1. What is vector? Explain its function with example.

Vector:

- Vector is a collection of elements, all of the same data type.
- In R a vector can't be of mixed type.
- To create a vector in R **`c()`** function is used.
- Here, in `c()` function "c" stands for combine.

Syntax: `c(x, …)`

✓ Here we can pass the multiple objects.

Example: `a <- c("a", "b", "c")`

✓ In above example the character vector should be created and in is stores in the variable a.

- If any character element is there in a vector then all other element of vector are automatically converted to character.

Example: `a <- c("a", 1, TRUE)`

✓ In above example the element "1", which is numeric and element "TRUE", which is logical, they both are converted to character.

To create a character vector: `a <- c("a", "b", "c")`
To create a numeric vector: `a <- c(1, 2, 3)`
To create a logical vector: `a <- c(TRUE, FALSE, TRUE)`

To check the data type of created vector: In R `class()` function is used.

Example:
```
a <- c("a", 1, TRUE)
class(a)
```

✓ In above example it will return "character" as an output.

To access the individual element from vector: `a[index]`

Example:
```
a <- c("a", 1, TRUE)
a[2]  #1
```

……

2. What is factor? Explain its function with example.

Factor:

- Factors are the data objects which are used to categorize the data and store it as levels.
- They can store both strings and integers.
- They are useful in the columns which have a limited number of unique values.
- Like "Male, "Female" and True, False etc.
- They are useful in data analysis for statistical modeling.
- Factors are created using the **`factor()`** function by taking a vector as input.

Syntax: `factor(obj, ordered, levels)`

Arguments:

- ✓ `obj`: R object (vector)
- ✓ `ordered`: Logical flag to determine if the levels should be observe in the order given.
- ✓ `levels`: Used to give the levels as user define order

Example:
```
a <- c('red','white','black','red','red')
b <- factor(a)
```

To check the data type of created vector: In R `class()` function is used.

Example: `class(b)`

- ✓ In above example it will return "factor" as an output.

To created ordered factor: Give the ordered argument as True

Example: `b <- factor(a, ordered = TRUE)`

To give the sequence wise levels: Pass the vector in the levels argument

Example: `b <- factor(a, levels = c('red','white','black',NA))`

To check the created object is factor or not: `is.factor(obj)`

Example: `is.factor(b)`

To check the created factor is ordered or not: `is.ordered(obj)`

Example: `is.ordered(b)`

To prints only levels from a factor: `levels(obj)`

Example: `levels(b)`

……

3. What is an array? Explain its function with example.

Array:

- ✓ Arrays are the R data objects which can store data in more than two dimensions.
- ✓ For example - If we create an array of dimension (2, 3, 4) then it creates 4 rectangular matrices each with 2 rows and 3 columns.
- ✓ Arrays can store only data type.
- ✓ An array is created using the **`array()`** function.
- ✓ It takes vectors as input and uses the values in the dim parameter to create an array.

Syntax: `array(data, dim, dimnames)`

Arguments:

- ✓ `data`: Can be a vector

- ✓ `dim`: Used to give dimension (row, col)
- ✓ `dimnames`: Used to give names of the dimensions `list(rows.nm, cols.nm)`

To create an array:

```
a <- array(data = letters, dim = c(2,4), dimnames = list(1:2,1:4))
```

To get or set the row names of array: `row.names(obj)`

Example: `row.names(a)`

To get or set the column names of array: `colnames(obj)`

Example: `colnames(a)`

To get the no. of rows from array: `nrow(obj)`

Example: `nrow(a)`

To get the no. of columns from array: `ncol(obj)`

Example: `ncol(a)`

To get the created object is an array or not: `is.array(obj)`

Example: `is.array(a)`

To extracting the elements from array:

Example:

- ✓ To extract 1st element of 1st row: `a[1,1]`
- ✓ To extract elements of 1st & 2nd columns: `a[,c(1,2)]`
- ✓ To extract elements of 1st & 2nd rows of 1st & 4th columns: `a[c(1,2),c(1,4)]`
- ✓ To extract 1st column 1st element and 2nd column 4th element: `c(a[1,1],a[2,4])`
- ✓ To extract 1st column 4th element and 2nd column 1st element: `c(a[1,4],a[2,1])`

To create multi dimension array:

```
a <- array(data = letters, dim = c(2,2,2))        #row, col, pairs
```

To set or modify the value of given position of array:

Example: `a[1,1,1] <- 'PM'`

To retrieve the dimensions of created array: `dim(obj)`

Example: `dim(a)`

……

4. Explain matrix. Explain its function with example.

Matrix:

- Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout.
- They contain elements of the same atomic types.

- Though we can create a matrix containing only characters or only logical values, they are not of much use.
- We use matrices containing numeric elements to be used in mathematical calculations.
- A Matrix is created using the **`matrix()`** function.

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

Arguments:

- ✓ `data` is the input vector which becomes the data elements of the matrix.
- ✓ `nrow` is the number of rows to be created.
- ✓ `ncol` is the number of columns to be created.
- ✓ `byrow` is a logical clue. If TRUE then the input vector elements are arranged by row.
- ✓ `dimname` is the names assigned to the rows and columns.

