**Practical No. 1**

**Familiarization of environments in R.**

The environment is a virtual space that is triggered when an interpreter of a programming language is launched. Simply, the environment is a collection of all the objects, variables, and functions. Or, Environment can be assumed as a top-level object that contains the set of names/variables associated with some values. In this experiment, let us discuss creating a new environment in R programming, list all environments, removing a variable from the environment, searching for a variable or function among environments and function environments with the help of examples.

The job of an environment is to associate, or bind, a set of names to a set of values.

***Why the Environment Differ from the List?***

* Every object in an environment has a name.
* The environment has a parent environment.
* Environments follow reference semantics.

***Create a New Environment***

An environment in R programming can be created using **new.env()** function. Further, the variables can be accessed using **$ or [[ ]]** operator. But, each variable is stored in different memory locations. There are four special environments: **globalenv(), baseenv(), emptyenv() and environment()**

**Syntax:** new.env(hash = TRUE)

**Parameters:**   
**hash:** indicates logical value. If TRUE, environments uses a hash table

To know about more optional parameters, use below command in console: **help(“new.env”)**

**Example**:

# R program to illustrate

# Environments in R

# Create new environment

newEnv<- new.env()

# Assigning variables

newEnv$x<- 1

newEnv$y<- “GFG”

newEnv$z<-1:10

# Print

print(newEnv$z)

**Output:**

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

**List all Environments**

Every environment has a parent environment but there is an empty environment that does not have any parent environment. All the environments can be listed using ls() function and search() function. Ls() function also list out all the bindings of the variables in a particular environment.

**Syntax**:

ls()

search()

Parameters:

These functions need no argument

**Example**:

# R program to illustrate

# Environments in R

# Prints all the bindings and environments

# attached to Global Environment

ls()

# Prints bindings of newEnv

ls(newEnv)

# Lists all the environments of the parent environment

search()

**Output**:

[1] “al” “e” “e1” “f” “newEnv” “pts” “x” “y”

[9] “z”

[1] “x” “y” “z”

[1] “.GlobalEnv” “package:stats” “package:graphics”

[4] “package:grDevices” “package:utils” “package:datasets”

[7] “package:methods” “Autoloads” “package:base”

**Removing a Variable From an Environment**

A variable in an environment is deleted using rm() function. It is different from deleting entries from lists as entries in lists are set as NULL to be deleted. But, using rm() function, bindings are removed from the environment.

**Syntax**: rm(…)

Parameters:

…: indicates list of objects

Example:

# R program to illustrate

# Environments in R

# Remove newEnv

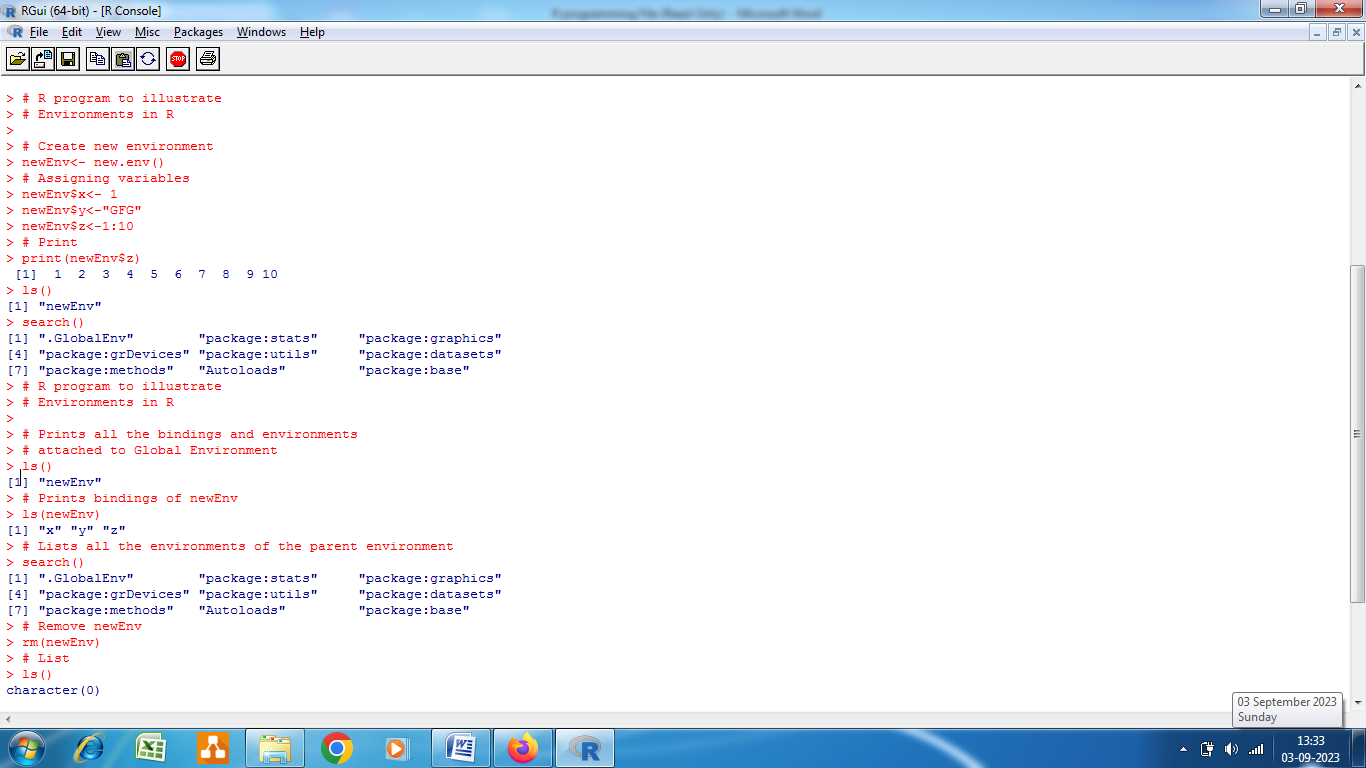
rm(newEnv)

# List

ls()

**Output**:

[1] “al” “e” “e1” “f” “pts” “x” “y” “z”



**Search a Variable or Function Among Environments**

A variable or a function can be searched in R programming by using where() function among all the environments and packages present. Where() function is present in pryr package. This function takes only two arguments, the name of the object to search for and the environment from where to start the search.

**Syntax**: where(name)

Parameters:

Name: indicates object to look for

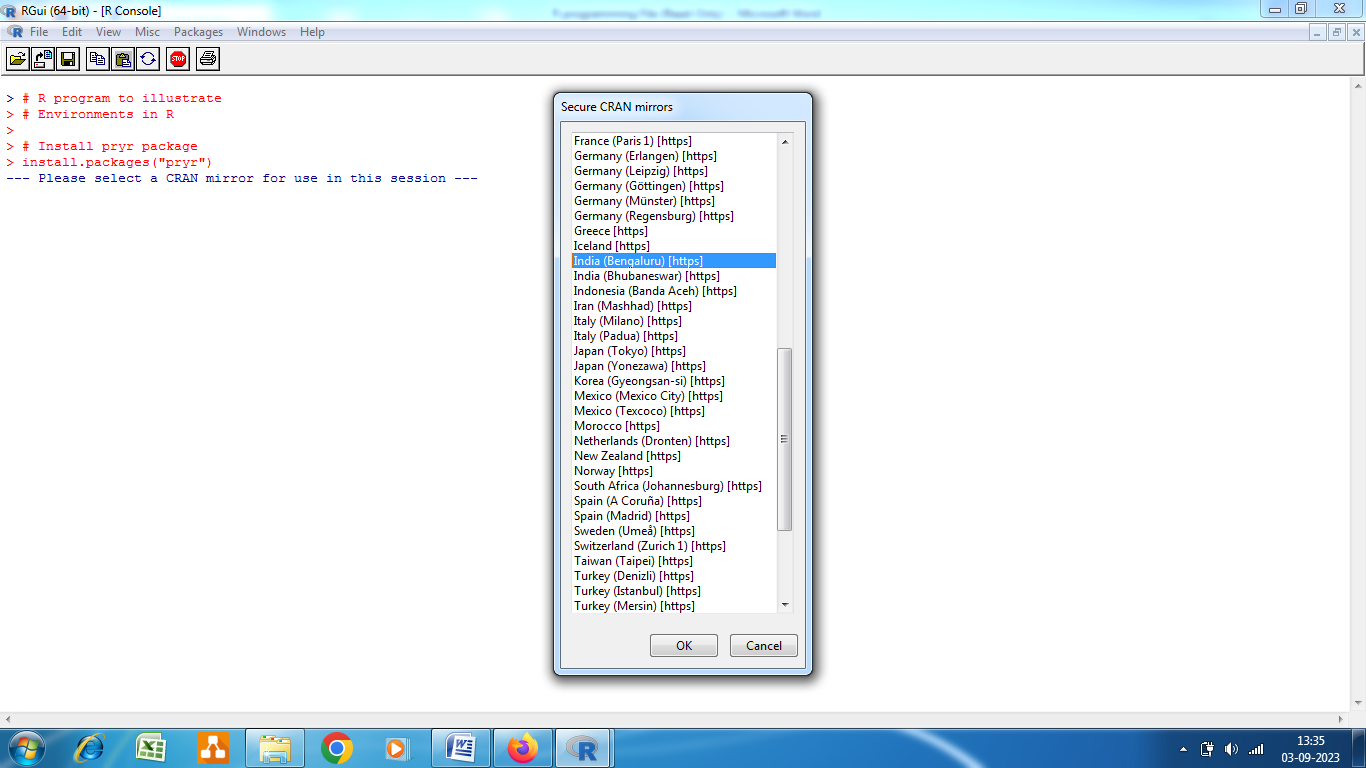
**Example**:

