B. Sc. Part-III Semester V

SUBJECT – STATISTICS - XII

DSE-E16: R-Programming and Quality Management

Unit-1: R Programming:

1.1: Introduction : History, Feathers of R, Character sets,
Identifiers: Variable, Constants, Symbolic constant, key words, Data Types and Data Structure.

Operators: Arithmetic, relational, logical, assignment, increasing, decreasing, special operators, Character vectors, Input and output functions, Data Import and Export function, Basic built-in function

1.2: Programming: Algorithm, flow chart, Structure of programme,
Conditional Statements: If, if else, **Loops:** for, while, Unconditional Statements, Writing of your own functions, Diagrams and Graphs, Simple programmes on

1) Finding Area of circle.
2) To check whether the given integer is positive or negative.
3) Reverse a given number.
4) To find greatest of three numbers.
5) Find Prime numbers in a given range.
6) To check if number is odd or even.
7) To check leap year.
8) To find sum of first n natural numbers.
9) To find AM, GM, and HM for ungrouped data.
10) To find Mean deviation, Variance, Standard deviation for ungroupeddata.
11) To generate random numbers from discrete distributions.
12) To generate random numbers from continuous distributions.

1.1 : Introduction

History

R programming language and software environment for statistical analysis, graphics, representation and reporting. R was initially written by Ross Ihaka and Robert Gentleman at the Department of Statistics of the University of Auckland in Auckland, New Zealand. R made its first appearance in 1993.

The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions. R allows integration with the procedures written in the C, C++, .Net, Python or FORTRAN languages for efficiency.
R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac. R is free software distributed under a GNU-style copy left, and an official part of the GNU project called GNU S.

The first official release came in 1995. The Comprehensive R Archive Network (CRAN) was officially announced 23 April 1997 with 3 mirrors and 12 contributed packages. The first official “stable beta" version (v1.0) was released on 29 February 2000.

Features of R

As stated earlier, R is a programming language and software environment for statistical analysis, graphics representation and reporting. The following are the important features of R:

- R is a well-developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.
- R is use large collection of tools of data collection.
- R has an effective data handling and storage facility,
- R provides a suite of operators for calculations on arrays, lists, vectors and matrices.
- R provides a large, coherent and integrated collection of tools for data analysis.
- R provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.

Identifiers

Variables are used to store data, whose value can be changed according to our need. Unique name given to variable (function and objects as well) is identifier.

❖ **Rules for writing Identifiers**

1. Identifiers can be a combination of letters, digits, period (.) and underscore (_).

2. It must start with a letter or a period. If it starts with a period, it cannot be followed by a digit.

3. Reserved words in R cannot be used as identifiers.

Example:

Valid identifiers

Sum, .fine.with.dot, this_is_acceptable, Number5

Invalid identifiers

tot@l, 5um, _fine, TRUE, .0ne

Below are the list of the identifier to study in that paper

1. Variable
2. Constants
3. Symbolic constant
4. key words
5. Data Types
6. Data Structure.

1. Variable

A variable provides us with named storage that our programs can manipulate. A variable in R can store an atomic vector, group of atomic vectors or a combination of many R objects. A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot not followed by a number. Variables are used to store dat

Types of variable

1) **Boolean Variables:** This is the simplest type of variable. It contains a single bit, and indicate a binary result (0 and 1, yes and no, or true and false).
e.g. a = TRUE
 b = FALSE

2) **Integer variables:** Numbers with no floating point are called integers. In R programming, sometimes it is difficult to declare a single integer. In most cases, try to do so will actually declare a numeric value.

3) **Numeric Variables:** Numeric variables are used to store numbers. It can contain floating point numbers.
   e.g. a = 1
   b = 3.14

4) **Characters Variables:** Character variables are used to store non-numeric data. Unlike other programming languages, there are a no differences between characters and strings in R.
   e.g. a a = “x”
   b = “6”

5) **String Variables:** String variables are those variables which contain one or more characters.
   e.g. x= “abcd2”
   y= “Hello World”
   = “x”

2. Constants

Constants are entities within a program whose value can't be changed. There are 2 basic types of constant. These are numeric constants and character constants.