To create a matrix:

```
a <- matrix(data = letters, nrow = 2, ncol = 3, byrow = TRUE,
dimnames = list(paste('r', 1:2, sep = ''), paste('c', 1:3, sep = '')) )
```

To get or set the row names of matrix: `row.names(obj)`
Example: `row.names(a)`

To get or set the column names of matrix: `colnames(obj)`
Example: `colnames(a)`

To get the no. of rows from matrix: `nrow(obj)`
Example: `nrow(a)`

To get the no. of columns from matrix: `ncol(obj)`
Example: `ncol(a)`

To get the created object is an matrix or not: `is.matrix(obj)`
Example: `is.martix(a)`

To extracting the elements from matrix:
Example:

- ✓ To extract 1st element of 1st row: `a[1,1]`
- ✓ To extract elements of 1st & 2nd columns: `a[,c(1,2)]`
- ✓ To extract elements of 1st & 2nd rows of 1st & 4th columns: `a[c(1,2),c(1,4)]`
- ✓ To extract 1st column 1st element and 2nd column 4th element: `c(a[1,1],a[2,4])`
- ✓ To extract 1st column 4th element and 2nd column 1st element: `c(a[1,4],a[2,1])`

To retrieve the dimensions of created array: `dim(obj)`
Example: `dim(a)`

……

5. Explain list. Explain its function with example.

List:

- Lists are the R objects which contain elements of different types.
- Like - numbers, strings, vectors and another list inside it.
- A list can also contain a matrix or a function as its elements.
- List is created using **`list()`** function.

Syntax: `list(...)`

Arguments:

- ✓ … no of R objects.

To create a list containing strings, numbers and vectors values:

```
my.list <- list("Pooja", "Hetal", c(11,12,13), 71.23, 9.55)
```

To give the name of list elements: `names(list.obj) <- name.vector`

Example:

```
names(my.list) <- c("String 1", " String 2", "A Numeric vector",
"Number 1", "Number 2")
```

To extract the elements from the list:

Example:

- ✓ To extract 1st element of list: `my.list[1]`
- ✓ To extract 1st element of list: `my.list$`String 1``
- ✓ To extract 3rd element of list: `my.list[3]`
- ✓ To extract 3rd element of list: `my.list$`A Numeric vector``
- ✓ To extract 3rd element's 2nd element of list: `my.list[[3]][[2]]`
- ✓ To extract 3rd element's 2nd element of list: `my.list$`A Numeric vector`[2]`

To manipulate the list element:

Example:

```
my.list[1] <- 'p'
my.list[[3]][[2]] <- 21
```

To convert list element in vector: `unlist(list.obj[index])`

Example: `unlist(my.list[3])`

……

6. Explain data frame. Explain its function with example.

Data frame:

- A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.
- Following are the characteristics of a data frame.
  - ✓ The column names should be non-empty.
  - ✓ The row names should be unique.

- ✓ The data stored in a data frame can be of numeric, factor or character type.
- ✓ Each column should contain same number of data items.

- To create the data frame in R **`data.frame()`** function is used.

Syntax: `data.frame(...)`

Arguments:

- ✓ … no of R objects.

To create the data frame:

Example:

```
emp_id <- c('E01', 'E02', 'E03', 'E04', 'E05', 'E06')
name <- c('A', 'B', 'C', 'D', 'E', 'F')
gender <- c('F', 'F', 'M', 'M', 'F', 'M')
salary <- c(58560, 60850, 45860, 78965, 75620, 85023)
my.data <- data.frame(emp_id, name, gender, salary)
```

Expanding data frame:

1) By adding new column by cbind() function:

Example:

```
skill <- c('.Net', 'SQL', '.Net', 'Management', 'Designing', 'Hacking')
new.data <- cbind(my.data, skill)
```

2) By adding new row rbind() function:

Example:

```
record <- data.frame('E07','G','M','45062','.Net')
names(record) <- names(new.data)
new.data <- rbind(new.data, record)
```

Extract the element & applying filtration to data frame:

Example:

- ✓ To extract the particular column: `new.data$name`
- ✓ To extract the particular value from column: `as.character(new.data$name[1])`
- ✓ To extract the 7th row from data frame: `new.data[7,]`
- ✓ To retrieve the column names: `names(new.data)`
- ✓ To extract the name which contain 'h': `new.data$name[(grep('*h',name))]`
  `new.data[(grep('*h',name)),]`
  `subset(new.data, grepl('*h', name))`
- ✓ To extract the whose gender is male and salary is greater than 70000:
  `subset(new.data, gender=='M' & salary > 70000)`
- ✓ To extract the employee who has '.Net' skills: `new.data[which(skill=='.Net'),]`

……

7. Explain csv file and its function with example.

- In R, we can read data from files stored outside the R environment.

- We can also write data into files which will be stored and accessed by the operating system.
- R can read and write into various file formats like csv, excel, xml etc.
- For reading the file, it should be present in current working directory so that R can read it.
- So, we have to set our own directory and read files from there.

To get the path of working directory: `getwd()`

To set the path of working directory: `setwd("path")`

csv file:

- The csv file is a text file in which the values in the columns are separated by a comma.
- The file extension must be .csv.