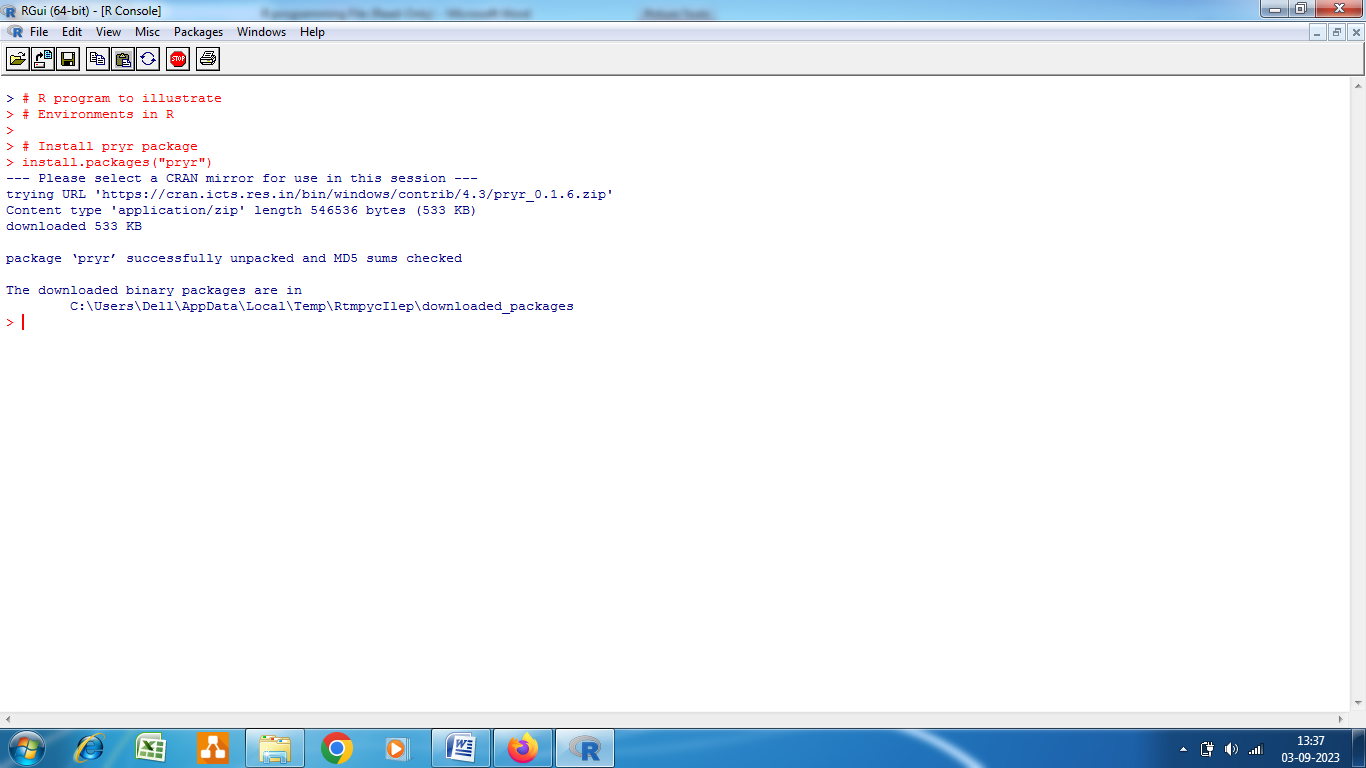
# R program to illustrate

# Environments in R

# Install pryr package

install.packages(“pryr”)





# Load the package

library(pryr)

# Search

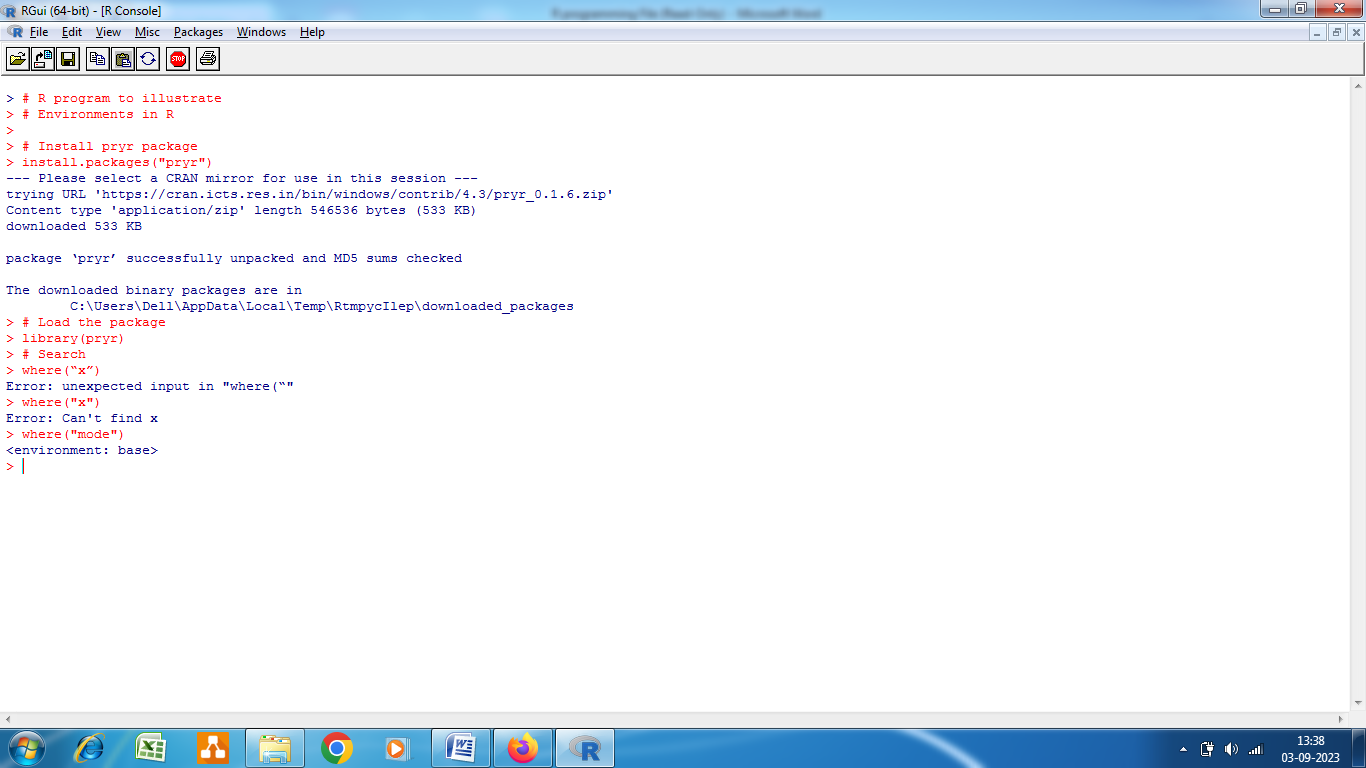
where(“x”)

where(“mode”)

**Output**:

<environment: R\_GlobalEnv>

<environment: base>



**Practical No. 2**

**Perform simple arithmetic’s using R.**

**R Operators**

Operators are the symbols directing the compiler to perform various kinds of operations between the operands. Operators simulate the various mathematical, logical, and decision operations performed on a set of Complex Numbers, Integers, and Numericals as input operands.

R supports majorly four kinds of binary operators between a set of operands. In this experiment, we will see various types of operators in R Programming language and their usage.

**Types of the operator in R language**

1. Arithmetic Operators
2. Logical Operators
3. Relational Operators
4. Assignment Operators
5. Miscellaneous Operator
6. Arithmetic Operators

Arithmetic operations in R simulate various math operations, like addition, subtraction, multiplication, division, and modulo using the specified operator between operands, which may be either scalar values, complex numbers, or vectors. The R operators are performed element-wise at the corresponding positions of the vectors.

**Addition operator (+)**

The values at the corresponding positions of both operands are added. Consider the following R operator snippet to add two vectors:

A <- c (1, 0.1)

B <- c (2.33, 4)

print (A+B)

Output : 3.33 4.10

**Subtraction Operator (-)**

The second operand values are subtracted from the first. Consider the following R operator snippet to subtract two variables:

A <- 6

B <- 8.4

print (A-B)

Output : -2.4

**Multiplication Operator (\*)**

The multiplication of corresponding elements of vectors and Integers are multiplied with the use of the ‘\*’ operator.

B= c(4,4)

C= c(5,5)

print (B\*C)

Output : 20 20

**Division Operator (/)**

The first operand is divided by the second operand with the use of the ‘/’ operator.

A <- 10

B <- 5

print (A/B)

Output : 2

**Power Operator (^)**

The first operand is raised to the power of the second operand.

A <- 4

B <- 5

Print(A^B)

Output : 1024

**Modulo Operator (%%)**

The remainder of the first operand divided by the second operand is returned.

List1<- c(2, 22)

List2<-c(2,4)

print(List1 %% List2)

Output : 0 2

1. **Logical Operators**

Logical operations in R simulate element-wise decision operations, based on the specified operator between the operands, which are then evaluated to either a True or False boolean value. Any non-zero integer value is considered as a TRUE value, be it a complex or real number.

**Element-wise Logical AND operator (&):**

Returns True if both the operands are True.

List1 <- c(TRUE, 0.1)

List2 <- c(0,4+3i)

print(List1 & List2)

Output : FALSE TRUE

Any non zero integer value is considered as a TRUE value, be it complex or real number.

**Element-wise Logical OR operator (|):**

Returns True if either of the operands is True.

List1 <- c(TRUE, 0.1)

List2 <- c(0,4+3i)

print(List1|List2)

Output : TRUE TRUE

**NOT operator (!):**

A unary operator that negates the status of the elements of the operand.

List1 <- c(0,FALSE)

print(!List1)

Output : TRUE TRUE

**Logical AND operator (&&):**

Returns True if both the first elements of the operands are True.

List1 <- c(TRUE, 0.1)

List2 <- c(0,4+3i)

print(List1 && List2)

Output : FALSE

Compares just the first elements of both the lists.

**Logical OR operator (||):**

Returns True if either of the first elements of the operands is True.

List1 <- c(TRUE, 0.1)

List2 <- c(0,4+3i)

print(List1||List2)

Output : TRUE

Output : FALSE FALSEFALSE

**Greater than equal to (>=)**

Returns TRUE if the corresponding element of the first operand is greater or equal to that of the second operand. Else returns FALSE.

List1 <- c(TRUE, 0.1, “apple”)

List2 <- c(TRUE, 0.1, “bat”)

print(List1 >= List2)

Output : TRUE TRUE FALSE

**Not equal to (!=) :**

Returns TRUE if the corresponding element of the first operand is not equal to the second operand. Else returns FALSE.

List1 <- c(TRUE, 0.1,’apple’)

List2 <- c(0,0.1,”bat”)

print(List1!=List2)

Output : TRUE FALSE TRUE

**Assignment Operators**

Assignment operators in R are used to assigning values to various data objects in R. The objects may be integers, vectors, or functions. These values are then stored by the assigned variable names. There are two kinds of assignment operators: Left and Right

**Left Assignment (<- or <<- or =)**

Assigns a value to a vector.

Vec1 = c(“ab”, TRUE)

print (vec1)

Output : “ab” “TRUE”

**Right Assignment (-> or ->>)**

Assigns value to a vector.

c(“ab”, TRUE) ->> vec1

print (vec1)

Output : “ab” “TRUE”

**# R program to illustrate**

**# the use of Miscellaneous operators**

Mat <- matrix (1:4, nrow = 1, ncol = 4)

print(“Matrix elements using : “)

print(Mat)

Product = Mat %\*% t(Mat)

print(“Product of matrices”)

print(Product,)

cat (“does 1 exist in prod matrix :”, “1” %in% Product)

Output

[1] “Matrix elements using : “

[,1] [,2] [,3] [,4]