1) **Numeric Constants:** numeric constants are the numbers which can be integer, double or complex. You can check the type of constant through the typeof() function. Numeric constant suffix with ***L*** are the integer type and suffix with ***i*** are called complex type.
   e.g.
   ```
   > typeof(10)
   [1] "double"
   > typeof(10L)
   [1] "Integer"
   > typeof(10i)
   [1] "complex"
   ```

2) **Character Constant:** Character constant can be declared using either single quote (‘ ’) or double quote (“ ”).
   e.g.
   ```
   > typeof(“nikita”)
   [1] “character”
   > typeof(‘hello’)
   [1] “character”
   ```

3) **Built-in Constants:** Some of the built-in constants of R along with their values are shown below:
   e.g.
   ```
   > LETTERS
   [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T"
   [21] "U" "V" "W" "X" "Y" "Z"
   ```

```
> letters
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"
[21] "u" "v" "w" "x" "y" "z"
> month.name
[1] "January"   "February"  "March"     "April"     "May"       "June"
[7] "July"      "August"    "September" "October"   "November"  "December"
> month.abb
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
> pi
[1] 3.141593
```

3. key words

In programming, a keyword is a word which is reserved by a program because it has a special meaning. A keyword can be a command or a parameter. Like in C, C++, Java, there is also a set of keywords in R. A keyword can't be used as a variable name. Keywords are also called as "reserved names."
There are the following keywords as per **?reserved** or **help(reserved)** command:

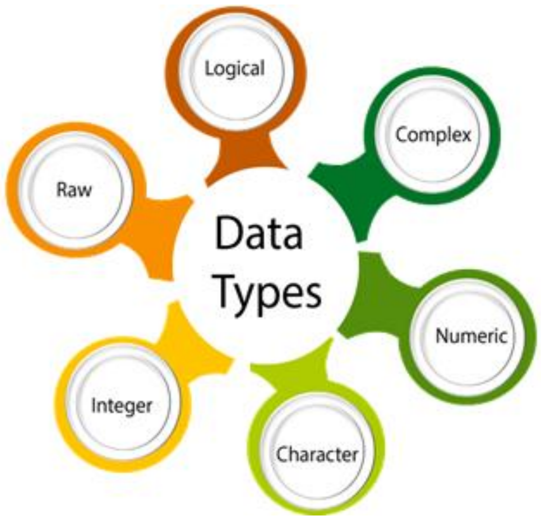
if	else	repeat
while	function	for
next	break	TRUE
FALSE	NULL	Inf
NaN	NA	NA_integer_
NA_real_	NA_complex_	NA_character_

4. Data Types

In programming languages, we need to use various variables to store various information. Variables are the reserved memory location to store values. As we create a variable in our program, some space is reserved in memory.

In R, there are several data types such as integer, string, etc. The operating system allocates memory based on the data type of the variable and decides what can be stored in the reserved memory.

There are the following data types which are used in R programming:



1) Numeric Data Type :

The numeric data type is for numeric values. It is the default data type for numbers in R.
Examples of numeric values would be 1, 34.5, 3.145, -24, -45.003, etc.

e.g.
```
> num <- 1
> class(num)
[1] "numeric"
> typeof(num)
[1] "double"
```

Note: When R stores a number in a variable, it converts the number into a 'double' value or a decimal type with at least two decimal places. This means that a value such as '1' is stored as 1.00 with a type of double and a class of numeric.

2) Integers Data Type :

The Integer data type is used for integer values. To store a value as an integer, we need to specify it as such. The integer data type is commonly used for discrete only values like unique ids. We can store as well as convert a value into an integer type using the as.integer() function.

e.g,

```
> int <- as.integer(16)
> class(int)
[1] "integer"
> typeof(int)
[1] "integer"
> num=1
> int2 <- as.integer(num)
> int2
[1] 1
> class(int2)
[1] "integer"
> typeof(int2)
[1] "integer”
```

Note: We can also use the capital ‘L’ notation to denote that a particular value is of the integer data type.

e.g. > int3 <- 5L

```
> class(int3)
[1] "integer"
> typeof(int3)
[1] "integer"
```

3) Complex Data Type :

The complex data type is to store numbers with an imaginary component. Examples of complex values would be 1+2i, 3i, 4-5i, -12+6i, etc.

e.g > comp <- 22-6i

```
> class(comp)
[1] "complex"
> typeof(comp)
[1] "complex"
```

4) Logical Data Type :

The logical data type stores logical or boolean values of TRUE or FALSE.

e.g.

```
> logi <- FALSE

> class(logi)

[1] "logical"

> typeof(logi)

[1] "logical"
```

5) Character Data Type :

The character data type stores character values or strings. Strings in R can contain the alphabet, numbers, and symbols. The easiest way to denote that a value is of character type in R is to wrap the value inside single or double inverted commas.

e.g.

```
> char <- "dataflair1234"

> class(char)

[1] "character"

> typeof(char)

[1] "character"
```

Code :

```
comp <- 22-6i

int2 <- as.integer(comp)

int2

char2 <- as.character("hello")

char3 <- as.character(comp)

char2

char3

num2 <- as.numeric(int)
```

```
num2

int4 <- as.integer(num)

int4

comp2 <- as.complex(num)

comp2

comp2 <- as.complex(num)

char2 <- as.character(num)
```

5. Data Structure.

A data structure is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks. Data structures in R programming are tools for holding multiple values.

