# R PROGRAMMING

**CLASS :** III B.Sc IT **SUBJECT CODE:** 20UIT6DE2B **STAFF : B DIANA**

## UNIT I

### TOPIC 1 : HOW TO RUN R

Running the R Program

Once R is installed you can run it in a variety of ways:

In Windows the program works like any other—you may have a desktop shortcut,a quick launch icon, or simply get to it via the Start button and the regular program list.

On a Macintosh the program is located in the Applications folder and you can drag this to the dock to create a launcher or create an alias in the usual manner.

On Linux the program is launched via the Terminal program, which is located in the Accessories section of the Applications menu.

Once the R program starts up you are presented with the main input window and a short introductory message that appears a little different on each OS:

In Windows a few menus are available at the top.

On the Macintosh OS X, the welcome message comes.

In Linux systems there are no icons and the menu items you see relate to the Terminal program rather than R itself.

## TOPIC 2:  INTRODUCTION TO 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.

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.

### 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
}
```

### Function Components

The different parts of a function are −

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

R has many **in-built** functions which can be directly called in the program without defining them first. We can also create and use our own functions referred as **user defined** functions.

## 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.

EG: 1

```
# 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 frm 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:

```
# Create a function to print squares of numbers in sequence.
new.function <- function(a) {
   for(i in 1:a) {
      b <- i^2
      print(b)
   }
}

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

## TOPIC 3: SOME IMPORTANT DATA STRUCTURES IN R

The most essential data structures used in R include:

- **Vectors**
- **Lists**
- **Dataframes**
- **Matrices**
- **Arrays**
- **Factors**

### Vectors

A vector is an ordered collection of basic data types of a given length. The only key thing here is all the elements of a vector must be of the identical data type e.g homogeneous data structures. Vectors are one-dimensional data structures.

```
# R program to illustrate Vector



# Vectors(ordered collection of same data type)

X = c(1, 3, 5, 7, 8)
```

```
# Printing those elements in console

print(X)
```

**Output:**

```
[1] 1 3 5 7 8
```

**Lists**

A list is a generic object consisting of an ordered collection of objects. Lists are heterogeneous data structures. These are also one-dimensional data structures. A list can be a list of vectors, list of matrices, a list of characters and a list of functions and so on.

**Example:**

```
# R program to illustrate a List



# The first attributes is a numeric vector

# containing the employee IDs which is
# created using the 'c' command here

empId = c(1, 2, 3, 4)



# The second attribute is the employee name

# which is created using this line of code here

# which is the character vector

empName = c("Debi", "Sandeep", "Subham", "Shiba")



# The third attribute is the number of employees

# which is a single numeric variable.

numberOfEmp = 4
```

```
# We can combine all these three different

# data types into a list

# containing the details of employees

# which can be done using a list command

empList = list(empId, empName, numberOfEmp)



print(empList)
```

**Output:**

```
[[1]]

[1] 1 2 3 4


[[2]]

[1] "Debi"    "Sandeep" "Subham"  "Shiba"


[[3]]

[1] 4
```

## Dataframes

Dataframes are generic data objects of R which are used to store the tabular data. Dataframes are the foremost popular data objects in R programming because we are comfortable in seeing the data within the tabular form. They are two-dimensional, heterogeneous data structures. These are lists of vectors of equal lengths.

Data frames have the following constraints placed upon them:

- A data-frame must have column names and every row should have a unique name.
- Each column must have the identical number of items.
- Each item in a single column must be of the same data type.
- Different columns may have different data types.

To create a data frame we use the data.frame() function.

**Example:**

```
# R program to illustrate dataframe



# A vector which is a character vector

Name = c("Amiya", "Raj", "Asish")



# A vector which is a character vector
```

```
Language = c("R", "Python", "Java")



# A vector which is a numeric vector

Age = c(22, 25, 45)



# To create dataframe use data.frame command

# and then pass each of the vectors

# we have created as arguments

# to the function data.frame()

df = data.frame(Name, Language, Age)



print(df)
```

**Output:**

```
  Name Language Age
1 Amiya        R  22
2   Raj   Python  25
3 Asish     Java  45
```

**Matrices**

A matrix is a rectangular arrangement of numbers in rows and columns. In a matrix, as we know rows are the ones that run horizontally and columns are the ones that run vertically. Matrices are two-dimensional, homogeneous data structures.
Now, let’s see how to create a matrix in R. To create a matrix in R you need to use the function called matrix. The arguments to this matrix() are the set of elements in the vector. You have to pass how many numbers of rows and how many numbers of columns you want to have in your matrix and this is the important point you have to remember that by default, matrices are in column-wise order.

**Example:**

```
# R program to illustrate a matrix



A = matrix(

    # Taking sequence of elements
    c(1, 2, 3, 4, 5, 6, 7, 8, 9),



    # No of rows and columns

    nrow = 3, ncol = 3,



    # By default matrices are

    # in column-wise order

    # So this parameter decides

    # how to arrange the matrix

    byrow = TRUE

)
```

```
print(A)
```

**Output:**

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

**Arrays**

Arrays are the R data objects which store the data in more than two dimensions. Arrays are n-dimensional data structures. For example, if we create an array of dimensions (2, 3, 3) then it creates 3 rectangular matrices each with 2 rows and 3 columns. They are homogeneous data structures.

Now, let’s see how to create arrays in R. To create an array in R you need to use the function called array(). The arguments to this array() are the set of elements in vectors and you have to pass a vector containing the dimensions of the array.

**Example:**

```
# R program to illustrate an array
```

```
A = array(

    # Taking sequence of elements

    c(1, 2, 3, 4, 5, 6, 7, 8),



    # Creating two rectangular matrices

    # each with two rows and two columns

    dim = c(2, 2, 2)