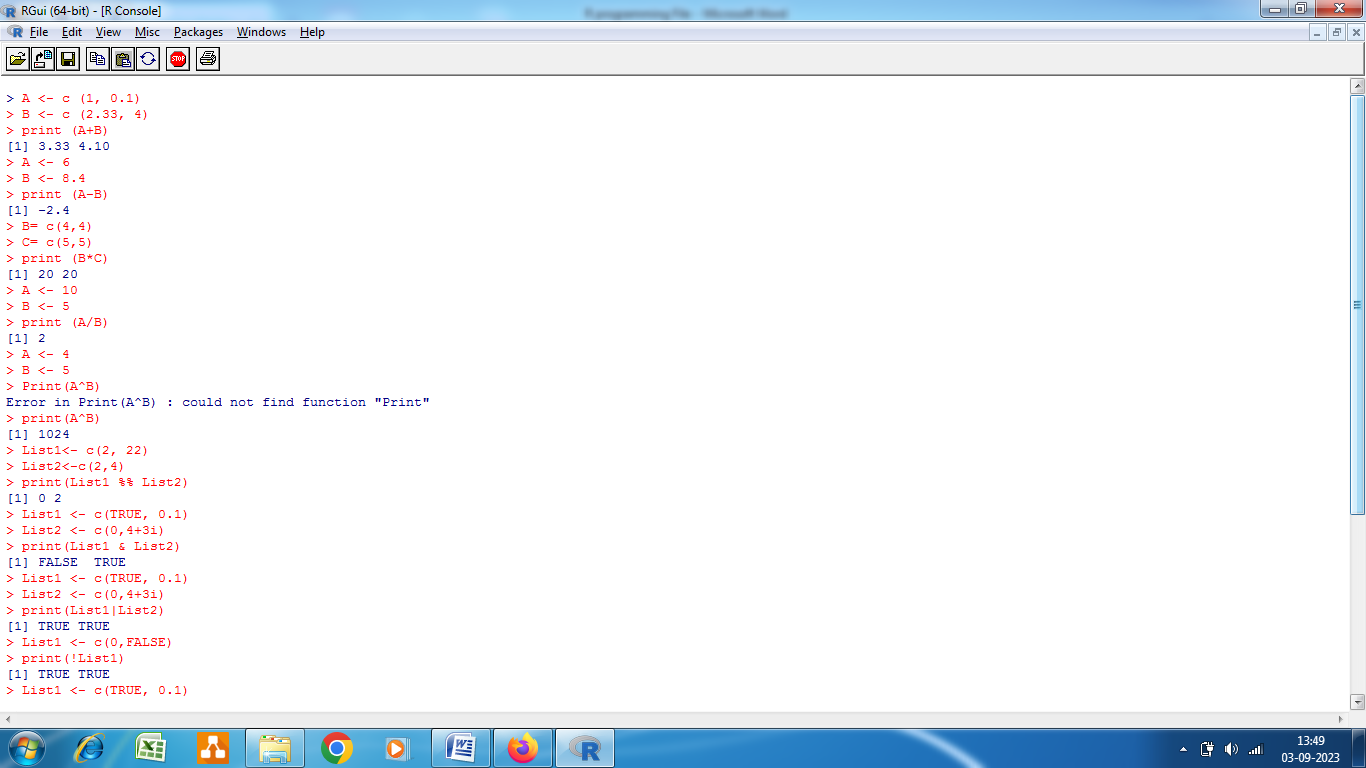
[1,] 1 2 3 4

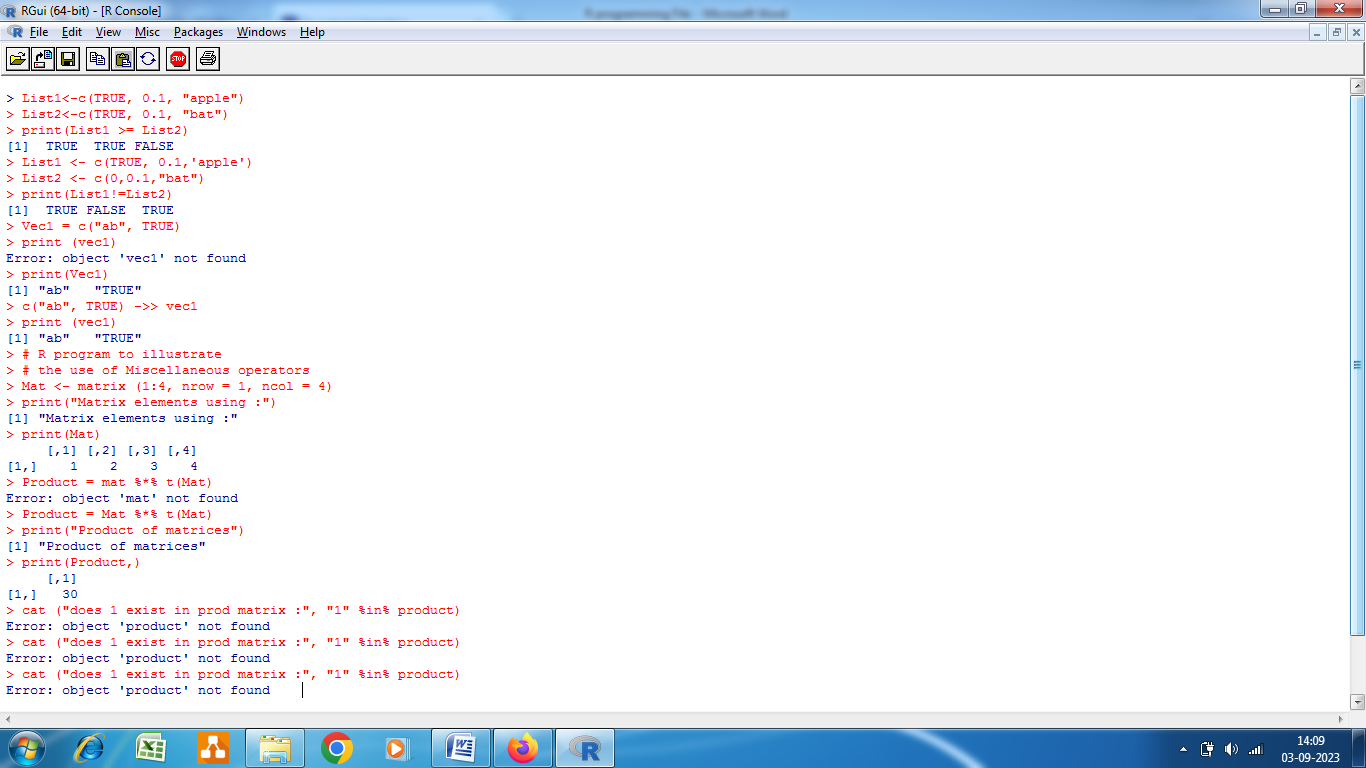
[1] “Product of matrices”

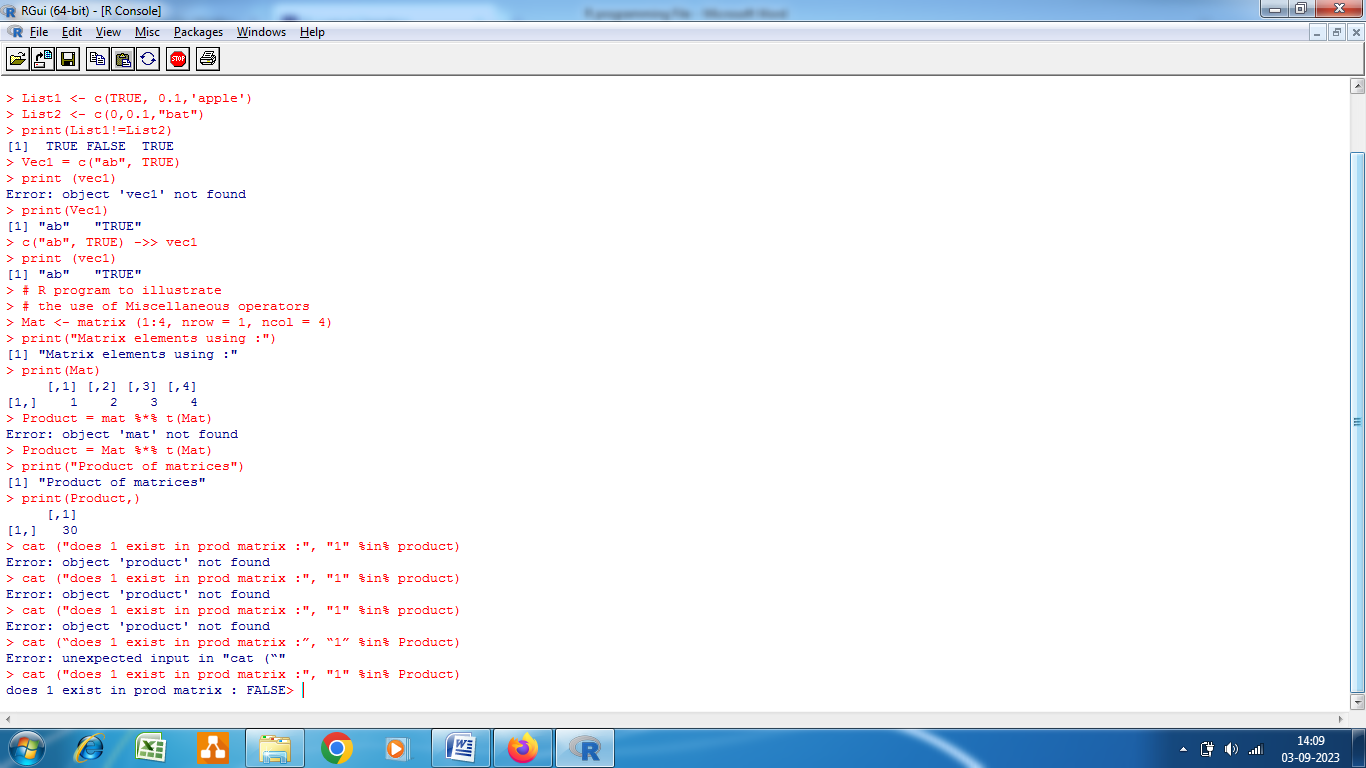
[,1]

[1,] 30

Does 1 exist in prod matrix : FALSE





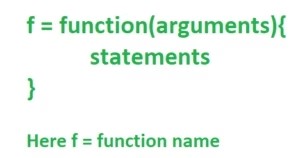


**Practical No. 3**

**Perform basic R Functions.**

Functions are useful when you want to perform a certain task multiple times. A function accepts input arguments and produces the output by executing valid R commands that are inside the function. In R Programming Language when you are creating a function the function name and the file in which you are creating the function need not be the same and you can have one or more functions in R.

**Creating a Function in R**

Functions are created in R by using the command function(). The general structure of the function file is as follows:

**Types of Function in R Language**

**Built-in Function:** Built-in functions in R are pre-defined functions that are available in R programming languages to perform common tasks or operations.