R’s base data structures are often organized by their dimensionality (1D, 2D, or nD) and whether they’re homogeneous (all elements must be of the identical type) or heterogeneous (the elements are often of various types). This gives rise to the six data types which are most frequently utilized in data analysis.

The most essential data structures used in R include:

- **Vectors**
- **Lists**
- **Data frames**
- **Matrices**
- **Arrays**
- **Factors**

a. 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.

```
e.g. > X = c(1, 3, 5, 7, 8)
     > X
     [1] 1 3 5 7 8
     > length(X)
     [1] 5
     > class(X)
     [1] "numeric"
```

b. 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.

e.g 1)
```
>X = list(1, 3, 5, 7, 8,"r")
> X
 [[1]]
[1] 1
[[2]]
[1] 3
[[3]]
[1] 5
[[4]]
 [1] 7
 [[5]]
[1] 8
[[6]]
[1] "r"
> length(X)
[1] 6
> class(X)
[1] "list"
```

2)
```
> empId = c(1, 2, 3, 4)
> empName = c("Debi", "Sandeep", "Subham", "Shiba")
> numberOfEmp = 4
> empList = list(empId, empName, numberOfEmp)
> print(empList)

[[1]]

[1] 1 2 3 4

[[2]]

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

[[3]]

[1] 4
```

c. Data frames

Data frames are generic data objects of R which are used to store the tabular data. Data frames 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.

```
e.g. > Name = c("Amiya", "Raj", "Asish")

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

  > Age = c(22, 25, 45)

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

  > print(df)

Output:

    Name   Language    Age

1   Amiya     R        22

2  Raj        Python   25

3  Asish      Java     45
```

d.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.

e.g.
```
A = matrix(

                c(1, 2, 3, 4, 5, 6, 7, 8, 9),

                nrow = 3, ncol = 3,

                byrow = TRUE

               )

> print(A)
```

Output:

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

[1,]    1    2    3

[2,]    4    5    6

[3,]    7    8    9
```

e. 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.

```
e.g. > A = array(
                 c(1, 2, 3, 4, 5, 6, 7, 8),
                 dim = c(2, 2, 2)
                )
       > print(A)
```

Output:

```
, , 1                          , , 2

    [,1] [,2]                  [,1] [,2]
[1,]    5    7             [1,]    1    3
[2,]    2    4             [2,]    2    4
```

f. 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.

e.g.

```
> fac = factor(c("Male", "Female", "Male", "Male", "Female", "Male", "Female"))

> print(fac)
```

Output:

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

Levels: Female Male
```

Character vector:

To create a character vector in R we can enclose the vector values in double quotation marks but if we want to use a data frame values to create a character vector then as.character function can be used.

A character is held as a one-byte integer in memory. In R, there are two different ways to create a character data type value, i.e., using as.character() function and by typing string between double quotes("") or single quotes('').

A vector which contains character elements is known as an integer vector.

e.g.

```
d<-'shubham'
e<-"Arpita"
f<-65
f<-as.character(f)
d
e
f
char_vec<-c(1,2,3,4,5)
char_vec<-as.character(char_vec)
char_vec1<-c("shubham","arpita","nishka","vaishali")
char_vec
class(d)
class(e)
class(f)
class(char_vec)
class(char_vec1)
```

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.

Following are the types of the operators:

1. Arithmetic	2. Relational
3. Logical	4. Assignment
5. Increasing	6.Decreasing
7.Specialoperators	

1. Arithmetic operator :

Arithmetic operations 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 operations are performed element-wise at the corresponding positions of the vectors.

a)Addition operator (+):

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

e.g

```
Input : a <- c (1, 0.1)
        b <- c (2.33, 4)
        print (a+b)
Output : 3.33  4.10
```

b)Subtraction Operator (-):

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

```
e.g.
Input : a <- 6
        b <- 8.4
        print (a-b)
Output : -2.4
```

c)Multiplication Operator (*):

The multiplication of corresponding elements of vectors and Integers are multiplied with the use of '*' operator

e.g.

```
Input : a <- c (1, 0.1)
        b <- c (2.33, 4)
        print (a+b)
```

d)Division Operator (/):

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

e.g.

```
Input : a <- 1
        b <- 0
        print (a/b)
Output : -Inf
```

e)Power Operator (^):
The first operand is raised to the power of the second operand

e.g..

```
Input : list1 <- c(2, 3)
        list2 <- c(2,4)
        print(list1^list2)
Output : 4  81
```

f)Modulo Operator (%%):
The remainder of the first operand divided by the second operand is returned.

e.g.

```
Input : list1<- c(2, 3)
        list2<-c(2,4)
        print(list1%%list2)
Output : 0  3
```