)



print(A)
```

**Output:**

```
, , 1


     [,1] [,2]
```

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

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

**Factors**

Factors are the data objects which are used to categorize the data and store it as levels. They are useful for storing categorical data. They can store both strings and integers. They are useful to categorize unique values in columns like “TRUE” or “FALSE”, or “MALE” or “FEMALE”, etc.. They are useful in data analysis for statistical modeling.

Now, let’s see how to create factors in R. To create a factor in R you need to use the function called factor(). The argument to this factor() is the vector.

**Example:**

```
# R program to illustrate factors
# Creating factor using factor()

fac = factor(c("Male", "Female", "Male",

               "Male", "Female", "Male", "Female"))



print(fac)
```

**Output:**

```
[1] Male   Female Male   Male   Female Male   Female

Levels: Female Male
```

---

## TOPIC 4:  COMMON VECTOR OPERATIONS

What is R Vector?

*A vector is a sequence of elements that share the same data type.* These elements are known as components of a vector.

R vector comes in two parts: **Atomic vectors** and **Lists.** They have three common properties:

- *Type function* – What it is?
- *Length function* – How many elements it contains.
- *Attribute function* – Extra arbitrary metadata.

These data structures share one difference, that is, they differ in the type of their elements:  All elements of an atomic vector must be of the same type, whereas the elements of a list can have different types.

Vectors are the most basic data types in R. Even a single object created is also stored in the form of a vector. Vectors are nothing but arrays as defined in other languages. Vectors contain a sequence of homogeneous types of data. If mixed values are given then it auto converts the data according to the precedence. There are various operations that can be performed on vectors in R.

6 Vector Operations:

I)  Creating a vector

II)  Modifying a vector

III)  Accessing a vector

IV)  Arithmetic operations on a vector

V)  Deleting a vector

VI)  Sorting a vector

***Creating a vector***
Vectors can be created in many ways as shown in the following example. The most usual is the use of ‘c’ function to combine different elements together.

```
X <- c(1, 4, 5, 2, 6, 7)
```

```
print('using c function')

print(X)



Y <- seq(1, 10, length.out = 5)

print('using seq() function')

print(Y)



# using ':' operator to create

# a vector of continuous values.

Z <- 5:10

print('using colon')

print(Y)
```

**Output:**

***Accessing vector elements***

Vector elements can be accessed in many ways. The most basic is using the '[]', subscript operator. Following are the ways of accessing Vector elements:

```
# Accessing elements using the position number.

X <- c(2, 5, 8, 1, 2)

print('using Subscript operator')

print(X[2])



# Accessing specific values by passing

# a vector inside another vector.

Y <- c(4, 5, 2, 1, 7)
```

```
print('using c function')

print(Y[c(4, 1)])



# Logical indexing

Z <- c(5, 2, 1, 4, 4, 3)

print('Logical indexing')

print(Z[Z>3])
```

**Output:**

```
using Subscript operator 5
using c function 1 4
Logical indexing 5 4 4
```

***Modifying a vector***

Vectors can be modified using different indexing variations which are mentioned in the below code:

```
# Creating a vector

X <- c(2, 5, 1, 7, 8, 2)



# modify a specific element

X[3] <- 11

print('Using subscript operator')

print(X)



# Modify using different logics.

X[X>9] <- 0

print('Logical indexing')

print(X)
```

```
# Modify by specifying the position or elements.

X <- X[c(5, 2, 1)]

print('using c function')

print(X)
```

**Output:**

```
Using subscript operator 2  5 11  7  8  2
Logical indexing 2 5 0 7 8 2
using c function 8 5 2
```

***Deleting a vector***

Vectors can be deleted by reassigning them as NULL. To delete a vector we use the NULL operator.

```
# Creating a vector
```

```
X <- c(5, 2, 1, 6)



# Deleting a vector

X <- NULL

print('Deleted vector')

print(X)
```

```
Deleted vector NULL
```

***Arithmetic operations***

We can perform arithmetic operations between 2 vectors. These operations are performed element-wise and hence the length of both the vectors should be the same.

```


# Creating Vectors

X <- c(5, 2, 5, 1, 51, 2)
```

```
Y <- c(7, 9, 1, 5, 2, 1)


# Addition

Z <- X + Y

print('Addition')

print(Z)


# Subtraction

S <- X - Y

print('Subtraction')

print(S)
```

```
# Multiplication

M <- X * Y

print('Multiplication')

print(M)



# Division

D <- X / Y

print('Division')

print(D)
```

**Output:**

```
Addition 12 11  6  6 53  3
Subtraction -2 -7  4 -4 49  1
Multiplication 35  18   5   5 102   2
Division 0.7142857  0.2222222  5.0000000  0.2000000 25.5000000  2.0000000
```

***Sorting of Vectors***
For sorting we use the **sort()** function which sorts the vector in ascending order by default.

```
# Creating a Vector

X <- c(5, 2, 5, 1, 51, 2)



# Sort in ascending order

A <- sort(X)

print('sorting done in ascending order')

print(A)



# sort in descending order.

B <- sort(X, decreasing = TRUE)
```

```
print('sorting done in descending order')

print(B)
```

**Output:**

```
sorting done in ascending order 1  2  2  5  5 51
sorting done in descending order 51  5  5  2  2  1
```

---

## TOPIC 5:  NA AND NULL VALUES

### NA and NULL Values

Readers with a background in other scripting languages may be aware of “no such animal” values, such as None in Python and undefined in Perl. R actually has two such values: NA and NULL.

In statistical data sets, we often encounter missing data, which we represent in R with the value NA. NULL, on the other hand, represents that the value in question simply doesn’t exist, rather than being existent but unknown. Let’s see how this comes into play in concrete terms.

### Using NA

In many of R’s statistical functions, we can instruct the function to skip over any missing values, or NAs. Here is an example:

```
> x <- c(88,NA,12,168,13) > x [1] 88 NA 12 168 13 > mean(x) [1] NA > mean(x,na.rm=T) [1] 70.25 > x <-
c(88,NULL,12,168,13) > mean(x) ...
```

---

## TOPIC 6: FILTERING

The **filter()** method in R is used to subset a data frame based on a provided condition. If a row satisfies the condition, it must produce TRUE. Otherwise, non-satisfying rows will return NA values. Hence, the row will be dropped.

Here listed, in the following table, are some functions and operators, used for filtering data that can be used as a breaking condition:

| **Condition** | **Description** |
|---|---|
| [\`==\`] | Checks equality of the column values |
| [\`>\`] | Checks for greater column values |
| [\`>=\`] | Checks for greater than or equal |
| [\`&\`] | Logical AND operation |
| [\`\|\`] | Logical OR operation |
| [\`!\`] | Logical NOR operation |
| [xor()] | Logical XOR operation |
| [is.na()] | Either value is NA or not |
| [between()] | Checks whether the numerical value(Column value) lies between the specified range or not. |
| [near()] | Compares numerical vector to nearest values. |

```
library(dplyr, warn.conflicts = FALSE)
# Creat a DataFrame
df= data.frame(x=c(2,3,4,5,7,4),
               y=c(1.1,2.1,4.5,45.1,3.2,66),
               z=c(TRUE,FALSE,TRUE,TRUE,FALSE,FALSE))
# condition to filter
filter(df, x < 5 & z==TRUE)
```

EG 2

```
library(dplyr, warn.conflicts = FALSE)
# Create a DataFrame
df=data.frame(x= c(1,2,3,4,5),
             y= c(2.1,4.5,8.2,10.4,50),
             z= c("Aries","Taurus", "Cancer", "Leo","Libra"))
# invoking filter()
df %>% filter(z %in% c("Cancer", "Taurus"))
```

---

## TOPIC 7:  MATRICES AND ARRAYS IN R

### Arrays in R

Arrays are data storage objects in R containing more than or equal to 1 dimension. Arrays can contain only a single data type. The **array()** function is an in-built function which takes input as a vector and arranges them according to **dim** argument. Array is an iterable object, where the array elements are indexed, accessed and modified individually. Operations on array can be performed with similar structures and dimensions. Uni-dimensional arrays are called vectors in R. Two-dimensional arrays are called matrices.

```
# R program to illustrate an array



# creating a vector

vector1 <- c("A", "B", "C")

# declaring a character array

uni_array <- array(vector1)

print("Uni-Dimensional Array")

print(uni_array)



# creating another vector

vector <- c(1:12)

# declaring 2 numeric multi-dimensional
# array with size 2x3

multi_array <- array(vector, dim = c(2, 3, 2))

print("Multi-Dimensional Array")

print(multi_array)
```

**Output:**

```
[1] "Uni-Dimensional Array"
[1] "A" "B" "C"
[1] "Multi-Dimensional Array"
, , 1


     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6


, , 2
```

```
     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

## Matrices in R

Matrix in R is a table-like structure consisting of elements arranged in a fixed number of rows and columns. All the elements belong to a single data type. R contains an in-built function **matrix()** to create a matrix. Elements of a matrix can be accessed by providing indexes of rows and columns. The arithmetic operation, addition, subtraction, and multiplication can be performed on matrices with the same dimensions. Matrices can be easily converted to data frames CSVs.

```
# R program to illustrate a matrix



A = matrix(

    # Taking sequence of elements

    c(1, 2, 3, 4, 5, 6, 7, 8, 9),
    # No of rows and columns

    nrow = 3, ncol = 3,



    # By default matrices are

    # in column-wise order

    # So this parameter decides

    # how to arrange the matrix

    byrow = TRUE

)



print(A)
```

**Output:**

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

***Arrays vs Matrices***

| **Arrays** | **Matrices** |
|---|---|
| Arrays can contain greater than or equal to 1 dimensions. | Matrices contains 2 dimensions in a table like structure. |
| Array is a homogeneous data structure. | Matrix is also a homogeneous data structure. |
| It is a singular vector arranged into the specified dimensions. | It comprises of multiple equal length vectors stacked together in a table. |
| **array()** function can be used to create matrix by specifying the third dimension to be 1. | **matrix()** function however can be used to create at most 2-dimensional array. |

| **Arrays** | **Matrices** |
|---|---|
| Arrays are superset of matrices. | Matrices are a subset, special case of array where dimensions is two. |
| Limited set of collection-based operations. | Wide range of collection operations possible. |
| Mostly, intended for storage of data. | Mostly, matrices are intended for data transformation. |

---

TOPIC 8:  GENERAL MATRIX OPERATIONS

Matrices in R are a bunch of values, either real or complex numbers, arranged in a group of fixed number of rows and columns. Matrices are used to depict the data in a structured and well-organized format. It is necessary to enclose the elements of a matrix in parentheses or brackets. A matrix with 9 elements is shown

below. This Matrix [M] has 3 rows and 3 columns. Each element of matrix [M] can be referred to by its row and column number. For example, $a_{23} = 6$ **Order of a Matrix :** The order of a matrix is defined in terms of its number of rows and columns. Order of a matrix = No. of rows × No. of columns Therefore Matrix [M] is a matrix of order 3 × 3.

**Operations on Matrices**

There are five basic operations (Division, Multiplication, Addition, Subtraction and Transpose)

Eg: Write your practical programs .

---

## TOPIC 9: APPLYING FUNCTIONS TO MATRIX ROWS AND COLUMNS

The **apply()** function can be applied to each row in a matrix in R.