**User-defined Function**: R language allow us to write our own function.

**Basic Functions in R:**

library(datasets) #Load Data

data<-iris$Sepal.Length

**Statistical Functions**

The following commands can be used to get the mean, median, quantiles, minimum, maximum, variance, and standard deviation.

mean(data)

median(data)

sd(data)

var(data)

max(data)

min(data)

quantile(data)

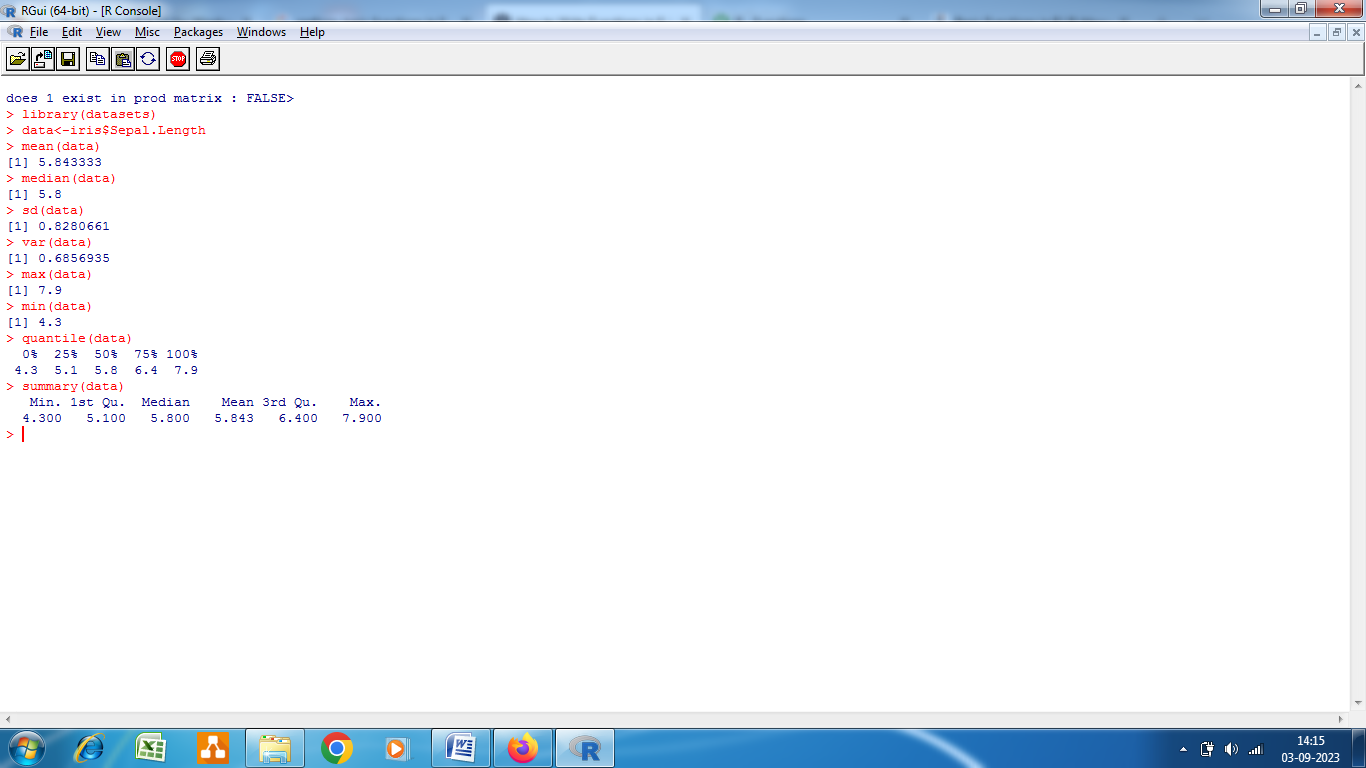
summary(data)

the summary function will provide the output based on the type of the dataset.

**Built-in Functions in R:**

There are plenty of helpful built-in functions in R used for various purposes. Some of the most popular ones are:

* min(), max(), mean(), median() – return the minimum / maximum / mean / median value of a numeric vector, correspondingly
* sum() – returns the sum of a numeric vector
* range() – returns the minimum and maximum values of a numeric vector
* abs() – returns the absulute value of a number
* str() – shows the structure of an R object
* print() – displays an R object on the console
* ncol() – returns the number of columns of a matrix or a dataframe
* length() – returns the number of items in an R object (a vector, a list, etc.)
* nchar() – returns the number of characters in a character object
* sort() – sorts a vector in ascending or descending (decreasing=TRUE) order
* exists() – returns TRUE or FALSE depending on whether or not a variable is defined in the R environment



**Creating a Function in R**

While applying built-in functions facilitates many common tasks, often we need to create our own function to automate the performance of a particular task. To declare a user-defined function in R, we use the keyword function. The syntax is as follows:

function\_name <- function(parameters){

function body

}

Above, the main components of an R function are: **function name**, **function parameters**, and **function body**. Let's take a look at each of them separately.

**Function Name**

This is the name of the function object that will be stored in the R environment after the function definition and used for calling that function. It should be concise but clear and meaningful so that the user who reads our code can easily understand what exactly this function does.

For example, if we need to create a function for calculating the circumference of a circle with a known radius, we'd better call this function circumference rather than function\_1 or circumference\_of\_a\_circle.

***Function Parameters***

Sometimes, they are called **formal arguments**. Function parameters are the variables in the function definition placed inside the parentheses and separated with a comma that will be set to actual values (called **arguments**) each time we call the function. For example:

circumference <- function(r){

2\*pi\*r

}

print(circumference(2))

[1] 12.56637

Above, we created a function to calculate the circumference of a circle with a known radius using the formula , so the function has the only parameter r. After defining the function, we called it with the radius equal to 2 (hence, with the argument 2).

It's possible, even though rarely useful, for a function to have no parameters:

hello\_world <- function(){

'Hello, World!'

}

print(hello\_world())

[1] "Hello, World!"

Also, some parameters can be set to default values (those related to a typical case) inside the function definition, which then can be reset when calling the function. Returning to our circumference function, we can set the default radius of a circle as 1, so if we call the function with no argument passed, it will calculate the circumference of a unit circle (i.e., a circle with a radius of 1). Otherwise, it will calculate the circumference of a circle with the provided radius:

circumference <- function(r=1){

2\*pi\*r

}

print(circumference())

print(circumference(2))

[1] 6.283185

[1] 12.56637

**Function Body**

The function body is a set of commands inside the curly braces that are run in a predefined order every time we call the function. In other words, in the function body, we place what exactly we need the function to do:

sum\_two\_nums <- function(x, y){

x + y

}

print(sum\_two\_nums(1, 2))

[1] 3

Note that the statements in the function body (in the above example – the only statement x + y) should be indented by 2 or 4 spaces, depending on the IDE where we run the code, but the important thing is to be consistent with the indentation throughout the program. While it doesn't affect the code performance and isn't obligatory, it makes the code easier to read.

It's possible to drop the curly braces if the function body contains a single statement. For example:

sum\_two\_nums <- function(x, y) x + y

print(sum\_two\_nums(1, 2))

[1] 3

As we saw from all the above examples, in R, it usually isn't necessary to explicitly include the return statement when defining a function since an R function just automatically returns the last evaluated expression in the function body. However, we still can add the return statement inside the function body using the syntax return(expression\_to\_be\_returned). This becomes inevitable if we need to return more than one result from a function. For example:

mean\_median <- function(vector){

mean <- mean(vector)

median <- median(vector)

return(c(mean, median))

}

print(mean\_median(c(1, 1, 1, 2, 3)))

[1] 1.6 1.0

Note that in the return statement above, we actually return a **vector** containing the necessary results, and not just the variables separated by a comma (since the return() function can return only a single R object). Instead of a vector, we could also return a list, especially if the results to be returned are supposed to be of different data types.

**Calling a Function in R**

In all the above examples, we actually already called the created functions many times. To do so, we just put the punction name and added the necessary arguments inside the parenthesis. In R, function arguments can be passed by position, by name (so-called *named arguments*), by mixing position-based and name-based matching, or by omitting the arguments at all.

If we pass the arguments *by position*, we need to follow the same sequence of arguments as defined in the function:

subtract\_two\_nums <- function(x, y){

x - y

}

print(subtract\_two\_nums(3, 1))

[1] 2

In the above example, x is equal to 3 and y – to 1, and not vice versa.

If we pass the arguments *by name*, i.e., explicitly specify what value each parameter defined in the function takes, the order of the arguments doesn't matter:

subtract\_two\_nums <- function(x, y){

x - y

}

print(subtract\_two\_nums(x=3, y=1))

print(subtract\_two\_nums(y=1, x=3))

[1] 2

[1] 2

Since we explicitly assigned x=3 and y=1, we can pass them either as x=3, y=1 or y=1, x=3 – the result will be the same.

It's possible to *mix position- and name-based matching* of the arguments. Let's look at the example of the function for calculating BMR (basal metabolic rate), or daily consumption of calories, for women based on their weight (in kg), height (in cm), and age (in years). The formula that will be used in the function is the [Mifflin-St Jeor equation](https://en.wikipedia.org/wiki/Basal_metabolic_rate):

calculate\_calories\_women <- function(weight, height, age){

(10 \* weight) + (6.25 \* height) - (5 \* age) - 161

}

Now, let's calculate the calories for a woman 30 years old, with a weight of 60 kg and a height of 165 cm. However, for the age parameter, we'll pass the argument by name and for the other two parameters, we'll pass the arguments by position:

print(calculate\_calories\_women(age=30, 60, 165))

[1] 1320.25

In the case like above (when we mix matching by name and by position), the named arguments are extracted from the whole succession of arguments and are matched first, while the rest of the arguments are matched by position, i.e., in the same order as they appear in the function definition. However, this practice isn't recommended and can lead to confusion.

Finally, we can omit some (or all) of the arguments at all. This can happen if we set some (or all) of the parameters to default values inside the function definition. Let's return to our calculate\_calories\_women function and set the default age of a woman as 30 y.o.:

calculate\_calories\_women <- function(weight, height, age=30){

(10 \* weight) + (6.25 \* height) - (5 \* age) - 161

}

print(calculate\_calories\_women(60, 165))

[1] 1320.25

In the above example, we passed only two arguments to the function, despite it having three parameters in its definition. However, since one of the parameters has a default value assigned to it when we pass two arguments to the function, R interprets that the third missing argument should be set to its default value and makes the calculations accordingly, without throwing an error.

When calling a function, we usually assign the result of this operation to a variable, to be able to use it later:

circumference <- function(r){

2\*pi\*r

}

circumference\_radius\_5 <- circumference(5)

print(circumference\_radius\_5)

[1] 31.41593

**Using Functions Inside Other Functions**

Inside the definition of an R function, we can use other functions. We've already seen such an example earlier, when we used the built-in mean() and median() functions inside a user-defined function mean\_median:

mean\_median <- function(vector){

mean <- mean(vector)

median <- median(vector)

return(c(mean, median))

}

It's also possible to pass the output of calling one function directly as an argument to another function:

radius\_from\_diameter <- function(d){

d/2

}

circumference <- function(r){

2\*pi\*r

}

print(circumference(radius\_from\_diameter(4)))

[1] 12.56637

In the above piece of code, we created two simple functions first: for calculating the radius of a circle given its diameter and for calculating the circumference of a circle given its radius. Since originally we knew only the diameter of a circle (equal to 4), we called the radius\_from\_diameter function inside the circumference function to calculate first the radius from the provided value of diameter and then calculate the circumference of the circle. While this approach can be useful in many cases, we should be careful with it and avoid passing too many functions as arguments to other functions since it can affect the code readability.