Examples:

```
vec1 <- c(0, 2)
vec2 <- c(2, 3)

# Performing operations on Operands
cat ("Addition of vectors :", vec1 + vec2, "\n")
cat ("Subtraction of vectors :", vec1 - vec2, "\n")
cat ("Multiplication of vectors :", vec1 * vec2, "\n")
cat ("Division of vectors :", vec1 / vec2, "\n")
cat ("Modulo of vectors :", vec1 %% vec2, "\n")
cat ("Power operator :", vec1 ^ vec2)
```

Operator	Description	Example
+	Adds two vectors	`v <- c( 2,5.5,6)` `t <- c(8, 3, 4)` `print(v+t)` it produces the following result − `[1] 10.0  8.5  10.0`
−	Subtracts second vector from the first	`v <- c( 2,5.5,6)` `t <- c(8, 3, 4)` `print(v-t)` it produces the following result − `[1] -6.0  2.5  2.0`

*	Multiplies both vectors	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v*t)`
		it produces the following result −
		`[1] 16.0 16.5 24.0`

/	Divide the first vector with the second	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v/t)`
		When we execute the above code, it produces the following result −
		`[1] 0.250000 1.833333 1.500000`

%%	Give the remainder of the first vector with the second	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v%%t)`
		it produces the following result −
		`[1] 2.0 2.5 2.0`

^	The first vector raised to the exponent of second vector	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v^t)`
		it produces the following result −
		`[1]  256.000  166.375 1296.000`

1. Relational operator :

The relational operators carry out comparison operations between the corresponding elements of the operands. Returns a boolean TRUE value if the first operand satisfies the relation compared to the second. A TRUE value is always considered to be greater than the FALSE.

Less than (<):

Returns TRUE if the corresponding element of the first operand is less than that of the second operand. Else returns FALSE.

```
Input :  list1 <- c(TRUE, 0.1)
         list2 <- c(0,0.1)
         print(list1<list2)
Output :  FALSE FALSE
```

Less than equal to (<=):

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

```
Input :  list1 <- c(TRUE, 0.1)
         list2 <- c(0,0.1)
         print(list<=list2)
Output : FALSE TRUE
```

Greater than (>):

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

```
Input :  list1 <- c(TRUE, 0.1)
         list2 <- c(0,0.1)
        print(list2 > list2)
Output : TRUE FALSE
```

Greater than equal to (>=):

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

```
Input :  list1 <- c(TRUE, 0.1)
         list2 <- c(0,0.1)
        print(list2 >= list2)
Output : TRUE TRUE
```

Not equal to (!=):

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

```
Input : list1 <- c(TRUE, 0.1)
        list2 <- c(0,0.1)
        print(list1!=list2)
Output : TRUE FALSE
```

Examples:

```
x <- 5

> y <- 16

> x<y

[1] TRUE

> x>y

[1] FALSE

> x<=5

[1] TRUE

> y>=20

[1] FALSE
```

```
> y == 16
[1] TRUE
> x != 5
[1] FALSE
```

Operator	Description	Example

>	Checks if each element of the first vector is greater than the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v>t)`
		it produces the following result −
		`[1] FALSE  TRUE FALSE FALSE`

<	Checks if each element of the first vector is less than the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v < t)`
		it produces the following result −
		`[1]  TRUE FALSE  TRUE FALSE`

==	Checks if each element of the first vector is equal to the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v == t)`
		it produces the following result −
		`[1] FALSE FALSE FALSE  TRUE`

<=	Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v<=t)`
		it produces the following result −
		`[1]  TRUE FALSE  TRUE  TRUE`

>=	Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v>=t)`
		it produces the following result −
		`[1] FALSE  TRUE FALSE  TRUE`

!=	Checks if each element of the first vector is unequal to the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v!=t)`
		it produces the following result −
		`[1]  TRUE  TRUE  TRUE FALSE`

3.Logical Operator :

Logical operations 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 complex or real number.

Element-wise Logical AND operator (&):

Returns True if both the operands are True.

```
Input : 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 are True.

```
Input : 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.

```
Input : list1 <- c(0,FALSE)
Output : TRUE  TRUE
```

Logical AND operator (&&):

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

```
Input : 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 are True.

