-value = 0.2904

alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:

-0.4362140 0.1311918

sample estimates:

mean of x mean of y

-0.05075633 0.10175478

Directional Hypothesis

Using the directional hypothesis, the direction of the hypothesis can be specified like, if the user wants to know the sample mean is lower or greater than another mean sample of the data.

Syntax: *t.test(x, mu, alternative)*

Parameters:

x: represents numeric vector data

mu: represents mean against which sample data has to be tested

alternative: sets the alternative hypothesis

Example:

```
# Defining sample vector
```

```
x <- rnorm(100)
```

```
# Directional hypothesis testing
```

```
t.test(x, mu = 2, alternative = 'greater')
```

Output:

One Sample t-test

data: x

t = -20.708, df = 99, p-value = 1

alternative hypothesis: true mean is greater than 2

95 percent confidence interval:

-0.2307534 Inf

sample estimates:

mean of x

-0.0651628

One Sample T-Test

This type of test is used when comparison has to be computed on one sample and the data is non-parametric. It is performed using `wilcox.test()` function in R programming.

Syntax: *wilcox.test(x, y, exact = NULL)*

Parameters:

x and y: represents numeric vector

exact: represents logical value which indicates whether p-value be computed

To know about more optional parameters of **wilcox.test()**, use below command:

`help("wilcox.test")`

Example:

```
# Define vector
```

```
x <- rnorm(100)
```

```
# one sample test
```

```
wilcox.test(x, exact = FALSE)
```

Output:

Wilcoxon signed rank test with continuity correction

```
data: x
```

```
V = 2555, p-value = 0.9192
```

```
alternative hypothesis: true location is not equal to 0
```

Two Sample T-Test

This test is performed to compare two samples of data.

Example:

```
# Define vectors
```

```
x <- rnorm(100)
```

```
y <- rnorm(100)
```

```
# Two sample test
```

```
wilcox.test(x, y)
```

Output:

Wilcoxon rank sum test with continuity correction

data: x and y

W = 5300, p-value = 0.4643

alternative hypothesis: true location shift is not equal to 0

Correlation Test

This test is used to compare the correlation of the two vectors provided in the function call or to test for the association between the paired samples.

Syntax: *cor.test(x, y)*

Parameters:

x and y: represents numeric data vectors

To know about more optional parameters in **cor.test()** function, use below command:

`help("cor.test")`

Example:

Using mtcars dataset in R

```
cor.test(mtcars$mpg, mtcars$hp)
```

Output:

Pearson's product-moment correlation

data: mtcars\$mpg and mtcars\$hp

t = -6.7424, df = 30, p-value = 1.788e-07

alternative hypothesis: true correlation is not equal to 0

95 percent confidence interval:

-0.8852686 -0.5860994

sample estimates:

cor

-0.7761684

14. Chi-Square Test

The chi-square test of independence evaluates whether there is an association between the categories of the two variables. There are basically two types of random variables and they yield two types of data: numerical and categorical. Chi-square statistics is used to investigate whether distributions of categorical variables differ

from one another. Chi-square test is also useful while comparing the tallies or counts of categorical responses between two(or more) independent groups.

In R, the function used for performing a chi-square test is **chisq.test()**.

Syntax:

chisq.test(data)

Parameters:

data: *data is a table containing count values of the variables in the table.*

Example

We will take the survey data in the **MASS** library which represents the data from a survey conducted on students.

```
# load the MASS package
```

```
library(MASS)
```

```
print(str(survey))
```

Output:

```
'data.frame': 237 obs. of 12 variables:
```

```
$ Sex : Factor w/ 2 levels "Female","Male": 1 2 2 2 2 1 2 1 2 2 ...
```

```
$ Wr.Hnd: num 18.5 19.5 18 18.8 20 18 17.7 17 20 18.5 ...
```

```
$ NW.Hnd: num 18 20.5 13.3 18.9 20 17.7 17.7 17.3 19.5 18.5 ...
```

```
$ W.Hnd : Factor w/ 2 levels "Left","Right": 2 1 2 2 2 2 2 2 2 2 ...
```

```
$ Fold : Factor w/ 3 levels "L on R","Neither",...: 3 3 1 3 2 1 1 3 3 3 ...
```

```
$ Pulse : int 92 104 87 NA 35 64 83 74 72 90 ...
```

```
$ Clap : Factor w/ 3 levels "Left","Neither",...: 1 1 2 2 3 3 3 3 3 3 ...
```

```
$ Exer : Factor w/ 3 levels "Freq","None",...: 3 2 2 2 3 3 1 1 3 3 ...
```

```
$ Smoke : Factor w/ 4 levels "Heavy","Never",...: 2 4 3 2 2 2 2 2 2 2 ...
```

```
$ Height: num 173 178 NA 160 165 ...
```

```
$ M.I : Factor w/ 2 levels "Imperial","Metric": 2 1 NA 2 2 1 1 2 2 2 ...
```

```
$ Age : num 18.2 17.6 16.9 20.3 23.7 ...
```

```
NULL
```