Finally, functions can be *nested*, meaning that we can define a new function inside another function. Let's say that we need a function that sums up the circle areas of 3 non-intersecting circles:

sum\_circle\_ares <- function(r1, r2, r3){

circle\_area <- function(r){

pi\*r^2

}

circle\_area(r1) + circle\_area(r2) + circle\_area(r3)

}

print(sum\_circle\_ares(1, 2, 3))

[1] 43.9823

Above, we defined the circle\_area function inside the sum\_circle\_ares function. We then called that inner function three times (circle\_area(r1), circle\_area(r2), and circle\_area(r3)) inside the outer function to calculate the area of each circle for further summing up those areas. Now, if we try to call the circle\_area function outside the sum\_circle\_ares function, the program throws an error, because the inner function exists and works only inside the function where it was defined:

print(circle\_area(10))

Error in circle\_area(10): could not find function "circle\_area"

Traceback:

1. print(circle\_area(10))

When nesting functions, we have to keep in mind two things:

1. Similar to creating any function, the inner function is supposed to be used at least 3 times inside the outer function. Otherwise, it isn't viable to create it.
2. If we want to be able to use the function independent of the bigger function, we should create it outside the bigger function instead of nesting these functions. For example, if we were going to use the circle\_area function outside the sum\_circle\_ares function, we would write the following code:

circle\_area <- function(r){

pi\*r^2

}

sum\_circle\_ares <- function(r1, r2, r3){

circle\_area(r1) + circle\_area(r2) + circle\_area(r3)

}

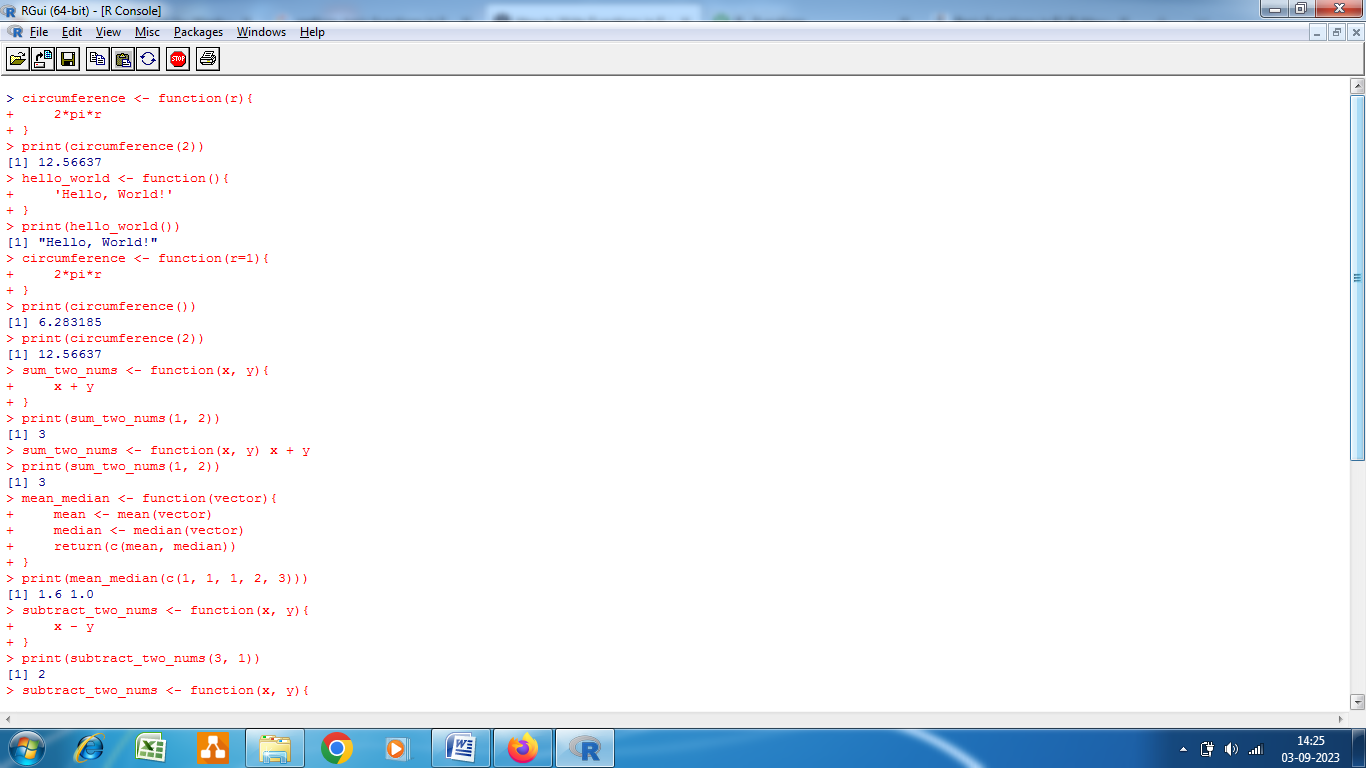
print(sum\_circle\_ares(1, 2, 3))

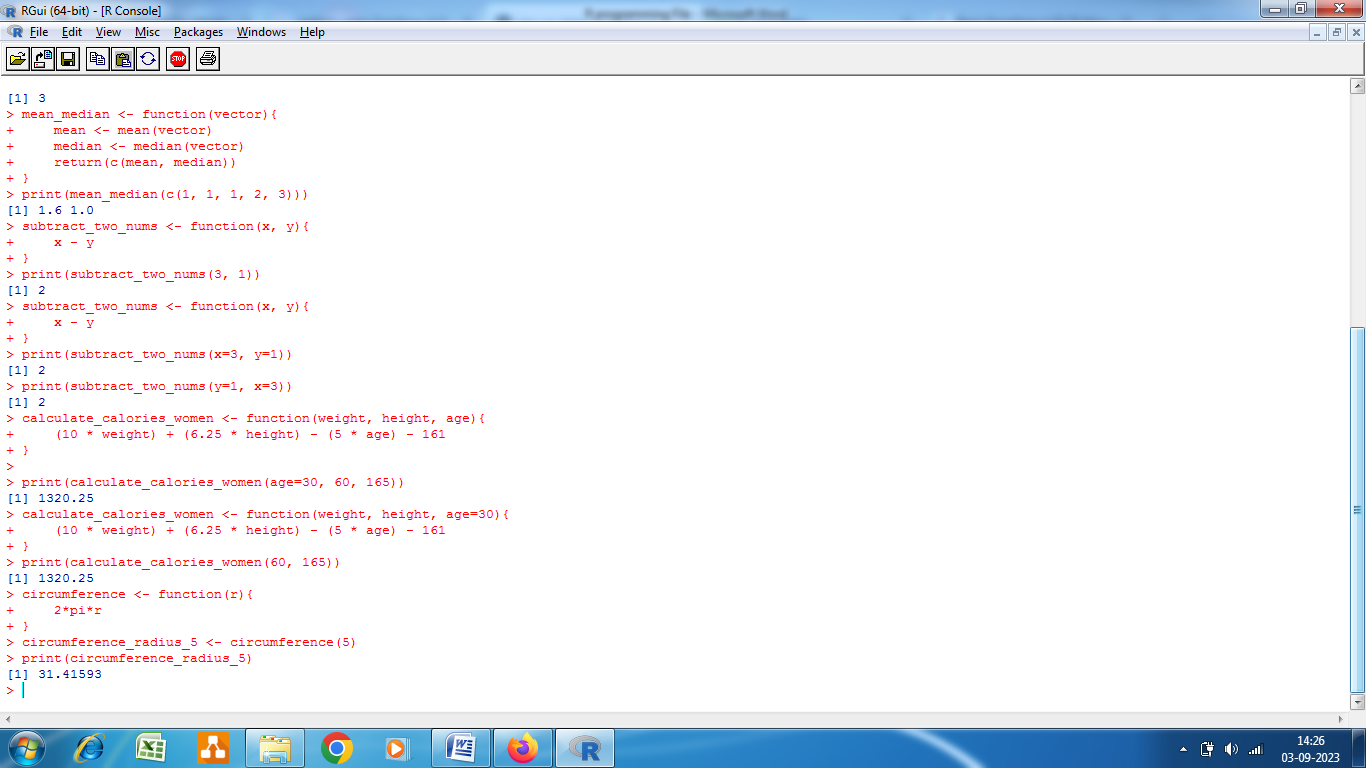
print(circle\_area(10))