```
Input : list1 <- c(TRUE, 0.1)
        list2 <- c(0,4+3i)
        print(list1||list2)
Output : TRUE
```

Operator	Description	Example

&	It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE.	`v <- c(3,1,TRUE,2+3i)`
		 `t <- c(4,1,FALSE,2+3i)`
		 `print(v&t)`
		 it produces the following result −
		 `[1]  TRUE  TRUE FALSE  TRUE`

|	It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if one the elements is TRUE.	`v <- c(3,0,TRUE,2+2i)`
		 `t <- c(4,0,FALSE,2+3i)`
		 `print(v|t)`
		 it produces the following result −
		 `[1]  TRUE FALSE  TRUE  TRUE`

!	It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value.	`v <- c(3,0,TRUE,2+2i)`
		 `print(!v)`
		 it produces the following result −
		 `[1] FALSE  TRUE FALSE FALSE`

The logical operator && and || considers only the first element of the vectors and give a vector of single element as output.

&&	Called Logical AND operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE.	`v <- c(3,0,TRUE,2+2i)`
		 `t <- c(1,3,TRUE,2+3i)`
		 `print(v&&t)`
		 it produces the following result −
		 `[1] TRUE`

||	Called Logical OR operator. Takes first element of both the vectors and gives the TRUE if one of them is TRUE.	
		```
		v <- c(0,0,TRUE,2+2i)
		t <- c(0,3,TRUE,2+3i)
		print(v||t)
		```
		 it produces the following result − `[1] FALSE`

4. Assignment Operator:

Assignment operators are used to assign values to various data objects in R. The objects may be integers, vectors, or functions. These values are then stores 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.

```
Input : vec1 = c("ab", TRUE)
        print (vec1)
Output : "ab"   "TRUE"
```

Right Assignment (-> or ->>):

Assigns value to a vector.

```
Input : c("ab", TRUE) ->> vec1
        print (vec1)
Output : "ab"   "TRUE"
```

Operator	Description	Example

<−
or
=
or
<<−
	Called Left Assignment	
		```
		v1 <- c(3,1,TRUE,2+3i)
		v2 <<- c(3,1,TRUE,2+3i)
		v3 = c(3,1,TRUE,2+3i)
		print(v1)
		print(v2)
		print(v3)
		```
		 it produces the following result −
		```
		[1] 3+0i 1+0i 1+0i 2+3i
		[1] 3+0i 1+0i 1+0i 2+3i
		[1] 3+0i 1+0i 1+0i 2+3i
		```

->
or
->>
	Called Right Assignment	`c(3,1,TRUE,2+3i) -> v1`
		`c(3,1,TRUE,2+3i) ->> v2`
		`print(v1)`
		`print(v2)`
		
		it produces the following result −
		
		`[1] 3+0i 1+0i 1+0i 2+3i`
		`[1] 3+0i 1+0i 1+0i 2+3i`

7.Special Operators or Miscellaneous Operators :

These are the mixed operators that simulate the printing of sequences and assignment of vectors, either left or right-handed.

%in% Operator:

Checks if an element belongs to a list and returns a boolean value TRUE if the value is present else FALSE.

```
Input : val <- 0.1
        list1 <- c(TRUE, 0.1,"apple")
        print (val %in% list1)
Output : TRUE

Checks for the value 0.1 in the specified list. It exists, therefore,
prints TRUE.
```

Colon Operator(:):

Prints a list of elements starting with the element before the color to the element after it.

```
Input :  print (1:5)
Output : 1 2 3 4 5

Prints a sequence of the numbers from 1 to 5.
```

%*% Operator:

This operator is used to multiply a matrix with its transpose. Transpose of the matrix is obtained by interchanging the rows to columns and columns to rows. The number of columns of first matrix must be equal to number of rows of second matrix. Multiplication of the matrix A with its transpose, B, produce a square matrix.

```
Input :  mat = matrix(c(1,2,3,4,5,6),nrow=2,ncol=3)
         print (mat)
         print( t(mat))
         pro = mat %*% t(mat)
         print(pro)
Output :     [,1] [,2] [,3]      #original matrix of order 2x3
        [1,]   1    3    5
        [2,]   2    4    6
             [,1] [,2]           #transposed matrix of order 3x2
```

```
[1,]    1    2
[2,]    3    4
[3,]    5    6
     [,1] [,2]          #product matrix of order 2x2
[1,]   35   44
[2,]   44   56
```

Operator	Description	Example

:	Colon operator. It creates the series of numbers in sequence for a vector.	
		```
		v <- 2:8
		print(v)
		```
		 it produces the following result − `[1] 2 3 4 5 6 7 8`

%in%	This operator is used to identify if an element belongs to a vector.	
		```
		v1 <- 8
		v2 <- 12
		t <- 1:10
		print(v1 %in% t)
		print(v2 %in% t)
		```
		 it produces the following result −
		```
		[1] TRUE
		[1] FALSE
		```

%*%	This operator is used to multiply a matrix with its transpose.	
		```
		M = matrix( c(2,6,5,1,10,4), nrow = 2,ncol = 3,byrow = TRUE)
		t = M %*% t(M)
		print(t)
		```
		 it produces the following result −
		```
		      [,1] [,2]
		[1,]   65   82
		[2,]   82  117
		```

Input/Output Functions in R

With R, we can read inputs from the user or a file using simple and easy-to-use functions. Similarly, we can display the complex output or store it to a file using the same. R's base package has many such functions, and there are packages that provide functions that can do the same and process the information in the required ways at the same time.

In this article, you'll get the answers to these:

- How to Read user input in R?
- How to display output in R?

How to Read User Input in R?

In R, there are multiple ways to read and save input given by the user. here are a few of them:

1. readline() function

We can read the input given by the user in the terminal with the **readline()** function.
Code:

```
input_read <- readline()
```

User Input:
412803 is pin code of wai
Code:

```
input_read
```

2. scan() function

We can also use the scan() function to read user input. This function, however, can only read numeric values and returns a numeric vector. If a non-numeric input is given, the function gives an error.

E.g.:

1)

```
input_scan <- scan()
```

User Input:

```
 34 54 65 75 25
```

```
input_scan
```

2)

```
input_scan2 <- scan()
```

User Input:

```
 34 566 2 a 2+1i
```

```
input_scan2
```

output:

```
Error in scan() : scan() expected ‘a real’, got ‘a’
```

How to Display Output in R?

To display the output of your program to the screen, you can use one of the following functions:

1. **print() functions**

We can use the print() function to display the output to the terminal. The print() function is a generic function. This means that the function has a lot of different methods for different types of objects it may need to print. The function takes an object as the argument. For example:

Example 1:

```
print(input_read)
```

Example 2:

```
print(input_scan)
```

Example 3:

```
print("abc")
```

Example 4:

```
print(34)
```

2. cat() function

We can also use the cat() function to display a string. The cat() function concatenates all of the arguments and forms a single string which it then prints. For example:

Code:

```
cat("hello", "this","is","techvidvan",12345,TRUE)
```

Data Import and Export function

The collection of numerical value is known as data. Data can be different forms. To analyze data using R programming language, first import data in R. This can be different formats CSV or any other delimiter separated. After importing data they can be manipulate, analyze and report it.

Data Import

1) **Importing the data into csv files from the syntax**

- **Method-1:**

Using read.csv() function

Syntax:

read.csv(“path/file_name.csv”,header=TRUE)

Or

read.csv(file.choose(),header=TRUE)

- **Method-2:**

Syntax:

read.csv(“path.csv”,header=TRUE,sep= “,”)

where,

path=the path of the files to be imported

header=by default TRUE

sep=the seprated of values in each row if we give (,) in “ “

# i.e “ , “ then separated the values in each row by ‘ , ‘

2) Importing data from text file (txt) :

We can easily import or read txt file using basic R function read.table().

read.table() is used to read is files in table format.

Syntax-

read.table(“path.txt”,header=TRUE)

3) Importing data from delimited file:

R has a function read.delim() to read the delimited file in the least the file is by default separated by a which is represented by Sep= “ “, that separated can be a comma(,) dollar symbol ($) etc.
Syntax-
read.delim(“path.delim”, header=TRUE, sep= “ “)

4) Importing data from excel file:

For importing data from excel file in R first we have to install “openxlsx” using command install.packages(“openxlsx”)
Syntax-
read.xlsx(“path.xlsx”)

Exporting Data

Importing data in R is surely important for the user. However, exporting data from R to other platform is equally important as well may want to export the data from R workspace in to an excel file or csv or text file.

1) **Exporting the data into text files from R-**
   Syntax-
   write.table(R-data file,“path/file_name.txt”,row.names=FALSE)

2) **Exporting the data into csv files from R-**
   Syntax-
   write.csv(R-data file,“path/file_name.csv”)

3) **Exporting the data into excel files from R-**
   Syntax-
   write.xlsx(R-data file,“path/file_name.xlsv”)

R built-in functions

The function Which are already created or define in the programming frame work are known as a built in functions in R has a reach set of functions that are used to perform almost every task for user. These built-in function are divided into 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))
			Output
			[1] 4

2.	sqrt(x)	It returns the square root of input x.	x<- 4
			print(sqrt(x))
			Output
			[1] 2

3.	ceiling(x)	It returns the smallest integer which is larger than or equal to x.	x<- 4.5
			print(ceiling(x))
			Output
			[1] 5

4.	floor(x)	It returns the largest integer, which is smaller than or equal to x.	x<- 2.5
			print(floor(x))
			Output
			[1] 2

5.	trunc(x)	It returns the truncate value of input x.	x<- c(1.2,2.5,8.1)
			print(trunc(x))
			Output
			[1] 1 2 8

6.	cos(x), sin(x), tan(x)	It returns cos(x), sin(x) value of input x.	x<- 4
			print(cos(x))
			print(sin(x))