1) Writing the CSV file:

- In R the **`write.csv()`** function is used to create the csv file.
- This file gets created in the working directory.

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

Arguments:

- ✓ `x` is the vector or data frame which becomes the data elements of the file.
- ✓ `file` is the output file name.
- ✓ `sep` is the field separator string. Values within each row of x are separated by this string.
- ✓ `append` is logical. Only relevant if file is a character string. If TRUE, the output is appended to the file. If FALSE, any existing file of the name is destroyed.

Example:

```
emp_id <- c('E01', 'E02', 'E03', 'E04', 'E05', 'E06')
name <- c('A', 'B', 'C', 'D', 'E', 'F')
gender <- c('F', 'F', 'M', 'M', 'F', 'M')
salary <- c(58560, 60850, 45860, 78965, 75620, 85023)
my.data <- cbind(emp_id, name, gender, salary)
write.csv(x = my.data, file = "a1.csv", append = FALSE)
```

2) Reading the CSV file:

- In R **`read.csv()`** function to read a CSV file.
- The file should be available in your current working directory.
- It returns a data frame as result.

Syntax: `read.csv(file, skip, header, sep)`

Arguments:

- ✓ `file` is the output file name.
- ✓ `skip` is integer: the number of lines of the data file to skip before beginning to read data.

- ✓ `header` is a logical value indicating whether the file contains the names of the variables as its first line. Header is set to TRUE if and only if the first row contains one fewer field than the number of columns.
- ✓ `sep` is the field separator string. Values within each row of x are separated by this string.

Example:
```
a <- read.csv(file = "a1.csv", skip = 2, header = F)
class(a)    # data.frame
```

.....

8. Explain xlsx file and its function with example.

- Microsoft Excel is the most widely used spreadsheet program which stores data in the .xls or .xlsx format.
- R can read directly from these files using some excel specific packages.
- Few such packages are - XLConnect, xlsx, gdata etc.
- We will be using **xlsx** package.
- R can also write into excel file using this package.

Package:

To install the package: `install.packages("xlsx")`
To check whether the package was installed or not: `any(grepl("xlsx",installed.packages()))`
To load the package: `require("xlsx")`

1) Writing the XLSX file:

- In R the **`write.xlsx()`** function is used to create the csv file.
- This file gets created in the working directory.

Syntax:
```
write.xlsx(x, file, sheetIndex)
```

Arguments:

- ✓ `x` is the vector or data frame which becomes the data elements of the file.
- ✓ `file` is the output file name.
- ✓ `sheetIndex` is integer, indicates the sheet no of file.

Example:
```
emp_id <- c('E01', 'E02', 'E03', 'E04', 'E05', 'E06')
name <- c('A', 'B', 'C', 'D', 'E', 'F')
gender <- c('F', 'F', 'M', 'M', 'F', 'M')
salary <- c(58560, 60850, 45860, 78965, 75620, 85023)
my.data <- cbind(emp_id, name, gender, salary)
write.xlsx(my.data, "data.xlsx", sheetIndex = 1)
```

2) Reading the XLSX file:

- The data.xlsx is read by using the **`read.xlsx()`** function as shown below.
- The result is stored as a data frame in the R environment

Syntax:
```
read.xlsx(file, sheetIndex)
```

Arguments:

- ✓ `file` is the output file name.
- ✓ `sheetIndex` is the

Example:

```
a <- read.xlsx("data.xlsx", sheetIndex = 1)
class(a)   # data.frame
```

……

9. Explain xml file and its function with example.

- 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 xml file in R using the "XML" package.

Package:

To install the package: `install.packages("XML")`
To check whether the package was installed or not: `any(grepl("XML",installed.packages()))`
To load the package: `require("XML")`

Syntax:

To parse the XML structure in R: `xmlParse(file)`
Extract the root node form the xml file: `xmlRoot(x, skip)`
To find number of nodes in the root: `xmlSize(obj)`
To convert XML format to the data frame: `xmlToDataFrame(doc)`

Arguments:

- ✓ `x` is the data frame which becomes the data elements of the file.
- ✓ `file` is the input file name.
- ✓ `doc` is the input file name.
- ✓ `skip` is integer the number of lines of the data file to skip before beginning to read data.
- ✓ `obj` is the data frame which becomes the data elements of the file.

Example:

- ✓ To read the XML file: `result <- xmlParse(file = "input.xml")`
- ✓ To extract the root node of XML file: `rootnode <- xmlRoot(result)`
- ✓ To find the no. of node in the root: `rootsize <- xmlSize(rootnode)`
- ✓ To extract the 1st node of root node: `firstnode <- rootnode[1]`
- ✓ To extract the 1st node 2nd element: `first_sec_ele <- rootnode[[1]][[2]]`

- ✓ To convert XML file structure into data frame: `xmldataframe <- xmlToDataFrame("data.xml")`

……

10. Explain JSON file and its function with example.

- JSON file stores data as text in human-readable format.
- JSON stands for JavaScript Object Notation.
- R can read JSON files using the rjson package.