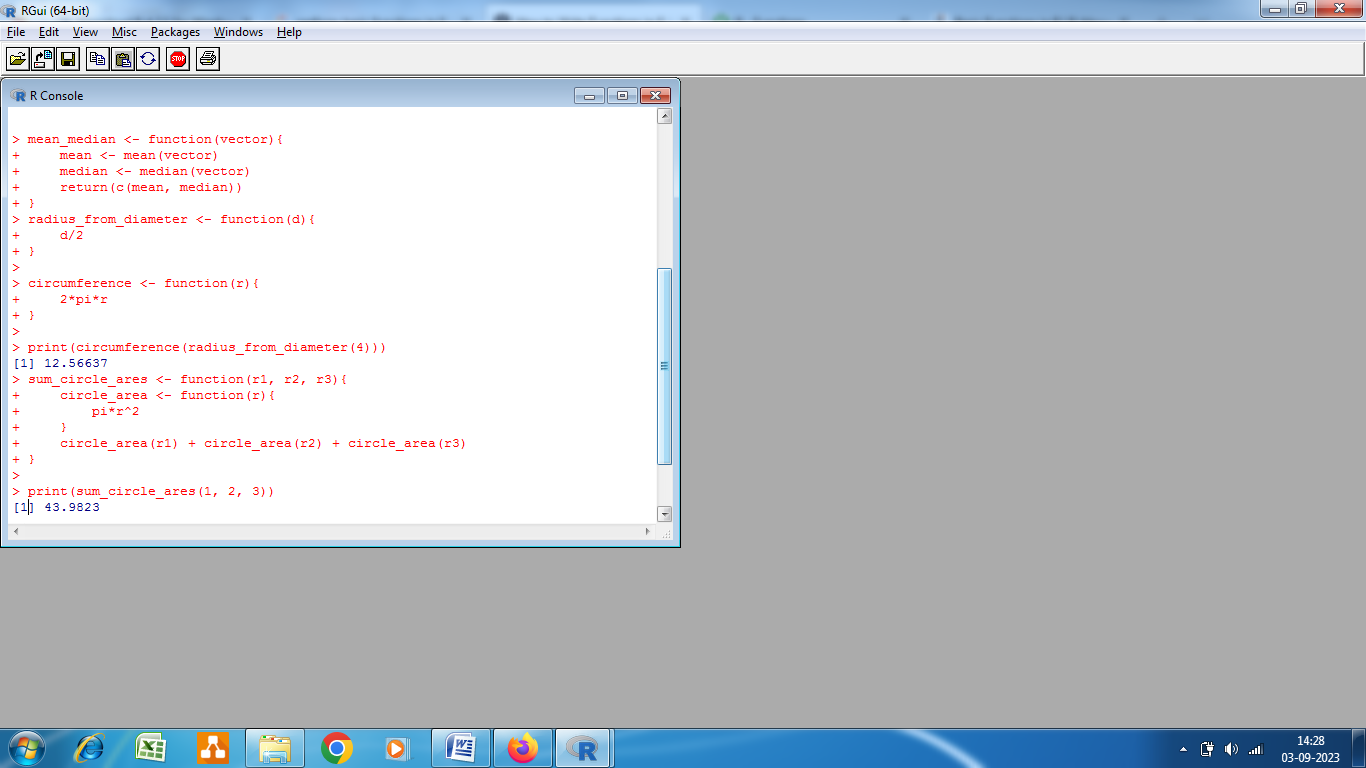
[1] 43.9823

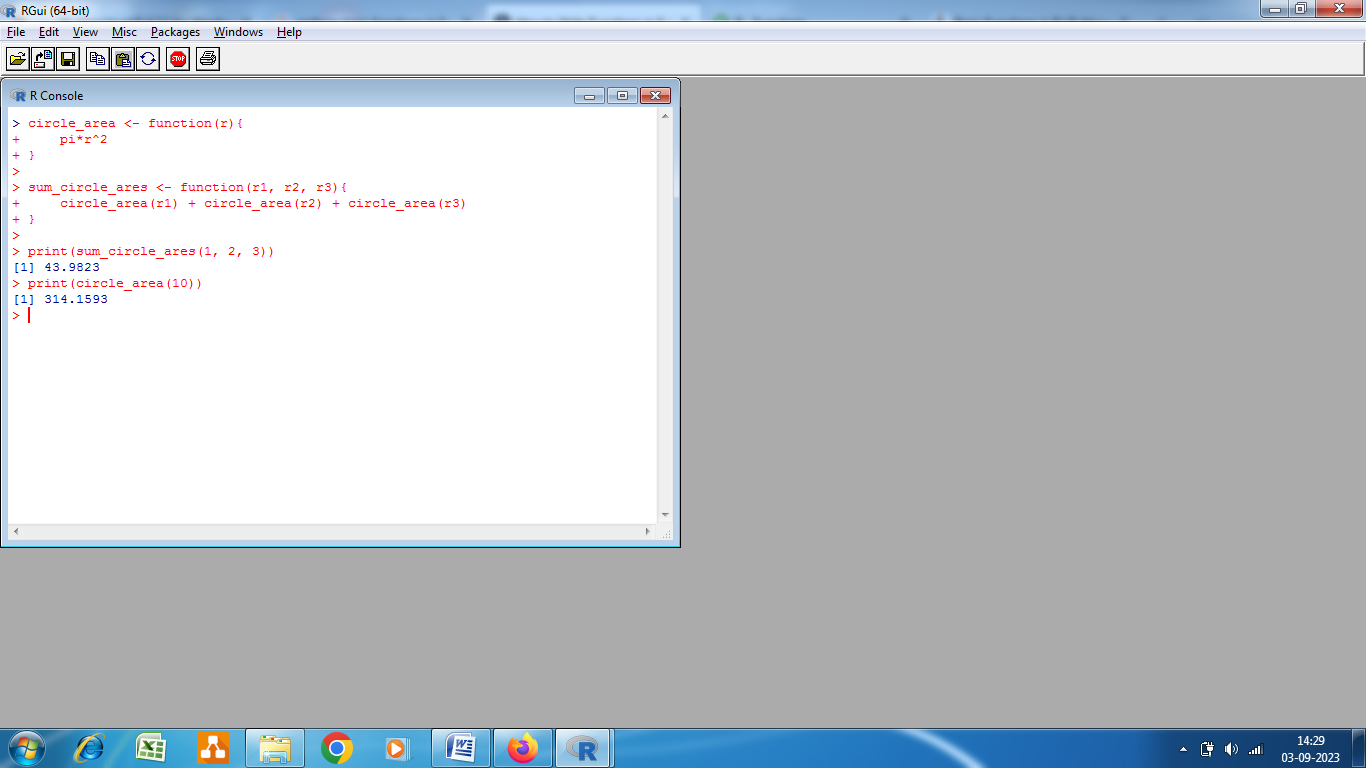
[1] 314.1593

Here, we go again used the circle\_area function inside the sum\_circle\_ares function. However, this time, we were also able to call it outside that function and get the result rather than an error.









**Practical No. 4.**

**use various graphical techniques in EDA**

**Exploratory Data Analysis or EDA** is a statistical approach or technique for analyzing data sets in order to summarize their important and main characteristics generally by using some visual aids. The EDA approach can be used to gather knowledge about the following aspects of data:

* Main characteristics or features of the data.
* The variables and their relationships.
* Finding out the important variables that can be used in our problem.

**EDA is an iterative approach that includes:**

* Generating questions about our data
* Searching for the answers by using visualization, transformation, and modeling of our data.
* Using the lessons that we learn in order to refine our set of questions or to generate a new set of questions.

**Exploratory Data Analysis in R**

In R Language, we are going to perform EDA under two broad classifications:

* **Descriptive Statistics,** which includes mean, median, mode, inter-quartile range, and so on.
* **Graphical Methods,** which includes histogram, density estimation, box plots, and so on.

> class <- c("numeric", "character", "factor", "numeric", "numeric")

> pollution <- read.csv("data/avgpm25.csv", colClasses = class)

> head(pollution)

pm25 fips region longitude latitude

1 9.771185 01003 east -87.74826 30.59278

2 9.993817 01027 east -85.84286 33.26581

3 10.688618 01033 east -87.72596 34.73148

4 11.337424 01049 east -85.79892 34.45913

5 12.119764 01055 east -86.03212 34.01860

6 10.827805 01069 east -85.35039 31.18973

> str(pollution)

'data.frame': 576 obs. of 5 variables:

$ pm25 : num 9.77 9.99 10.69 11.34 12.12 ...

$ fips : chr "01003" "01027" "01033" "01049" ...

$ region : Factor w/ 2 levels "east","west": 1 1 1 1 1 1 1 1 1 1 ...

$ longitude: num -87.7 -85.8 -87.7 -85.8 -86 ...

$ latitude : num 30.6 33.3 34.7 34.5 34 ...

## Five Number Summary

A five-number summary can be computed with the fivenum() function, which takes a vector of numbers as input. Here, we compute a five-number summary of the PM2.5 data in the pollution dataset.

> fivenum(pollution$pm25)

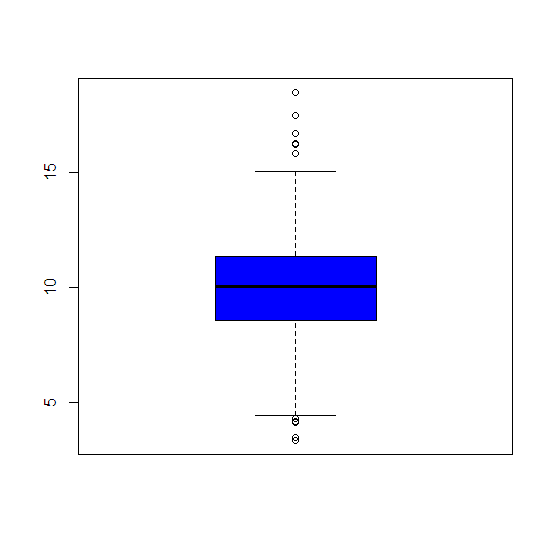
[1] 3.382626 8.547590 10.046697 11.356829 18.440731

**Boxplot**

**Box Plot**

Boxplots are a visual representation of the five-number summary plus a bit more information. In particular, boxplots commonly plot outliers that go beyond the bulk of the data. This is implemented via the boxplot() function

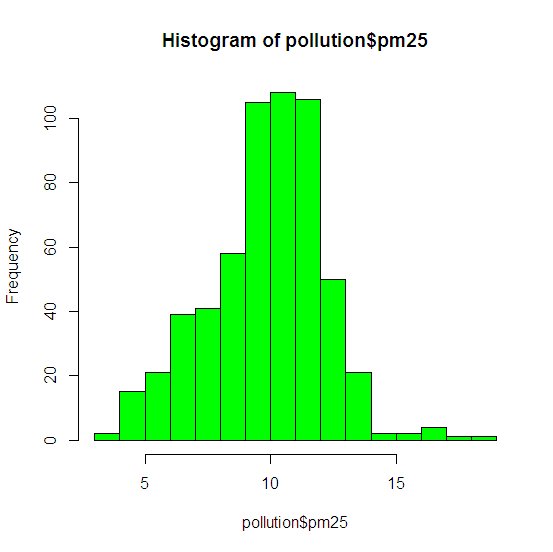
> boxplot(pollution$pm25, col = "blue")



**Histogram**

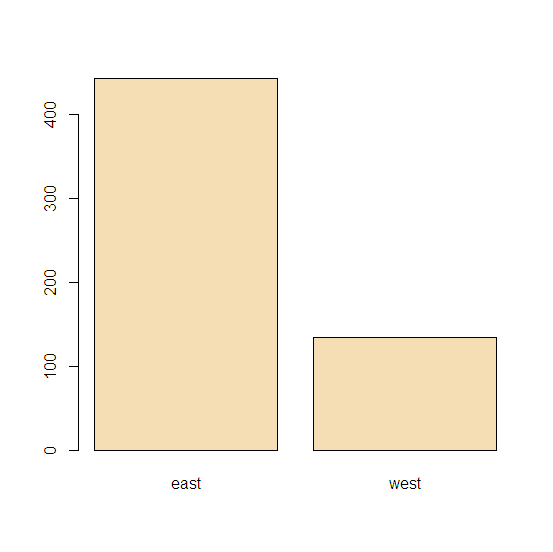
Histograms show the complete empirical distribution of the data, beyond the five data points shown by the boxplots. Here, you can easily check skewwness of the data, symmetry, multi-modality, and other features. The hist() function makes a histogram, and a handy function to go with it sometimes is the rug() function.

> hist(pollution$pm25, col = "green")



Barplot

Barplots are useful for visualizing categorical data, with the number of entries for each category being proportional to the height of the bar. Think “pie chart” but actually useful. The barplot can be made with the barplot() function.



> library(dplyr)

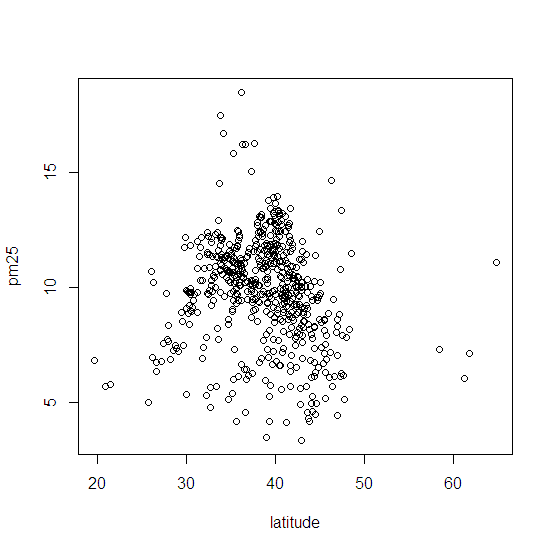
> table(pollution$region) %>% barplot(col = "wheat")

**Scatter Plot**

A scatter plot is a set of dotted points representing individual data pieces on the horizontal and vertical axis. In a graph in which the values of two variables are plotted along the X-axis and Y-axis, the pattern of the resulting points reveals a correlation between them.