			print(tan(x))
			Output
			[1] -06536436
			[2] -0.7568025
			[3] 1.157821

7.	log(x)	It returns natural logarithm of input x.	x<- 4
			print(log(x))
			Output
			[1] 1.386294

8.	log10(x)	It returns common logarithm of input x.	x<- 4
			print(log10(x))
			Output
			[1] 0.60206

9.	exp(x)	It returns exponent.	x<- 4
			print(exp(x))
			Output
			[1] 54.59815

String Function

R provides various string functions to perform tasks. These string functions allow us to extract sub string from string, search pattern etc. There are the following string functions in R:

S. No	Function	Description	Example
1.	sub(pattern, replacement,x, ignore.case=FALSE, fixed=FALSE)	It finds pattern in x and replaces it with replacement (new) text.	st1<- "England is beautiful but no the part of EU" sub("England', "UK", st1) **Output** [1] "UK is beautiful but not a part of EU"
2.	paste(..., sep="")	It concatenates strings after using sep string to separate them.	paste('one',2,'three',4,'five') **Output** [1] one 2 three 4 five

3.	strsplit(x, split)	It splits the elements of character vector x at split point.	a<-"Split all the character" print(strsplit(a, "")) **Output** [[1]] [1] "split" "all" "the" "character"

4.	tolower(x)	It is used to convert the string into lower case.	st1<- "shuBHAm"
			print(tolower(st1))
			Output
			[1] shubham

5.	toupper(x)	It is used to convert the string into upper case.	st1<- "shuBHAm"
			print(toupper(st1))
			Output
			[1] SHUBHAM

Other Statistical Function

Apart from the functions mentioned above, there are some other useful functions which helps for statistical purpose. There are the following functions:

S. No	Function	Description	Example

1.	mean(x)	It is used to find the mean for x object	a<-c(0:10, 40)
			xm<-mean(a)
			print(xm)
			Output
			[1] 7.916667

2.	sd(x)	It returns standard deviation of an object.	a<-c(0:10, 40)
			xm<-sd(a)
			print(xm)
			Output
			[1] 10.58694

3.	median(x)	It returns median.	a<-c(0:10, 40)
			xm<-meadian(a)
			print(xm)
			Output
			[1] 5.5

4.	quantilie(x, probs)	It returns quantile where x is the numeric vector whose quantiles are desired and probs is a numeric vector with probabilities in [0, 1]	

5.	range(x)	It returns range.	a<-c(0:10, 40)
			xm<-range(a)
			print(xm)
			Output
			[1] 0 40

6.	sum(x)	It returns sum.	a<-c(0:10, 40)
			xm<-sum(a)
			print(xm)
			Output
			[1] 95

7.	min(x)	It returns minimum value.	a<-c(0:10, 40)
			xm<-min(a)
			print(xm)
			Output
			[1] 0

8.	max(x)	It returns maximum value	a<-c(0:10, 40)
			xm<-max(a)
			print(xm)
			Output
			[1] 40

Other Useful Functions

Function	**Description**
seq(*from* **,** *to***,** *by***)**	generate a sequence indices <- seq(1,10,2) #indices is c(1, 3, 5, 7, 9)

rep(*x***,** *ntimes***)**	repeat *x n* times
	y <- rep(1:3, 2)
	# y is c(1, 2, 3, 1, 2, 3)

Programming:

Algorithms:

To make R do anything at all, you write an R script. In your R script, you tell the computer, step by step, exactly what you want it to do, in the proper order. R then *executes* each line of your script, following each step according to how you have designe the script.

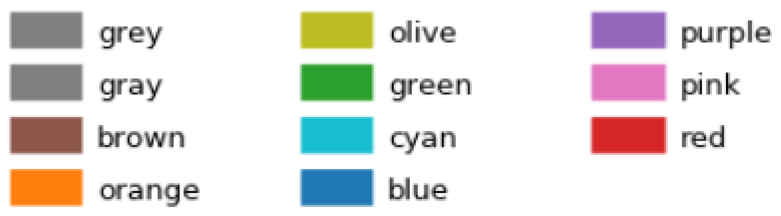
When you are telling the computer *what* to do, you also get to choose *how* it's going to be done. That's where computer algorithms come in. An algorithm is the basic technique used to get the job done. For example, let's say you have a friend arriving at the airport and she needs to get from the airport to your house. She might use the following algorithm:

1. Catch bus number 70 outside the baggage claim area
2. Transfer to bus 14 on Main St.
3. Get off at Elm St.
4. Walk two blocks north to my house.

You will note that the algorighm is written in the *order* in which it is to be executed. It wouldn't make sense to perform Step 4 (Walk two blocks north) until after the other three steps have been computed.

An R script is also written as an algorithm.

2.1 Example 1: Color Names



Let's say we have a bunch of words - say, the names of colors. We want to compute the average number of characters in these words. If we were going to do this by hand, we would use the following algorithm:

1. Create a list of the words
2. Count the number of characters in each word
3. Compute the average from Step 2.

Code:

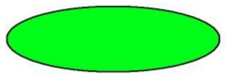
```
Col_name=c(“grey”, “grey”, “brown”, “orange”, “olive”, “green”, “cyan”, “blue”, “purple”, “pink”, “red”)