Package:
```
install.packages("rjson")
install.packages("jsonlite")
install.packages("RJSONIO")
```

Read JSON file:

- The JSON file is read by R using the function `fromJSON()`. It is stored as a list in R.

Syntax: `fromJSON(content, handler = NULL)`

Arguments:

- ✓ `content / file` is either a file name or a character string I handler R object responsible for processing each individual token/element `file` is the input file name.

Example: `result <- fromJSON(file="input.json")`

Convert JSON to data frame:

- We can convert the extracted data above to a R data frame for further analysis using the `as.data.frame()` function.

Syntax: `as.data.frame(obj)`

Arguments:

- ✓ `obj` is the data frame which becomes the data elements of the file.

Example: `json.data.frame <- as.data.frame(result)`

Write JSON file:

- The JSON file is read by R using the function `toJSON()`. It is stored as a list in R.
- toJSON(x, container = isContainer(x, asIs, .level), collapse = "\n", ...)
- I x the R object to be converted to JSON format I container whether to treat the object as a vector/container or a scalar I collapse string used as separator when combining the individual lines of the generated JSON content I ... additional arguments controlling the JSON formatting

Syntax: `toJSON(list)`

Arguments:

- ✓ `list` : It can be a multiple vectors

Example: `toJSON(file="input.json")`

……

11. Explain web data and its function with example.

- Many websites provide data for consumption by its users.
- For example the World Health Organization (WHO) provides reports on health and medical information in the form of CSV, txt and XML files.
- Using R programs, we can programmatically extract specific data from such websites.
- Some packages in R which are used to scrap data from the web are - "RCurl", XML", and "stringr".
- They are used to connect to the URL's, identify required links for the files and download them to the local environment.

Package:
```
install.packages("RCurl")
install.packages("XML")
install.packages("stringr")
install.packages("pylr")
```

- We will use the function **`getHTMLLinks()`** to gather the URLs of the files.
- Then we will use the function **`downlaod.file()`** to save the files to the local system.
- As we will be applying the same code again and again for multiple files, we will create a function to be called multiple times.
- The filenames are passed as parameters in form of a R list object to this function.

To get the file links from url:

Syntax: `getHTMLLinks(doc)`

Arguments:

- ✓ `doc` is the url of the documents or files.

Example:
```
url <- "http://www.geos.ed.ac.uk/~weather/jcmb_ws/"
links <- getHTMLLinks(url)
```

To filter the downloaded file:

Syntax: `str_detect(string, pattern)`

Arguments:

- ✓ string is the input vector (object of link)
- ✓ pattern is used to filter the files of the link

Example:
```
filenames <- links[str_detect(links, "JCMB_2015")]
```
To Store the file names as a list:
```
filenames_list <- as.list(filenames)
```

To download the file from the internet:

Syntax: `download.file(url, destfile)`

Arguments:

- ✓ `url` is the link of the files.
- ✓ `destfile` is the file name to be stored in the destination.

Example:

```
downloadcsv <- function (mainurl,filename)
{
  filedetails <- str_c(mainurl,filename)
  download.file(filedetails,filename)
}
```

To filter the downloaded file:

Syntax: `l_ply(.data, .fun, mainurl)`

Arguments:

- ✓ .data is the list to be processed
- ✓ .fun is function to apply to each piece
- ✓ mainurl is the original URL

Example:

```
l_ply(filenames, downloadcsv,
      mainurl="http://www.g eos.ed.ac.uk/~weather/jcmb_ws/ ")
```

……

12. Explain ms access data base operations.

Package: `install.packages("RODBC")`

To connect with the mysql (to establish a connection with mysql):

Syntax: `odbcDriverConnect(connection)`

Argument:

- ✓ connection is a character string Purpose: to establish connection.

Example:

```
library("RODBC")
channel <- odbcDriverConnect("driver={Microsoft Access Driver;*.mdb,*.accdb};
                             DBQ=d/emp.accd")
```

To fire the query on the ms access:

Syntax: `sqlQuery(channel, query)`

Argument:

- ✓ channel is the object of connection.

- ✓ query is used to fire the query on the mysql.

Example:

```
data <- sqlQuery(channel, paste("select * from employee"))
data <- sqlQuery(channel, paste("select * from employee where id>5 order by
                name"))
```

To fetch the ms access table's data:

Syntax: `sqlFetch(channel, sqtable)`

Argument:

- ✓ channel is the object of connection.
- ✓ sqtable is a character sequence which is used for the table name.

Example:

```
data <- sqlFetch(channel," employee ")
```

……

13. Explain ms access data base operation

- The data is Relational database systems are stored in a normalized format.
- So, to carry out statistical computing we will need very advanced and complex Sql queries.
- But R can connect easily to many relational databases like MySql, Oracle, Sql server etc.
- And fetch records from them as a data frame.
- Once the data is available in the R environment, it becomes a normal R data set and can be manipulated or analyzed using all the powerful packages and functions.
- In this tutorial we will be using MySql as our reference database for connecting to R.

Package:

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

Connecting R to MySql

- Once the package is installed we create a connection object in R to connect to the database.
- It takes the username, password, database name and host name as input.