This function uses the following basic syntax:

**apply(X, MARGIN, FUN)**

where:

- **X:** Name of the matrix or data frame.
- **MARGIN:** Dimension to perform operation across. Use 1 for row, 2 for column.
- **FUN:** The function to apply.

```
#create matrix
mat <- matrix(1:15, nrow=3)

#view matrix
mat

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
#find mean of each row
```

```
apply(mat, 1, mean)

[1] 7 8 9

#find sum of each row
apply(mat, 1, sum)

[1] 35 40 45

#find standard deviation of each row
apply(mat, 1, sd)

[1] 4.743416 4.743416 4.743416

#multiply the value in each row by 2 (using t() to transpose the results)
t(apply(mat, 1, function(x) x * 2))

     [,1] [,2] [,3] [,4] [,5]
[1,]    2    8   14   20   26
[2,]    4   10   16   22   28
[3,]    6   12   18   24   30

#normalize every row to 1 (using t() to transpose the results)
t(apply(mat, 1, function(x) x / sum(x) ))

           [,1]      [,2] [,3]      [,4]      [,5]
[1,] 0.02857143 0.1142857  0.2 0.2857143 0.3714286
[2,] 0.05000000 0.1250000  0.2 0.2750000 0.3500000
[3,] 0.06666667 0.1333333  0.2 0.2666667 0.3333333
```

## TOPIC 10 : ADDING AND DELETING ROWS AND COLUMNS OF A MATRIX

Technically, matrices are of fixed length and dimensions, so we cannot add or delete rows or columns. However, matrices can be *reassigned*, and thus we can achieve the same effect as if we had directly done additions or deletions.

**Changing the Size of a Matrix**

Recall how we reassign vectors to change their size:

```
> x

[1] 12  5 13 16  8

> x <- c(x,20)  # append 20

> x

[1] 12  5 13 16  8 20

> x <- c(x[1:3],20,x[4:6])  # insert 20

> x

[1] 12  5 13 20 16  8 20
```

```
> x <- x[-2:-4]  # delete elements 2 through 4

> x

[1] 12 16  8 20
```

---

## TOPIC 11:  NAMING MATRIX ROWS AND COLUMNS

### **Changing the row and column names**

The matrix baskets.team already has some row names. It would be better if the names of the rows would just read “Granny” and “Geraldine”. You can easily change these row names like this:

```
> rownames(baskets.team) <- c("Granny", "Geraldine")
```

You can look at the matrix to check if this did what it’s supposed to do, or you can take a look at the row names itself like this:

```
> rownames(baskets.team)
[1] "Granny"  "Geraldine"
```

The colnames() function works exactly the same. You can, for example, add the number of the game as a column name using the following code:

```
> colnames(baskets.team) <- c("1st", "2nd", "3th", "4th", "5th", "6th")
```

This gives you the following matrix:

```
> baskets.team
    1st 2nd 3th 4th 5th 6th
Granny   12  4  5  6  9  3
Geraldine  5  4  2  4 12  9
```

---

## TOPIC 12: HIGHER DIMENSIONAL ARRAYS

Arrays are the R data objects which can store data in more than two dimensions. For example: If we create an array of dimensions (2, 3, 4) then it creates 4 rectangular matrices each with 2 rows and 3 columns. These types of arrays are called Multidimensional Arrays. Arrays can store only data types.

***Creating a Multidimensional Array***
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. A multidimensional array can be created by defining the value of '**dim**' argument as the number of dimensions that are required.
**Syntax:**
MArray = array(c(vec1, vec2), dim)

```
# Create two vectors of different lengths.
```

```
vector1 <- c(5, 9, 3)

vector2 <- c(10, 11, 12, 13, 14, 15)



# Take these vectors as input to the array.

result <- array(c(vector1, vector2), dim = c(3, 3, 2))

print(result)
```

**Output:**

```
, , 1


     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15


, , 2
```

```
      [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15
```

# UNIT III

## TOPIC 1: CONTROL STATEMENTS

**Control statements** are expressions used to control the execution and flow of the program based on the conditions provided in the statements. These structures are used to make a decision after assessing the variable. In this article, we'll discuss all the control statements with the examples.
In R programming, there are 8 types of control statements as follows:

- if condition
- if-else condition
- for loop
- nested loops
- while loop
- repeat and break statement
- return statement
- next statement

## *if condition*

This control structure checks the expression provided in parenthesis is true or not. If true, the execution of the statements in braces {} continues.

```
x <- 100



if(x > 10){

print(paste(x, "is greater than 10"))

}
```

**Output:**
[1] "100 is greater than 10"

***if-else condition***

It is similar to **if** condition but when the test expression in if condition fails, then statements in **else** condition are executed.

```
x <- 5



# Check value is less than or greater than 10

if(x > 10){

  print(paste(x, "is greater than 10"))

}else{

  print(paste(x, "is less than 10"))

}
```

**Output:**
[1] "5 is less than 10"

***for loop***

It is a type of loop or sequence of statements executed repeatedly until exit condition is reached.

```
x <- letters[4:10]



for(i in x){

  print(i)

}
```

**Output:**

```
[1] "d"

[1] "e"

[1] "f"

[1] "g"

[1] "h"

[1] "i"

[1] "j"
```

## ***Nested loops***

Nested loops are similar to simple loops. Nested means loops inside loop. Moreover, nested loops are used to manipulate the matrix.

**Example:**

```
# Defining matrix

m <- matrix(2:15, 2)



for (r in seq(nrow(m))) {

  for (c in seq(ncol(m))) {

    print(m[r, c])

  }

}
```

**Output:**

```
[1] 2

[1] 4
```

```
[1] 6
[1] 8
[1] 10
[1] 12
[1] 14
[1] 3
[1] 5
[1] 7
[1] 9
[1] 11
[1] 13
[1] 15
```

## ***while loop***

while loop is another kind of loop iterated until a condition is satisfied. The testing expression is checked first before executing the body of loop.

```
x = 1
```

```
# Print 1 to 5

while(x <= 5){

  print(x)

  x = x + 1

}
```

**Output:**

```
[1] 1

[1] 2

[1] 3

[1] 4

[1] 5
```

***repeat loop and break statement***

**repeat** is a loop which can be iterated many number of times but there is no exit condition to come out from the loop. So, break statement is used to exit from the loop. **break** statement can be used in any type of loop to exit from the loop.

```
x = 1


# Print 1 to 5

repeat{

  print(x)

  x = x + 1

  if(x > 5){

    break

   }

}
```

**Output:**

```
[1] 1

[1] 2
```

```
[1] 3
[1] 4
[1] 5
```

## ***return statement***

**return** statement is used to return the result of an executed function and returns control to the calling function.

```
# Checks value is either positive, negative or zero

func <- function(x){

  if(x > 0){

    return("Positive")

  }else if(x < 0){

    return("Negative")

  }else{

    return("Zero")

  }
}



func(1)

func(0)

func(-1)
```

**Output:**

```
[1] "Positive"

[1] "Zero"

[1] "Negative"
```

## ***next statement***

**next** statement is used to skip the current iteration without executing the further statements and continues the next iteration cycle without terminating the loop.