> with(pollution, plot(latitude, pm25))

> abline(h = 12, lwd = 2, lty = 2)



**Experiment No. 5**

Create different charts for visualization of given set of data

**Histogram**

Histogram is basically a plot that breaks the data into bins (or breaks) and shows frequency distribution of these bins.

library(RColorBrewer)

data(VADeaths)

par(mfrow=c(2,3))

hist(VADeaths,breaks=10, col=brewer.pal(3,"Set3"),main="Set3 3 colors")

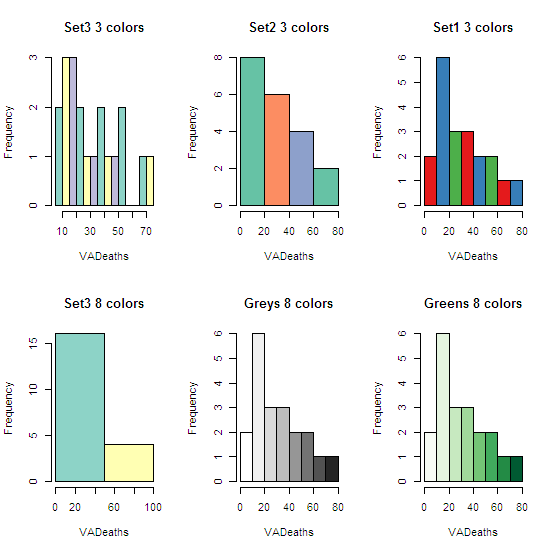
hist(VADeaths,breaks=3 ,col=brewer.pal(3,"Set2"),main="Set2 3 colors")

hist(VADeaths,breaks=7, col=brewer.pal(3,"Set1"),main="Set1 3 colors")

hist(VADeaths,,breaks= 2, col=brewer.pal(8,"Set3"),main="Set3 8 colors")

hist(VADeaths,col=brewer.pal(8,"Greys"),main="Greys 8 colors")

hist(VADeaths,col=brewer.pal(8,"Greens"),main="Greens 8 colors")

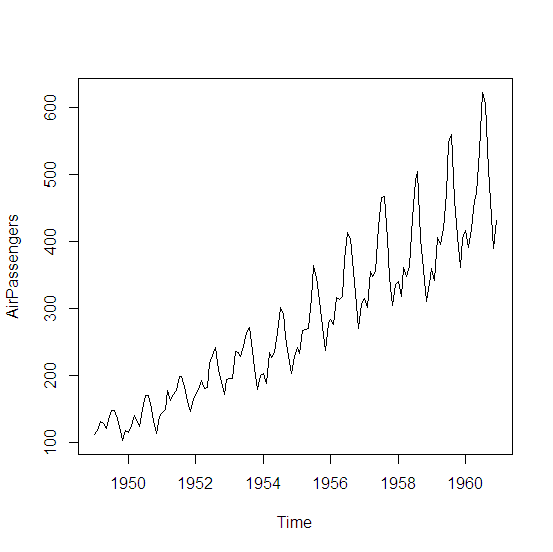


Bar/ Line Chart

Line Chart

Below is the line chart showing the increase in air passengers over given time period. Line Charts are commonly preferred when we are to analyse a trend spread over a time period. Furthermore, line plot is also suitable to plots where we need to compare relative changes in quantities across some variable (like time). Below is the code:

plot(AirPassengers,type="l")  #Simple Line Plot



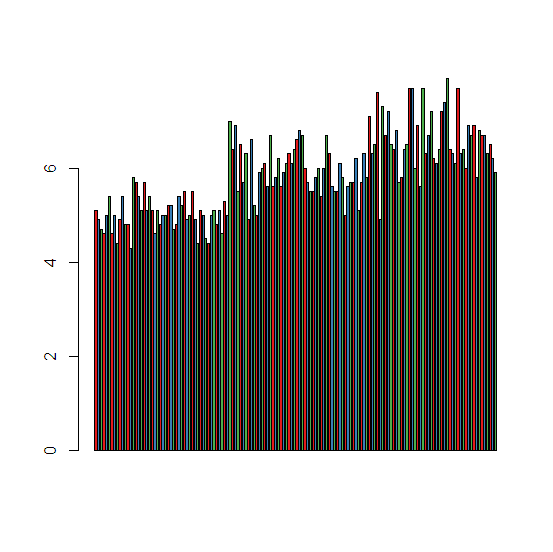
Bar Chart

Bar Plots are suitable for showing comparison between cumulative totals across several groups.  Stacked Plots are used for bar plots for various categories.

barplot(iris$Petal.Length) #Creating simple Bar Graph

barplot(iris$Sepal.Length,col  = brewer.pal(3,"Set1"))

barplot(table(iris$Species,iris$Sepal.Length),col  = brewer.pal(3,"Set1")) #Stacked Plot



**Box Plot**

The statistical summary of the given data is presented graphically using a boxplot. A boxplot depicts information like the minimum and maximum data point, the median value, first and third quartile, and interquartile range.

data(iris)

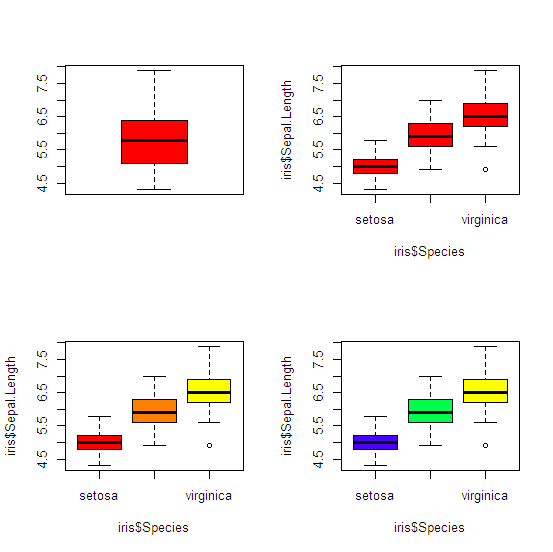
par(mfrow=c(2,2))

boxplot(iris$Sepal.Length,col="red")

boxplot(iris$Sepal.Length~iris$Species,col="red")

oxplot(iris$Sepal.Length~iris$Species,col=heat.colors(3))

boxplot(iris$Sepal.Length~iris$Species,col=topo.colors(3))

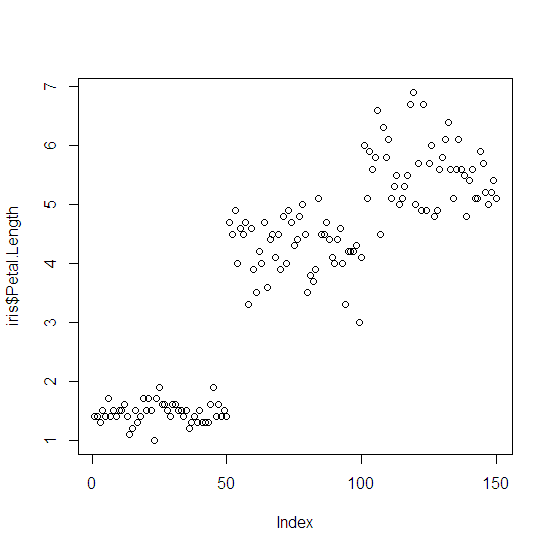


Scatter Plot

A scatter plot is a set of dotted points representing individual data pieces on the horizontal and vertical axis. In a graph in which the values of two variables are plotted along the X-axis and Y-axis, the pattern of the resulting points reveals a correlation between them.

**plot**(x=iris$Petal.Length) #Simple Scatter Plot

**plot**(x=iris$Petal.Length,y=iris$Species) #Multivariate Scatter Plot



**Heat Map**

Heatmap is defined as a graphical representation of data using colors to visualize the value of the matrix. heatmap() function is used to plot heatmap.

**Syntax:** heatmap(data)

**Parameters:** data: It represent matrix data, such as values of rows and columns

**Return:** This function draws a heatmap.

# Set seed for reproducibility

# set.seed(110)

# Create example data

data <- matrix(rnorm(50, 0, 5), nrow = 5, ncol = 5)

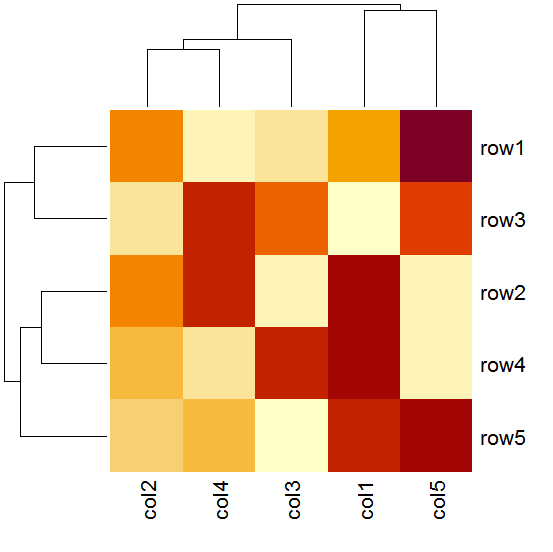
# Column names

colnames(data) <- paste0("col", 1:5)

rownames(data) <- paste0("row", 1:5)

# Draw a heatmap

heatmap(data)



Experiment No. 6

Find the mean median standard deviation and quartiles of a set of observations in r programming.

cars

**mean**(cars$speed)

**median**(cars$speed)

**sd**(cars$speed)

**quantile**(cars$speed)

Output:

[1] 15.4

[1] 15

[1] 5.287644

0% 25% 50% 75% 100%

4 12 15 19 25

Experiment No. 7

Find the Skewness and Kurtosis of a given dataset distribution.

# Required for skewness() function

library(moments)

# Defining data vector

x <- c(40, 41, 42, 43, 50)

# output to be present as PNG file

png(file = “positiveskew.png”)

# Print skewness of distribution

print(skewness(x))

# Histogram of distribution

hist(x)

# Saving the file

dev.off()

[1] 1.2099

# Required for kurtosis() function

library(moments)

# Defining data vector

x <- c(rep(61, each = 10), rep(64, each = 18),

rep(65, each = 23), rep(67, each = 32), rep(70, each = 27),

rep(73, each = 17))

# Print skewness of distribution

print(kurtosis(x))

[1] 2.258318

Experiment No. 8

Given the scenario, implement the Bayes rule by finding the posterior probability.

#define function for Bayes' Theorem

bayesTheorem <- function(pA, pB, pBA) {

pAB <- pA \* pBA / pB

return(pAB)

}

#define probabilities

pRain <- 0.2

pCloudy <- 0.4

pCloudyRain <- .85

#use function to calculate conditional probability

bayesTheorem(pRain, pCloudy, pCloudyRain)

[1] 0.425

Experiment No 9.

Find the mass function of a binomial distribution with n=20,p=0.4. Also draw the graphs of the mass function and cumulative distribution function.

success <- 0:20

dbinom(success, size=20, prob=.4)

pbinom(success, size=20, prob=.4)

plot(success, dbinom(success, size=20, prob=.4),type='h')

plot(success, pbinom(success, size=20, prob=.4),type='h')

[1] 3.656158e-05 4.874878e-04 3.087423e-03 1.234969e-02 3.499079e-02

[6] 7.464702e-02 1.244117e-01 1.658823e-01 1.797058e-01 1.597385e-01

[11] 1.171416e-01 7.099488e-02 3.549744e-02 1.456305e-02 4.854351e-03

[16] 1.294494e-03 2.696862e-04 4.230371e-05 4.700412e-06 3.298535e-07

[21] 1.099512e-08

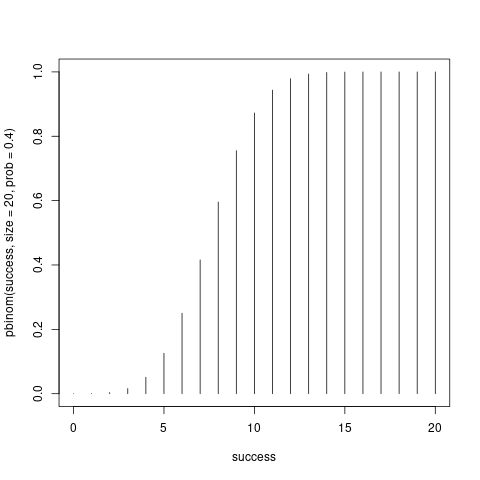
[1] 3.656158e-05 5.240494e-04 3.611472e-03 1.596116e-02 5.095195e-02

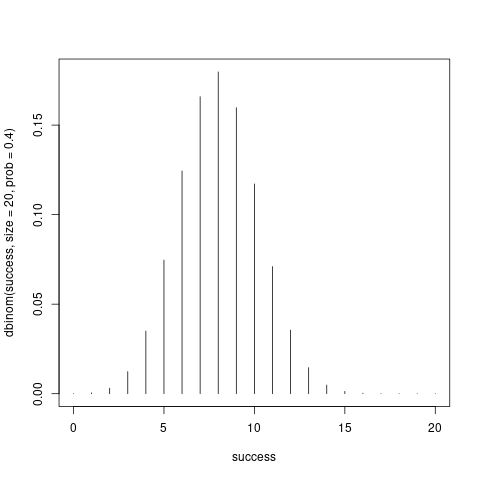
[6] 1.255990e-01 2.500107e-01 4.158929e-01 5.955987e-01 7.553372e-01

[11] 8.724788e-01 9.434736e-01 9.789711e-01 9.935341e-01 9.983885e-01

[16] 9.996830e-01 9.999527e-01 9.999950e-01 9.999997e-01 1.000000e+00

[21] 1.000000e+00





Experiment No. 10

Generate and draw the cdf and pdf of a normal distribution with mean=10 and standard

deviation=3. Use values of xx from 0 to 20 in intervals of 1.