The above result shows the dataset has many Factor variables which can be considered as categorical variables. For our model, we will consider the variables “**Exer**” and “**Smoke**“.The Smoke column records the students smoking habits while the Exer column records their exercise level. Our aim is to test the hypothesis whether the students smoking habit is independent of their exercise level at .05 significance level.

```
# Create a data frame from the main data set.

stu_data = data.frame(survey$Smoke,survey$Exer)

# Create a contingency table with the needed variables.

stu_data = table(survey$Smoke,survey$Exer)

print(stu_data)
```

Output:

```
      Freq None Some
Heavy    7   1   3
Never   87  18  84
Occas   12   3   4
Regul    9   1   7
```

And finally we apply the `chisq.test()` function to the contingency table `stu_data`.

```
# applying chisq.test() function

print(chisq.test(stu_data))
```

Output:

```
Pearson's Chi-squared test

data: stu_data

X-squared = 5.4885, df = 6, p-value = 0.4828
```

As the p-value 0.4828 is greater than the .05, we conclude that the smoking habit is independent of the exercise level of the student and hence there is a weak or no correlation between the two variables.

The complete R code is given below.

```
# R program to illustrate
```

```
# Chi-Square Test in R

library(MASS)

print(str(survey))

stu_data = data.frame(survey$Smoke,survey$Exer)

stu_data = table(survey$Smoke,survey$Exer)

print(stu_data)

print(chisq.test(stu_data))
```

So, in summary, it can be said that it is very easy to perform a Chi-square test using R. One can perform this task using **chisq.test()** function in R.

15. ANOVA testing

ANOVA also known as Analysis of variance is used to investigate relations between categorical variables and continuous variable in R Programming. It is a type of hypothesis testing for population variance.

R – ANOVA Test

ANOVA test involves setting up:

- **Null Hypothesis:** All population means are equal.
- **Alternate Hypothesis:** Atleast one population mean is different from other.

ANOVA tests are of two types:

- **One way ANOVA:** It takes one categorical group into consideration.
- **Two way ANOVA:** It takes two categorical group into consideration.

The Dataset

The mtcars(motor trend car road test) dataset is used which consist of 32 car brands and 11 attributes. The dataset comes preinstalled in **dplyr** package in R.

To get started with ANOVA, we need to install and load the **dplyr** package.

Performing One Way ANOVA test in R language

One way ANOVA test is performed using mtcars dataset which comes preinstalled with dplyr package between disp attribute, a continuous attribute and gear attribute, a categorical attribute.


```

# Installing the package

install.packages(dplyr)

# Loading the package

library(dplyr)

# Variance in mean within group and between group

boxplot(mtcars$disp~factor(mtcars$gear),

        xlab = "gear", ylab = "disp")


# Step 1: Setup Null Hypothesis and Alternate Hypothesis

# H0 =  $\mu_1 = \mu_2$  (There is no difference

# between average displacement for different gear)

# H1 = Not all means are equal

# Step 2: Calculate test statistics using aov function

mtcars_aov <- aov(mtcars$disp~factor(mtcars$gear))

summary(mtcars_aov)

# Step 3: Calculate F-Critical Value

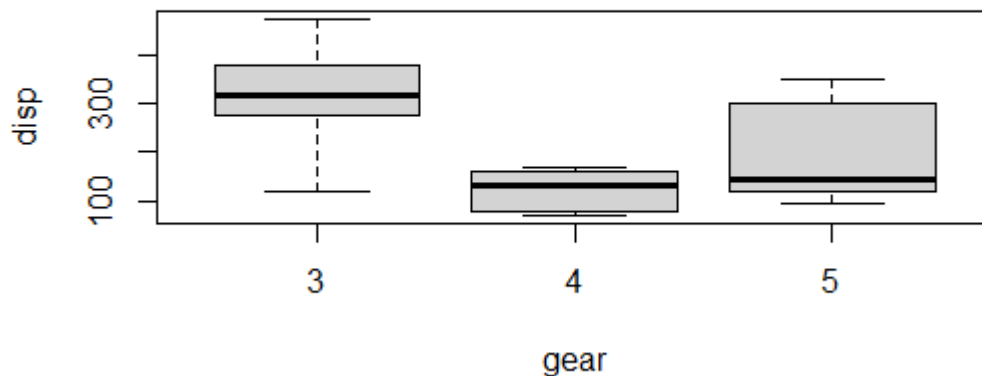
# For 0.05 Significant value, critical value =  $\alpha = 0.05$ 

# Step 4: Compare test statistics with F-Critical value

# and conclude test  $p < \alpha$ , Reject Null Hypothesis