**Example:**

```
# Defining vector

x <- 1:10
```

```
# Print even numbers

for(i in x){

  if(i%%2 != 0){

    next #Jumps to next loop

  }

  print(i)

}
```

**Output:**

```
[1] 2

[1] 4

[1] 6

[1] 8

[1] 10
```

---

TOPIC 2:  ARITHMETIC AND BOOLEAN OPERATORS AND VALUES

| Operation | Description |
|---|---|
| x + y | Addition |
| x - y | Subtraction |
| x * y | Multiplication |
| x / y | Division |
| x ^ y | Exponentiation |
| x %% y | Modular arithmetic |
| x %/% y | Integer division |
| x == y | Test for equality |
| x <= y | Test for less than or equal to |
| x >= y | Test for greater than or equal to |
| x && y | Boolean AND for scalars |
| x \|\| y | Boolean OR for scalars |
| x & y | Boolean AND for vectors (vector x,y,result) |
| x \| y | Boolean OR for vectors (vector x,y,result) |

| Operation | Description |
|---|---|
| !x | Boolean negation |

TOPIC 3:  DEFAULT VALUES FOR ARGUMENTS

In R, function can have default values of it's parameters. It means while defining a function, it's parameters can be set values. It makes function more generic. For example

```
function_name ← function(a, b= 10){
   print(a+b);
}
```

# Calling function

```
function_name(10)
```

**Output:**

```
20
```

TOPIC 4:  RETURN VALUES

Many a times, we will require our functions to do some processing and return back the result. This is accomplished with the return() function in R.

```
check <- function(x) {
```

```
if (x > 0) {

result <- "Positive"

}

else if (x < 0) {

result <- "Negative"

}

else {

result <- "Zero"

}

return(result)

}
```

## TOPIC 5:  NO POINTERS IN R

R does not have variables corresponding to *pointers* or *references* like those of, say, the C language. This can make programming more difficult in some cases. (As of this writing, the current version of R has an experimental feature called *reference classes*, which may reduce the difficulty.)

For example, you cannot write a function that directly changes its arguments. In Python, for instance, you can do this:

```
>>> x = [13,5,12]

>>> x.sort()

>>> x

[5, 12, 13]
```

Here, the value of x, the argument to sort(), changed. By contrast, here's how it works in R:

```
> x <- c(13,5,12)

> sort(x)

[1]  5 12 13

> x
```

```
[1] 13  5 12
```

The argument to sort() does not change. If we do want x to change in this R code, the solution is to reassign the arguments:

## TOPIC 6:  WRITING YOUR OWN BINARY OPERATIONS IN R

You can invent your own operations! Just write a function whose name begins and ends with %, with two arguments of a certain type, and a return value of that type.

For example, here's a binary operation that adds double the second operand to the first:

```
> "%a2b%" <- function(a,b) return(a+2*b)

> 3 %a2b% 5

[1] 13
```

## TOPIC 7:  ANONYMOUS FUNCTIONS

An Anonymous Function (also known as a *lambda experssion*) is a function definition that is not bound to an identifier. That is, it is a function that is created and used, but never assigned to a variable.

Anonymous functions are those functions which are not assigned a variable name. These functions are also known as lambda functions (just like the ones in python) These functions are just created for the time being and are used without defining a variable name to them. The "*apply" function family mostly uses anonymous functions in R. This recipe demonstrates an example of anonymous functions in R.

Step 1 - Define a function using sapply()