Syntax:

```
dbConnect(drv, user, password, dname, host)
dbListTables(conn)    # used to list the tables
dbListFields(conn)    # used to list the fields of all table
```

Arguments:

- ✓ `drv`: to inherit DBDriver
- ✓ `user`: user name
- ✓ `password`: login password
- ✓ `dname`: database name
- ✓ `host`: host name

Example:

```
mydb <- dbConnect(MySQL(), user = "root", password = "",
                  dname = "mydb", host = "10.9.2.81")
dblisttbl <- dbListTables(mydb)
dblistfields <- dbListFields(mydb)
```

Queryring the tables

- We can query the database tables in MySql using the function dbSendQuery().
- The query gets executed in MySql and the result set is returned using the R fetch() function.
- Finally it is stored as a data frame in R.

Syntax: `dbSendQuery(conn, statement)`

Arguments:

- ✓ `conn`: connection object
- ✓ `statement`: sql query Purpose

Example:

```
result <- dbSendQuery(mydb,"select * from emp")
```

Query with filter clause

- We can pass any valid select query to get the result.

Syntax: `fetch(res, n)`

Arguments:

- ✓ `fetch`: object of dbSendQuery()
- ✓ `n`: (-1 all) records fetched from table

Example:

```
result = dbSendQuery(mysqlconnection, "select * from actor where
           last_name='TORN'")
data <- fetch(result, n=5)
```

Updating rows in the table

- We can update the rows in a Mysql table by passing the update query to the dbSendQuery() function.

```
dbSendQuery(mydb, "update emp set name='pooja' where id=3")
```

Inserting data into the table

```
dbSendQuery(mydb, "insert into emp(id,name) values(5,'don')")
```

Creating tables in MySql

- We can create tables in the MySql using the function dbWriteTable().
- It overwrites the table if it already exists and takes a data frame as input.

Syntax: `dbWriteTable(conn, sqtable, value)`

Argument:

- ✓ `conn`: connection object
- ✓ `sqtable`: table name
- ✓ `value`: data frame

Example:

```
dbWriteTable(mydb, "dept",dept[,], overwrite = TRUE)
```

Dropping table in MySql

- We can drop the tables in MySql database passing the drop table statement into the dbSendQuery() in the same way we used it for querying data from tables.

```
dbSendQuery(mydb,"drop table if exist")
```

……

Explain pie chart with its function and example.

Pie chart:

- R Programming language has numerous libraries to create charts and graphs.
- A pie-chart is a **representation of values as slices of a circle** with different colors.
- The slices are labeled and the numbers corresponding to each slice is also represented in the chart.
- In R the pie chart is created using the **`pie()`** function which takes positive numbers as a vector input.
- The additional parameters are used to control labels, color, title etc.

Syntax: `pie(x, labels, radius, main, col, clockwise, density)`

Arguments:

- ✓ `x` is a vector containing the numeric values used in the pie chart.
- ✓ `radius` indicates the radius of the circle of the pie chart.(value between -1 and +1).
- ✓ `clockwise` is a logical value indicating if the slices are drawn clockwise or anti clockwise.
- ✓ `labels` is used to give description to the slices
- ✓ `main` indicates the title of the chart.
- ✓ `col` indicates the color palette.
- ✓ `density` is the integer, indicates the filling of the graph.
- ✓ `border` indicates the color of slice's border.
- ✓ `lty` indicates the line type.

Example:

```
data <- c(54, 77, 50, 96, 23)
program <- c("R", ".Net", "Java", "Android", "SQL")
percent<- round(100 * data / sum(data), 1)
pie(x = data, border = "blue", lty = 2, labels = paste(program,
                                        "[", percent, "%]" ))
```

3D Pie chart:

- A pie chart with 3 dimensions can be drawn using additional packages.
- The package **plotrix** has a function called **`pie3D()`** function is used.

Package:

To install the package: `install.packages("plotrix")`

To check whether the package was installed or not: `any(grepl("plotrix",installed.packages()))`

To load the package: `require("plotrix")`

Syntax: `pie3D(x, labels, explode, main, radius, main, col, clockwise)`

Arguments:

- ✓ `explode` is the amount to "explode" the pie in user units

Example:

```
pie3D(x = data, border = "blue", explode = 0.2, labels = paste(program,
                                                "[", percent, "%]"))
```

……

14. Explain bar char with its function and example.

Bar chart:

- A bar chart represents data in rectangular bars with length of the bar proportional to the value of the variable.
- R uses the function **`barplot()`** to create bar charts.
- R can draw both vertical and horizontal bars in the bar chart.
- In bar chart each of the bars can be given different colors.
- If you passed the matrix as an argument then R will generate stacked bar chart.

Syntax: `barplot(H, xlab, ylab, main, names.arg, col)`

Arguments:

- ✓ `H` is a vector or matrix containing numeric values used in bar chart.
- ✓ `names.arg` is a vector of names appearing under each bar.
- ✓ `horiz` is logical, If FALSE, the bars are drawn vertically with the first bar to the left. If TRUE, the bars are drawn horizontally with the first at the bottom.
- ✓ `xlab` is the label for x axis.
- ✓ `ylab` is the label for y axis.
- ✓ `xlim` is limits for the x axis.
- ✓ `ylim` is limits for the y axis.
- ✓ `main` indicates the title of the chart.
- ✓ `col` indicates the color palette.
- ✓ `density` is the integer, indicates the filling of the graph.
- ✓ `border` indicates the color of slice's border.