**PDF**

# Create a sequence of numbers between -10 and 10 incrementing by 0.1.

x <- seq(0, 20, by = 1)

# Choose the mean as 2.5 and standard deviation as 0.5.

y<- dnorm(x, mean = 10, sd = 3)

y

#z<- pnorm(x, mean = 10, sd = 3#p

#z

# Give the chart file a name.

#png(file = “dnorm.png”)

plot(x,y)

#plot(x,z)

# Save the file.

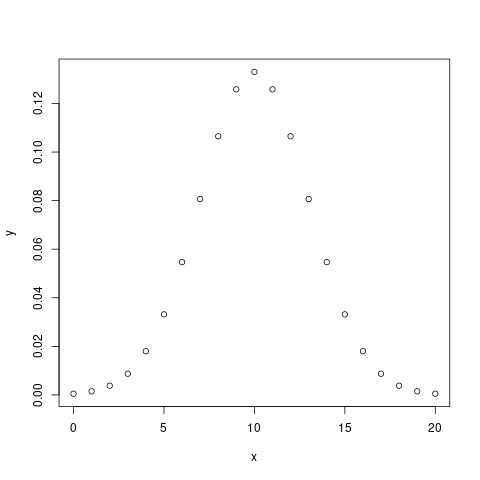
#dev.off()

[1] 0.000514093 0.001477283 0.003798662 0.008740630 0.017996989 0.033159046

[7] 0.054670025 0.080656908 0.106482669 0.125794409 0.132980760 0.125794409

[13] 0.106482669 0.080656908 0.054670025 0.033159046 0.017996989 0.008740630

[19] 0.003798662 0.001477283 0.000514093



**CDF**

# Create a sequence of numbers between -10 and 10 incrementing by 0.1.

x <- seq(0, 20, by = 1)

# Choose the mean as 2.5 and standard deviation as 0.5.

#y <- dnorm(x, mean = 10, sd = 3)

#y

z<- pnorm(x, mean = 10, sd = 3)

z

# Give the chart file a name.

#png(file = "dnorm.png")

#plot(x,y)

plot(x,z)

# Save the file.

#dev.off()

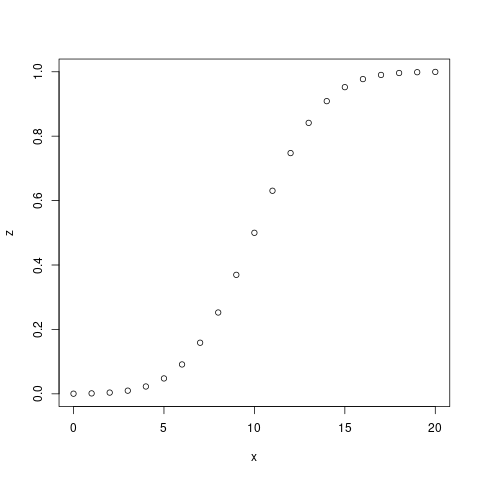
[1] 0.0004290603 0.0013498980 0.0038303806 0.0098153286 0.0227501319

[6] 0.0477903523 0.0912112197 0.1586552539 0.2524925375 0.3694413402

[11] 0.5000000000 0.6305586598 0.7475074625 0.8413447461 0.9087887803

[16] 0.9522096477 0.9772498681 0.9901846714 0.9961696194 0.9986501020

[21] 0.9995709397



Experiment No. 11

Construct a scatter plot to investigate the relationship between two variables

input <- mtcars[, c('wt', 'mpg')]

print(head(input))

# Get the input values.

input <- mtcars[, c('wt', 'mpg')]

# Plot the chart for cars with

# weight between 1.5 to 4 and

# mileage between 10 and 25.

plot(x = input$wt, y = input$mpg,

xlab = "Weight",

ylab = "Milage",

xlim = c(1.5, 4),

ylim = c(10, 25),

main = "Weight vs Milage"

)

wt mpg

Mazda RX4 2.620 21.0

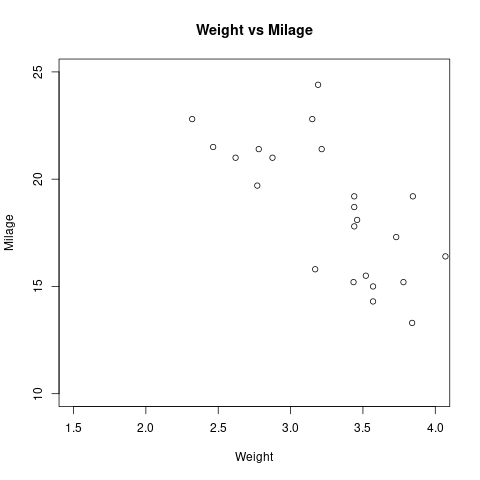
Mazda RX4 Wag 2.875 21.0

Datsun 710 2.320 22.8

Hornet 4 Drive 3.215 21.4

Hornet Sportabout 3.440 18.7

Valiant 3.460 18.1



Experiment No. 12

Perform the Z- test for single proportion, single mean etc.

# Using prop.test()

prop.test(x = 80, n = 150, p = 0.3, correct = FALSE)

1-sample proportions test without continuity correction

data: 80 out of 150, null probability 0.3

X-squared = 38.889, df = 1, p-value = 4.486e-10

alternative hypothesis: true p is not equal to 0.3

95 percent confidence interval:

0.4536625 0.6113395

sample estimates:

p

0.5333333

# Using binom.test()

binom.test(x =25, n = 100, p = 0.15)

Exact binomial test

data: 25 and 100

number of successes = 25, number of trials = 100, p-value = 0.007633

alternative hypothesis: true probability of success is not equal to 0.15

95 percent confidence interval:

0.1687797 0.3465525

sample estimates:

probability of success

0.25

Experiment No. 13

Calculate the regression coefficient and obtain the lines of regression [or the given data

library(tidyverse)

library(ggpubr)

theme\_set(theme\_pubr())

# Load the packsale

data(“marketing”, package = “datarium”)

Head(marketing, 4)

ggplot(marketing, aes(x = youtube, y = sales)) + geom\_point() + stat\_smooth()

cor(marketing$sales, marketing$youtube)

model <- lm(sales ~ youtube, data = marketing)

model

ggplot(marketing, aes(youtube, sales)) +geom\_point() +stat\_smooth(method = lm)

-- Attaching packages --------------------------------------- tidyverse 1.3.0 --

v ggplot2 3.3.2 v purrr 0.3.4

v tibble 3.0.4 v dplyr 1.0.2

v tidyr 1.1.2 v stringr 1.4.0

v readr 1.4.0 v forcats 0.5.0

-- Conflicts ------------------------------------------ tidyverse\_conflicts() --

x dplyr::filter() masks stats::filter()

x dplyr::lag() masks stats::lag()

youtube facebook newspaper sales

1 276.12 45.36 83.04 26.52

2 53.40 47.16 54.12 12.48

3 20.64 55.08 83.16 11.16

4 181.80 49.56 70.20 22.20

`geom\_smooth()` using method = 'loess' and formula 'y ~ x'

[1] 0.7822244

Call:

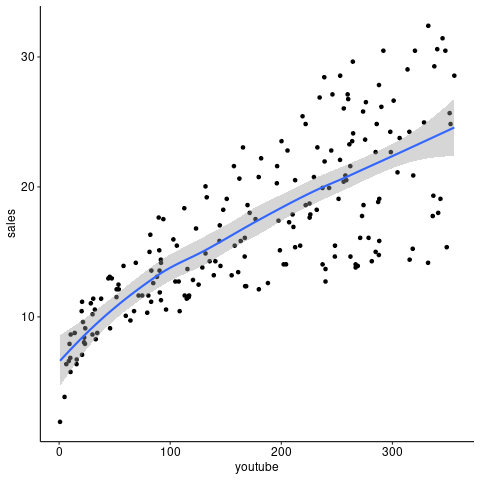
lm(formula = sales ~ youtube, data = marketing)

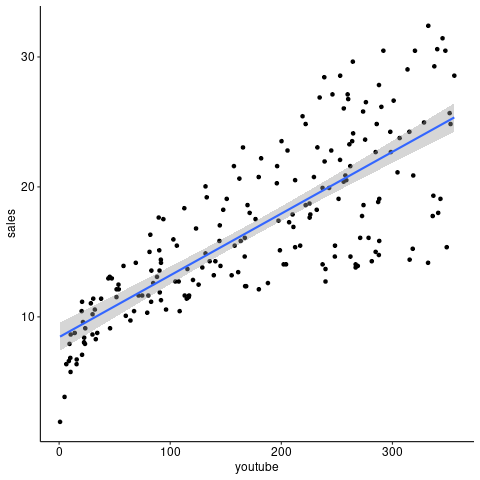
Coefficients:

(Intercept) youtube

8.43911 0.04754

`geom\_smooth()` using formula 'y ~ x'





Experiment No. 14

Compute confidence intervals for the mean when the standard deviation is known.

data("mtcars")

sample.mean <- mean(mtcars$mpg)

print(sample.mean)

sample.n <- length(mtcars$mpg)

sample.sd <- sd(mtcars$mpg)

sample.se <- sample.sd/sqrt(sample.n)

print(sample.se)

alpha = 0.05

degrees.freedom = sample.n - 1

t.score = qt(p=alpha/2, df=degrees.freedom,lower.tail=F)

print(t.score)

margin.error <- t.score \* sample.se

lower.bound <- sample.mean - margin.error

upper.bound <- sample.mean + margin.error

print(c(lower.bound,upper.bound))

[1] 20.09062

[1] 1.065424

[1] 2.039513

[1] 17.91768 22.26357

Or

# Calculate the mean and standard error

l.model <- lm(mpg ~ 1, mtcars)

# Calculate the confidence interval

confint(l.model, level=0.95)

2.5 % 97.5 %

(Intercept) 17.91768 22.26357

Experiment No. 15.

Perform F test

# Taking two samples

x <- rnorm(249, mean = 20)

y <- rnorm(79, mean = 30)

# var test in R

var.test(x, y, alternative = "two.sided")

F test to compare two variances

data: x and y

F = 0.67906, num df = 248, denom df = 78, p-value = 0.02751

alternative hypothesis: true ratio of variances is not equal to 1

95 percent confidence interval:

0.4647713 0.9584937

sample estimates:

ratio of variances

0.679058

Experiment No. 16

Perform Chi-Square test.

# 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))

'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

Freq None Some

Heavy 7 1 3

Never 87 18 84

Occas 12 3 4

Regul 9 1 7

Pearson's Chi-squared test

data: stu\_data

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

Warning message:

In chisq.test(stu\_data) : Chi-squared approximation may be incorrect