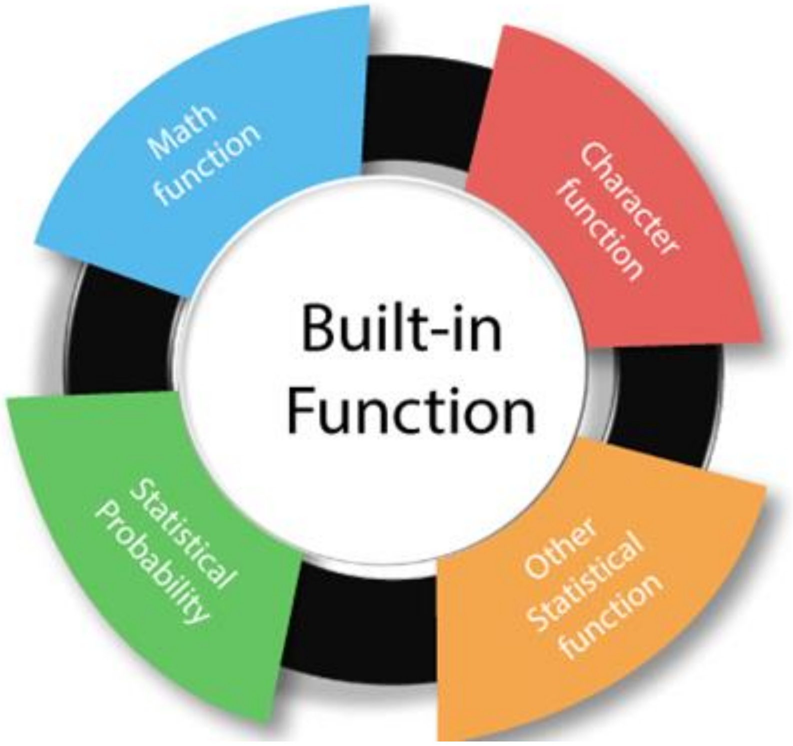
```
sapply(1:20,function(i) i + 5) #Using sapply()
```

```
 "Output of code is:"
6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

---

## TOPIC 8:  MATH FUNCTIONS IN R

The functions which are already created or defined in the programming framework are known as a built-in function. R has a rich set of functions that can be used to perform almost every task for the user. These built-in functions are divided into the following categories based on their functionality.



Math Functions

R provides the various mathematical functions to perform the mathematical calculation. These mathematical functions are very helpful to find absolute value, square value and much more calculations. In R, there are the following functions which are used:

| S. No | Function | Description | Example |
|---|---|---|---|
| **1.** | abs(x) | It returns the absolute value of input x. | x<- -4 |

| | | | |
|---|---|---|---|
| | | | print(abs(x))<br>**Output**<br>[1]  4 |
| **2.** | sqrt(x) | It returns the square root of input x. | x<- 4<br>print(sqrt(x))<br>**Output**<br>[1]  2 |
| **3.** | ceiling(x) | It returns the smallest integer which is larger than or equal to x. | x<- 4.5<br>print(ceiling(x))<br>**Output**<br>[1]  5 |
| **4.** | floor(x) | It returns the largest integer, which is smaller than or equal to x. | x<- 2.5<br>print(floor(x))<br>**Output**<br>[1]  2 |
| **5.** | trunc(x) | It returns the truncate value of input x. | x<- c(1.2,2.5,8.1)<br>print(trunc(x))<br>**Output**<br>[1]  1  2  8 |
| **6.** | round(x, digits=n) | It returns round value of input x. | x<- -4<br>print(abs(x))<br>**Output**<br>4 |

|  |  |  |  |
|---|---|---|---|
| **7.** | cos(x), sin(x), tan(x) | It returns cos(x), sin(x) value of input x. | x<- 4<br>print(cos(x))<br>print(sin(x))<br>print(tan(x))<br>**Output**<br>[1]  -06536436<br>[2]  -0.7568025<br>[3]  1.157821 |
| **8.** | log(x) | It returns natural logarithm of input x. | x<- 4<br>print(log(x))<br>**Output**<br>[1]  1.386294 |
| **9.** | log10(x) | It returns common logarithm of input x. | x<- 4<br>print(log10(x))<br>**Output**<br>[1]  0.60206 |
| **10.** | exp(x) | It returns exponent. | x<- 4<br>print(exp(x))<br>**Output**<br>[1]  54.59815 |

## TOPIC 9:  FUNCTIONS FOR STATISTICAL DISTRIBUTIONS

R has functions to handle many probability distributions. The table below gives the names of the functions for each distribution and a link to the on-line documentation that is the authoritative reference for how the functions are used. But don't read the on-line documentation yet. First, try the examples in the sections following the table.

| Distribution | Functions | | | |
|---|---|---|---|---|
| Beta | pbeta | qbeta | dbeta | Rbeta |
| Binomial | pbinom | qbinom | dbinom | rbinom |
| Cauchy | pcauchy | qcauchy | dcauchy | rcauchy |
| Chi-Square | pchisq | qchisq | dchisq | rchisq |
| Exponential | pexp | qexp | dexp | Rexp |
| F | pf | qf | df | Rf |
| Gamma | pgamma | qgamma | dgamma | rgamma |
| Geometric | pgeom | qgeom | dgeom | rgeom |
| Hypergeometric | phyper | qhyper | dhyper | rhyper |
| Logistic | plogis | qlogis | dlogis | Rlogis |
| Log Normal | plnorm | qlnorm | dlnorm | rlnorm |
| Negative Binomial | pnbinom | qnbinom | dnbinom | rnbinom |
| Normal | pnorm | qnorm | dnorm | rnorm |
| Poisson | ppois | qpois | dpois | Rpois |
| Student t | pt | qt | dt | Rt |

| | | | | |
|---|---|---|---|---|
| Studentized Range | ptukey | qtukey | dtukey | rtukey |
| Uniform | punif | qunif | dunif | Runif |
| Weibull | pweibull | qweibull | dweibull | rweibull |
| Wilcoxon Rank Sum Statistic | pwilcox | qwilcox | dwilcox | rwilcox |
| Wilcoxon Signed Rank Statistic | psignrank | qsignrank | dsignrank | rsignrank |

---

## TOPIC 10: SORTING IN R

```
x <- c(8,2,4,1,-4,NA,46,8,9,5,3)

order(x,na.last = TRUE)
```

The above code gives the following output:

```
5 4 2 11 3 10 1 8 9 7 6


order(x,decreasing=TRUE,na.last=TRUE)
```

The above code gives the following output:

```
7 9 1 8 10 3 11 2 4 5 6
```

## TOPIC 11: LINEAR ALGEBRA OPERATIONS ON VECTORS AND MATRICES

```
> y

[1]  1  3  4 10

> 2*y

[1]  2  6  8 20
```

If you wish to compute the inner product (or dot product) of two vectors, use crossprod(), like this:

```
> crossprod(1:3,c(5,12,13))

     [,1]

[1,]   68
```

The function computed 1 · 5 + 2 · 12 + 3 · 13 = 68.

## TOPIC 12: SET OPERATIONS IN R

union(x, y)
intersect(x, y)

```
setdiff(x, y)
setequal(x, y)

is.element(el, set)
```

Eg

```
(x <- c(sort(sample(1:20, 9)), NA))
(y <- c(sort(sample(3:23, 7)), NA))
union(x, y)
intersect(x, y)
setdiff(x, y)
setdiff(y, x)
setequal(x, y)

## True for all possible x & y :
setequal( union(x, y),
          c(setdiff(x, y), intersect(x, y), setdiff(y, x)))

is.element(x, y) # length 10
is.element(y, x) # length  8
```

---

## TOPIC 13:  SIMULATION PROGRAMMING