- ✓ `las` is used to display the names in vertical.

Example:

```
data <- c(54, 77, 50, 96, 23)
program <- c("R", ".Net", "Java", "Android", "SQL")
barplot(data, main = "Programming Languages in 2017",
        xlab = "Languages", ylab = "Percent (%)",
        names.arg = program, col = rainbow(length(data)),
        ylim = c(0,100), density = 15)
```

To generate horizontal bar chart:

```
barplot(data, main = "Programming Languages in 2017",
        ylab = "Languages", xlab = "Percent (%)",
        names.arg = program, col = rainbow(length(data)),
        xlim = c(0,100), density = 15, horiz = T, las=2)
```

To generate stacked bar chart:

```
data <- head(mtcars,4)
attach(data)
mat <- matrix(c(disp,hp), nrow=2, ncol=4, byrow = T)
barplot(mat, main = "Stack Bar Chart",
        names.arg = as.character(row.names(data)),
        ylab = " disp & hp", xlab = "Cars",
        col = rainbow(nrow(mat)), density = 25, las=2)
legend("topright", fill = rainbow(nrow(mat)),
       c("disp","hp"), cex = 0.7, density = 25)
```

---

15. Explain box plot with its function and example.

Box plot:

- Boxplots are a measure of how well distributed is the data in a data set.
- It divides the data set into **three quartiles**.
- This graph represents the **minimum, maximum, median, first quartile and third quartile** in the data set.
- It is also useful in comparing the distribution of data across data sets by drawing boxplots for each of them.
- Boxplots are created in R by using the **`boxplot()`** function.

Syntax: `boxplot(x, data, notch, varwidth, names, main)`

Arguments:

- ✓ `x` is a vector or a formula.
- ✓ `formula` is such as y ~ grp, where y is a numeric vector of data values to be split into groups according to the grouping variable grp.
- ✓ `data` is the data frame.
- ✓ `notch` is a logical value. Set as TRUE to draw a notch.
- ✓ `varwidth` is a logical value. Set as true to draw width of the box proportionate to the sample size.

- ✓ `names` are the group labels which will be printed under each boxplot.
- ✓ `main` indicates the title of the chart.
- ✓ `col` indicates the color palette.
- ✓ `density` is the integer, indicates the filling of the graph.
- ✓ `border` indicates the color of slice's border.
- ✓ `las` is used to display the names in vertical.

Example:

```
data <- mtcars
attach(data)
boxplot(formula = mpg ~ cyl, data = data, xlab = "Number of Cylinders",
        ylab = "Miles Per Gallon", main = "Mileage Data", col = rainbow(3),
        varwidth = T, las = 2, notch = T)
```

……

16. Explain histogram with its function and example.

Histogram:

- A histogram represents the frequencies of values of a variable bucketed into ranges.
- Histogram is similar to bar chart but the difference is it groups the values into continuous ranges.
- Each bar in histogram represents the height of the number of values present in that range.
- R creates histogram using **`hist()`** function.
- This function takes a vector as an input and uses some more parameters to plot histograms.

Syntax: `hist(x, main, xlab, xlim, ylim, breaks, col, border)`

Arguments:

- ✓ `x` is a vector containing numeric values used in histogram.
- ✓ `breaks` is used to mention the width of each bar.
- ✓ `freq` is logical; if TRUE, the histogram graphic is a representation of frequencies, the counts component of the result; if FALSE, probability densities, component density, are plotted (so that the histogram has a total area of one)
- ✓ `main` indicates title of the chart.
- ✓ `col` is used to set color of the bars.
- ✓ `border` is used to set border color of each bar.
- ✓ `xlab` is used to give description of x-axis.
- ✓ `xlim` is used to specify the range of values on the x-axis.
- ✓ `ylim` is used to specify the range of values on the y-axis.
- ✓ `density` is the integer, indicates the filling of the graph.
- ✓ `border` indicates the color of slice's border.
- ✓ `plot` is logical. If TRUE (default), a histogram is plotted, otherwise a list of breaks and counts is returned.

Example:

```
data <- c(54, 77, 50, 96, 23)
hist(x = data, xlab = "Data", xlim = c(0,100), freq = F,
```

```
        col = rainbow(length(data)), border = "blue", labels = T)
    # breaks = c(54, 77, 50, 96, 23)
```

To give the labels to bars:

```
    hist(x = data, xlab = "Data", xlim = c(0,100), freq = F,
        col = rainbow(length(data)), border = "blue",
        labels = c('A','B','C','D'))
```

To get the list of breaks, count and density:

```
    a <- hist(x = data, xlab = "Data", xlim = c(0,100), plot = F,
              freq = F, col = rainbow(length(data)), border = "blue")
    a$density
```

……

17. Explain line chart with its function and example.

Line chart:

- A line chart is a graph that connects a series of points by drawing line segments between them.
- These points are ordered in one of their coordinate (usually the x-coordinate) value.
- Line charts are usually used in identifying the trends in data.
- The **`plot()`** function in R is used to create the line graph.