```

Output:



The box plot shows the mean values of gear with respect of displacement. Here categorical variable is gear on which factor function is used and continuous variable is disp.

```

      Df Sum Sq Mean Sq F value    Pr(>F)
factor(mtcars$gear)  2 280221   140110    20.73 2.56e-06 ***
Residuals          29 195964     6757
---
signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

The summary shows that the gear attribute is very significant to displacement(Three stars denoting it). Also, the P value is less than 0.05, so proves that gear is significant to displacement i.e related to each other and we reject the Null Hypothesis.

Performing Two Way ANOVA test in R

Two-way ANOVA test is performed using mtcars dataset which comes preinstalled with dplyr package between disp attribute, a continuous attribute and gear attribute, a categorical attribute, am attribute, a categorical attribute.

```
# Installing the package
```

```
install.packages(dplyr)
```

```
# Loading the package
```

```
library(dplyr)
```

```
# Variance in mean within group and between group
```

```
boxplot(mtcars$disp~mtcars$gear, subset = (mtcars$am == 0),
```

```
      xlab = "gear", ylab = "disp", main = "Automatic")
```

```
boxplot(mtcars$disp~mtcars$gear, subset = (mtcars$am == 1),
```

```
      xlab = "gear", ylab = "disp", main = "Manual")
```

```
# Step 1: Setup Null Hypothesis and Alternate Hypothesis
```

```
# H0 =  $\mu_0 = \mu_{01} = \mu_{02}$ (There is no difference between
```

```
# average displacement for different gear)
```

```
# H1 = Not all means are equal
```

```
# Step 2: Calculate test statistics using aov function
```

```
mtcars_aov2 <- aov(mtcars$disp~factor(mtcars$gear) * factor(mtcars$am))
```

```
summary(mtcars_aov2)
```

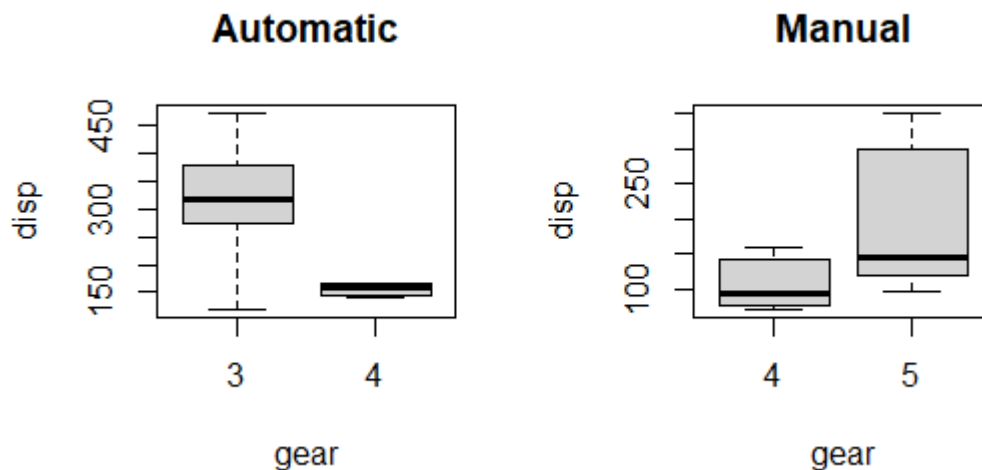
```
# Step 3: Calculate F-Critical Value
```

```
# For 0.05 Significant value, critical value =  $\alpha = 0.05$ 
```

```
# Step 4: Compare test statistics with F-Critical value
```

```
# and conclude test  $p < \alpha$ , Reject Null Hypothesis
```

Output:



The box plot shows the mean values of gear with respect of displacement. Here categorical variables are gear and am on which factor function is used and continuous variable is disp.

```

      Df Sum Sq Mean Sq F value    Pr(>F)
factor(mtcars$gear)  2 280221   140110   20.695 3.03e-06 ***
factor(mtcars$am)    1   6399     6399    0.945  0.339
Residuals          28 189565     6770
---
signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

The summary shows that gear attribute is very significant to displacement (Three stars denoting it) and am attribute is not much significant to displacement. P-value of gear is less than 0.05, so it proves that gear is significant to displacement i.e. related to each other. P-value of am is greater than 0.05, am is not significant to displacement i.e. not related to each other.

Results

We see significant results from boxplots and summaries.

- Displacement is strongly related to Gears in cars i.e. displacement is dependent on gears with $p < 0.05$.
- Displacement is strongly related to Gears but not related to transmission mode in cars with $p > 0.05$ with am.