```
> set.seed(1)
> sample(1:10, 4)
[1] 9 4 7 1
> sample(1:10, 4)
```

```
[1] 2 7 3 6
>
> ## Doesn't have to be numbers
> sample(letters, 5)
[1] "r" "s" "a" "u" "w"
>
> ## Do a random permutation
> sample(1:10)
 [1] 10  6  9  2  1  5  8  4  3  7
> sample(1:10)
 [1]  5 10  2  8  6  1  4  3  9  7
>
> ## Sample w/replacement
> sample(1:10, replace = TRUE)
 [1]  3  6 10 10  6  4  4 10  9  7
```

# UNIT V

## TOPIC 1: CREATING GRAPHS IN R

In R, graphs are typically created interactively.

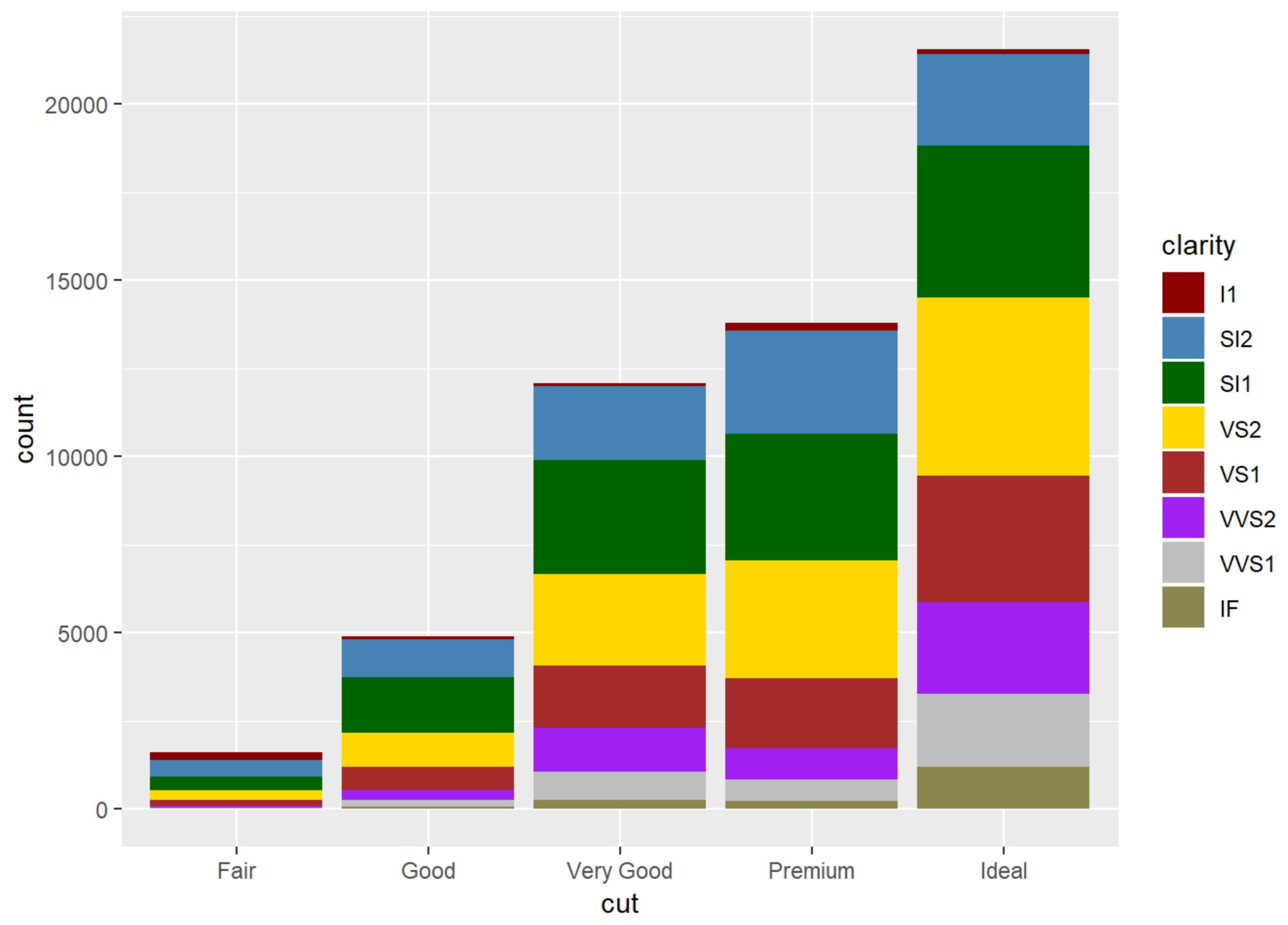
```
# Creating a Graph
attach(mtcars)
plot(wt, mpg)
abline(lm(mpg~wt))
title("Regression of MPG on Weight")
```

The **plot( )** function opens a graph window and plots weight vs. miles per gallon. The next line of code adds a regression line to this graph. The final line adds a title.

## TOPIC 2: CUSTOMIZING GRAPHS

Customization can improve the clarity and attractiveness of a graph.

```
# specify fill color manually
library(ggplot2)
ggplot(diamonds, aes(x = cut, fill = clarity)) +
  geom_bar() +
  scale_fill_manual(values = c("darkred", "steelblue",
                               "darkgreen", "gold",
                               "brown", "purple",
                               "grey", "khaki4"))
```



---

## TOPIC 3:  SAVING GRAPHS TO FILES

i) Saving graphs as jpeg

ii) Saving graphs as png

iii) Saving graphs as pdf

iv) Saving graphs to files using export option

Eg.

We can create a jpeg file and then save a graph in it by using the **jpeg()** function.