Syntax: `plot(x, type, col, xlab, ylab)`

Arguments:

- ✓ `x` is a vector containing numeric values used in line chart.
- ✓ `type` is used for what type of plot should be drawn. Possible types are
  - o "`p`" for points,
  - o "`l`" for lines,
  - o "`b`" for both,
  - o "`c`" for the lines part alone of "b",
  - o "`o`" for both 'overplotted',
  - o "`h`" for 'histogram' like (or 'high-density') vertical lines,
  - o "`s`" for stair steps,
  - o "`S`" for other steps, see 'Details' below,
  - o "`n`" for no plotting.
- ✓ `main` indicates title of the chart.
- ✓ `col` is used to set color of the bars.
- ✓ `xlab` is used to give description of x-axis.
- ✓ `ylab` is used to give description of y-axis.
- ✓ `xlim` is used to specify the range of values on the x-axis.
- ✓ `ylim` is used to specify the range of values on the y-axis.
- ✓ `lty` is the integer, indicates the filling of the graph.

Example:

```
a <- c(54, 77, 50, 96, 23)
plot(x = a, type = 'o', names = c(2:5), ylim = c(0,100),
```

```
        main = "Line Chart", col="blue")
```

To draw a multiple line on a graph the `lines()` function will be used:

Syntax: `lines(x, type, ...)`

Example:

```
b <- c(34, 47, 60, 26, 43)
lines(x = b, type = 'o', col="red")
legend("topright", fill = c("blue","red"),
       c("A","B"), cex = 0.7)
```

……

18. Explain scatter plot with its function and example.

Scatter plot:

- Scatterplots show many points plotted in the Cartesian plane.
- Each point represents the values of two variables.
- One variable horizontal axis and another in the vertical axis.
- The simple scatterplot is created using the **`plot()`** function.

Syntax: `plot(x, y, main, xlab, ylab, xlim, ylim, axes)`

Arguments:

- ✓ `x` is a vector containing numeric values used in line chart.
- ✓ `axes` indicates whether both axes should be drawn on the plot.
- ✓ `pch` is integer, plotting symbol (patch pattern of the plotting).
- ✓ `main` indicates title of the chart.
- ✓ `col` is used to set color of the bars.
- ✓ `xlab` is used to give description of x-axis.
- ✓ `ylab` is used to give description of y-axis.
- ✓ `xlim` is used to specify the range of values on the x-axis.
- ✓ `ylim` is used to specify the range of values on the y-axis.

Example:

```
a <- c(54, 77, 50, 96, 23)
b <- c(34, 47, 60, 26, 43)
plot(x = a, y = b, xlab = "Data of A", ylab = "Data of B",
     xlim = c(0,100), ylim = c(0,100),
     col= rainbow(length(a)), pch=16,
     main = "Scatter Plot")
```

Scatterplot Matrices:

- When we have more than two variables and we want to find the correlation between one variable versus the remaining ones scatterplot matrix.
- We use **`pairs()`** function to create matrices of scatterplots.

Syntax: `pairs(formula, data)`

Arguments:

- ✓ `formula` is represents the series of variables used in pairs.
- ✓ `data` is represents a data frame in pairs function.

Example:

```
pairs(~a+b, data = data.frame(a,b),
      main="Scatterplot Matrix", pch=16,
      col= rainbow(length(a)))
```

……

19. Explain function and its types and function calling types in R.

Function: 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.
- In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.
- The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.
- In R function is created by using the keyword function.

Syntax/structure of function:

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

Arguments:

- ✓ `arg` is argument which was passed by user or default set by developer.
- ✓ … is extra argument passed by user.

Function types:

1. 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. You can refer most widely used R functions.

Example:

```
print(seq(32,44))       # Create a sequence of numbers from 32 to 44.
print(mean(25:82))      # Find mean of numbers from 25 to 82.
print(sum(41:68))       # Find sum of numbers form 41 to 68.
```

2. 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.

Example: **Create a function to print squares of numbers in sequence.**

```
my.function <- function(a) {
  for(i in 1:a) {
    b <- i^2
    print(b)
  }
}
my.function(5)   #calling a function
```

Calling function:

1. Function without argument

Example: **Create a function to print squares of numbers 1 to 5 in sequence.**

```
my.function <- function() {
  for(i in 1:5) {
    print(i^2)
  }
}
my.function()    #calling a function
```

2. Function with argument

Example: **Create a function to get the user name and print it on the console.**

```
my.function <- function(user.nm) {
    paste("Hi, ",user.nm)
}
my.function("P.B.Mandaliya")     #calling a function
```

3. Function with default argument

Example: **Create a function to get the two integer values from user and display the sum of it.**

```
my.function <- function(a = 5, b = 6) {
  print(sum(a+b))
}
my.function()    #calling a function with default argument   [11]
my.function(4,2)      #calling a function with argument     [6]
```

4. Calling a Function with Argument Values (by position and by name)

Example:

```
my.function <- function(fnm, lnm) {
  cat("First name: ", fnm , "\nLast name:", lnm)
}
my.function(lnm = "Mandaliya", fnm = "Pooja")    #calling a function
```

5. Function with extra argument

Example:

```
my.function <- function(a ,...) {
  cat("U entered : ", a )
}
my.function(5,8,5,9,5)      #calling a function
```

EXTRA: Function to create binary operator

Example:

```
`%pm%` <- function(p,m)
{
  paste("Answer : ",p*2+m*2)
}
2 %pm% 3   # calling operator [Answer :10]
```

……

20. Explain conditional statements with example.

- The conditional statements are also known as Decision making structures.
- It requires the programmer to specify one or more conditions to be evaluated or tested by the program.
- Along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

R provides the following types of decision making statements:

1. if statement

- An if statement consists of a Boolean expression followed by one or more statements.

Syntax: `if(cond) expr`

Example:

```
a <- function(val)
{
  if(nchar(val)>=5)
    print(val)
}
a('Pooja')
```

2. if...else statement

- An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.

Syntax: `if(cond) expr else expr`

Example:

```
a <- function(val)
{
```

```
  if(nchar(val)>=5)
    print(val)
  else
    print("Invalid...!")
}
a('abcd')
```

3. if...else if…else statement

- Nested if conditions. Executes only one condition.

Syntax: `if(cond) expr else if(cond) expr else expr`

Example:

```
a <- function(val)
{
  if(val>0)
    print("Positive")
  else if(val<0)
    print("Negative")
  else
    print("Zero")
}
a(0)        #Zero
a(1)        #Positive
a(-1)       #Negative
```

4. switch statement

- A switch statement allows a variable to be tested for equality against a list of values.

Syntax: `switch(EXPR, ...)`

Example:

```
a <- function(val)
{
  switch(val,
         "A"=,"a"="Char A",
         "B"=,"b"="Char B",
         "C"=,"c"="Char C",
         "Unknown")
}
a('A')      # Char A
a('a')      # Char A
a('P')      # Unknown
```

5. ifelse

- ifelse it is works like ternary operator

Syntax: ifelse(test, yes, no)

Example:

```
ifelse(test, yes, no)
a <- function(val)
{
  ifelse(val==1,yes = "Equal", no = "Not equal")
}
a(1)  #Equal
a(11) #Not equal
```

……

21. Explain looping statements with example.

- There may be a situation when you need to execute a block of code several number of times.
- In general, statements are executed sequentially.
- The first statement in a function is executed first, followed by the second, and so on.
- Programming languages provide various control structures that allow for more complicated execution paths.
- A loop statement allows us to execute a statement or group of statements multiple times.
- R programming language provides the following kinds of loop to handle looping requirements.

Loop types:

1. for loop

- A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.
- R's for loops are particularly flexible in that they are not limited to integers, or even numbers in the input. We can pass character vectors, logical vectors, lists or expressions.

Syntax:

```
for (value in vector) {
  statements
}
```

Example:

```
v <- LETTERS[1:4]
for ( i in v) {
  print(i)
}
```

2. while loop

- The While loop executes the same code again and again until a stop condition is met.
- The while loop is that the loop might not ever run.
- When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Syntax:

```
while (test_expression) {
   statement
}
```

Example:

```
v <- c("Hello")
cnt <- 2
while (cnt < 7){
  cat(v, cnt, "\n")
  cnt = cnt + 1
}
```

3. repeat loop

- The Repeat loop executes the same code again and again until a stop condition is met.
- Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

Syntax:

```
repeat {
   commands
   if(condition){
        break
   }
}
```

Example:

```
cnt <- 2
repeat{
  cat(cnt, " ")
  cnt <- cnt+1
  if(cnt > 5){
    break
  }
}
```

Loop control statements:

- Loop control statements change execution from its normal sequence.
- When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
- R supports the following control statements.

1. break statement

- The break statement in R programming language has the following usage.
- When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.

Syntax: `break`

Example:

```
v <- LETTERS[1:4]
for ( i in v) {
  if(i=='C')
    break
  print(i)
}
```

2. next statement

- The next statement in R programming language is useful when we want to skip the current iteration of a loop without terminating it.
- On encountering next, the R parser skips further evaluation and starts next iteration of the loop.

Syntax: `next`

Example:

```
v <- LETTERS[1:4]
for ( i in v) {
  if(i=='C')
    next
  print(i)
}
```

……

22. Explain normal distribution with example.

- In a random collection of data from independent sources, it is generally observed that the distribution of data is normal.
- Which means, on plotting a graph with the value of the variable in the horizontal axis and the count of the values in the vertical axis we get a bell shape curve.
- The center of the curve represents the mean of the data set.
- In the graph, fifty percent of values lie to the left of the mean and the other fifty percent lie to the right of the graph.
- This is referred as normal distribution in statistics.
- R has four in built functions to generate normal distribution

1. dnorm():

- This function gives height of the probability distribution at each point for a given mean and standard deviation.

Syntax: `dnorm(x, mean, sd)`

Arguments:

- ✓ `x` is a vector of quantiles.
- ✓ `mean` is the mean value of the sample data. It's default value is zero.