```
jpeg("barplot.jpeg")

barplot(airquality$Ozone,

        main="Ozone concentration in air",

        xlab='ozone levels',

        col="blue",

        horiz=TRUE)

dev.off()
```

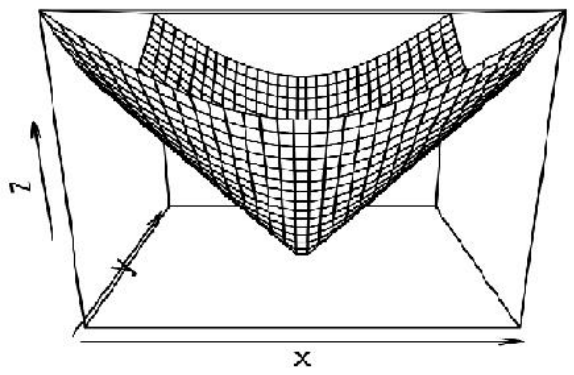
## TOPIC 4:  CREATING 3D PLOTS IN R

3D plot in R Language is used to add title, change viewing direction, and add color and shade to the plot. The **persp()** function which is used to create 3D surfaces in perspective view. This function will draw perspective plots of a surface over the x–y plane. persp() is defines as a generic function. Moreover, it can be used to superimpose additional graphical elements on the 3D plot, by lines() or points(), using the function trans3d().

```
# To illustrate simple right circular cone

cone <- function(x, y){

sqrt(x ^ 2 + y ^ 2)

}



# prepare variables.

x <- y <- seq(-1, 1, length = 30)

z <- outer(x, y, cone)
```

```
# plot the 3D surface

persp(x, y, z)
```

**Output:**



---

TOPIC 5:  WRITING C/C++ FUNCTIONS TO BE CALLED FROM R

We can call a function written in C from R using .C() or .call(). We begin with .C as it is simpler.

**Using .C()**

The basic recipe for using .C is as follows:

- Write a function in .C that returns void and stores the desired result in one or more arguments expressly created for this purpose. Your function should have a C_ prefix.
- Compile your function into a shared library accessible to R using this command: R CMD SHLIB my_func.c with my_func.c replaced by your c program.
- Within an active R session, link to the shared library using dyn.load("my_func.so").
- Write an R wrapper to call your C function using .C with the following syntax: .C("C_my_func", arg1, arg2). The reason to use an R wrapper is to ensure that the arguments passed are of the correct size and type in order to avoid potentially fatal errors.

**Using .Call()**

We can use .Call() to create and modify R level objects directly using functions in C. R level objects are of type SEXP for “S-expression”. These types are defined in the header file Rinternals.h which should be included in all C functions to be called with .Call().

When we create R level objects, we must protect them from garbage collection using PROTECT or a similar function. We later UNPROTECT them to allow the memory to be reallocated.

Recall that scalars in R are length one vectors. Consequently, the must be coerced to C type scalars to be uses as such. This can be done using the C functions as*: asLogical, asInteger, asReal, CHAR(asChar()).

**Interfacing with C++ via Rcpp**

The Rcpp package greatly simplifies the process of exposing functions written in C++ to R.

**Using sourceCpp**

We can use the function sourceCpp to read a function written in C++ into R interactively. The function takes care of the compilation using R CMD SHLIB and automatically generates an R wrapper for the underlying function. The shared library and other files will be written to the directory specified by cacheDir which defaults to a temporary directory for automated clean up.

To use a C++ function via sourceCpp:

1. Write a your C++ function in a file with extension .cpp.
2. In the source file, be sure to #include <Rcpp.h>.
3. Designate functions exposed to R using the tag // [[Rcpp::export]].
4. Compile and source your function using sourceCpp() and a link to the file.

## TOPIC 6:  USING R FROM PYTHON

Thanks to the rpy2 package, Pythonistas can take advantage of the great work already done by the R community. rpy2 provides an interface that allows you to run R in Python processes. Users can move between languages and use the best of both programming languages.

### Installing rpy2

First up, install the necessary packages. You must have Python >=3.7 and R >= 4.0 installed to use rpy2 3.5.2. Once R is installed, install the rpy2 package by running pip install rpy2. If you'd like to see where you installed rpy2 on your machine, you can run python -m rpy2.situation.
If you are working in a Jupyter notebook, you may want to see your ggplot2 plots within your notebook. Run conda install r-ggplot2 in your Jupyter environment so that they show up.

*Installation*
```
pip install rpy2
conda install r-ggplot2
```

---

Importing rpy2 packages and subpackages

Then, you’ll import the packages and subpackages. Import the top-level rpy2 package by running `import rpy2`. Import the top-level subpackage `robjects` with `import rpy2.robjects as robjects`. Running `robjects` also initializes R in the current Python process.

There are a couple of other steps to make working in a notebook a bit easier:

- rpy2 customizes the display of R objects, such as data frames in a notebook. Run `rpy2.ipython.html.init_printing()` to enable this customization.
- You may want to see ggplot2 objects in an output cell of a notebook. Enable this with `from rpy2.ipython.ggplot import image_png`.

---

*Import rpy2 packages and subpackages*
```
import rpy2
import rpy2.robjects as robjects

## To aid in printing HTML in notebooks
import rpy2.ipython.html
rpy2.ipython.html.init_printing()

## To see plots in an output cell
from rpy2.ipython.ggplot import image_png
```

---

## Installing and loading R packages with rpy2

Installing and loading R packages are often the first steps in R scripts. The rpy2 package provides a function rpy2.robjects.packages.importr() that mimics these steps. Run the below, where you'll also import the function data() for later.

```
from rpy2.robjects.packages import importr, data
```

Use importr() to load the utils and base packages, which normally come preinstalled with R.

```
utils = importr('utils')
base = importr('base')
```

## Using the Python interface for R

With everything set up, you can begin using the Python interface! However, this is not as simple as writing R code in your Jupyter notebook. You have to 'translate' R functions to call them from Python.

---- UNITS I, III AND V COMPLETED----

ALL THE BEST