Col_length=nchar(Col_name)

mean(col_length)
```

Flowchart

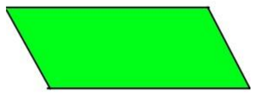
Flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of information and processing. The process of drawing a flowchart for an algorithm is known as "flowcharting".

Basic Symbols used in Flowchart Designs

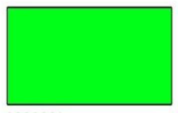
1. **Terminal:** The oval symbol indicates Start, Stop and Halt in a program's logic flow. A pause/halt is generally used in a program logic under some error conditions. Terminal is the first and last symbols in the flowchart.



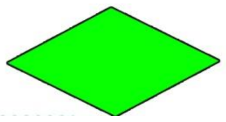
- **Input/Output:** A parallelogram denotes any function of input/output type. Program instructions that take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.



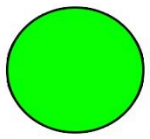
- **Processing:** A box represents arithmetic instructions. All arithmetic processes such as adding, subtracting, multiplication and division are indicated by action or process symbol.



- **Decision** Diamond symbol represents a decision point. Decision based operations such as yes/no question or true/false are indicated by diamond in flowchart.

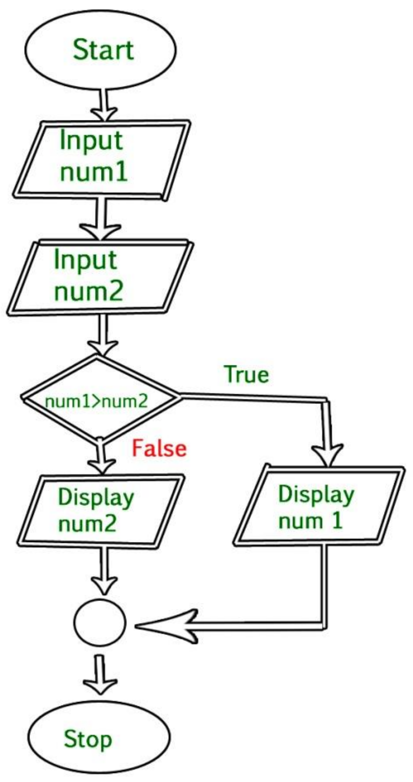


- **Connectors:** Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. It is represented by a circle.



- **Flow lines:** Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of flow of control and relationship among different symbols of flowchart.

  Example : Draw a flowchart to input two numbers from user and display the largest of two numbers



Conditional Statements

The conditional statement is mainly use for decision making on R-programming. Here we can discuss two type of conditional statement

1. If statement
2. If-else statement

1) If statement:

If statement is one of the Decision-making statements in the R programming language. It is one of the easiest decision-making statements. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

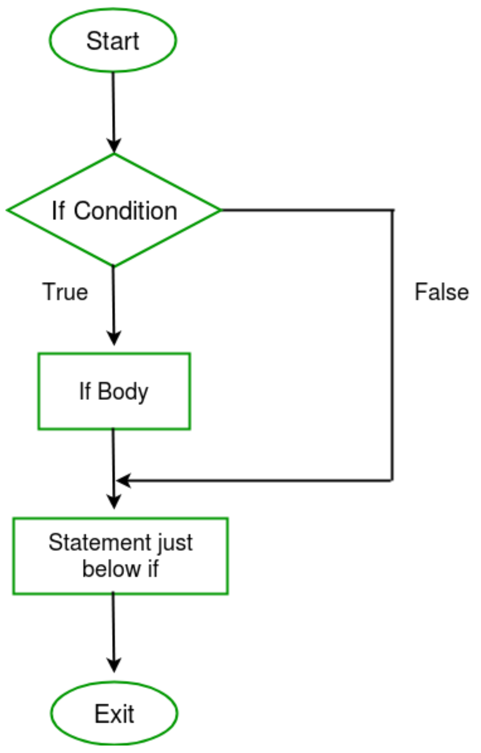
The basic structure of if statement is given by

Syntax:

```
if (expression) {
                    #statement to execute if condition is true
                }
```

If the expression is true, the statement gets executed. But if the expression is FALSE, nothing happens. The expression can be a logical/numerical vector, but only the first element is taken into consideration. In the case of numeric vector, zero is taken as FALSE, rest as TRUE.

Flowchart R Programming if statement



Examples:

1)# assigning value to variable a

```
  a <- 5
  if(a > 0)
  {
     print("Positive Number")  # Statement
   }
```

2) # Assigning value to variable x

```
   x <- 12
  if (x > 20)
   {
        print("12 is less than 20")  # Statement
    }
```

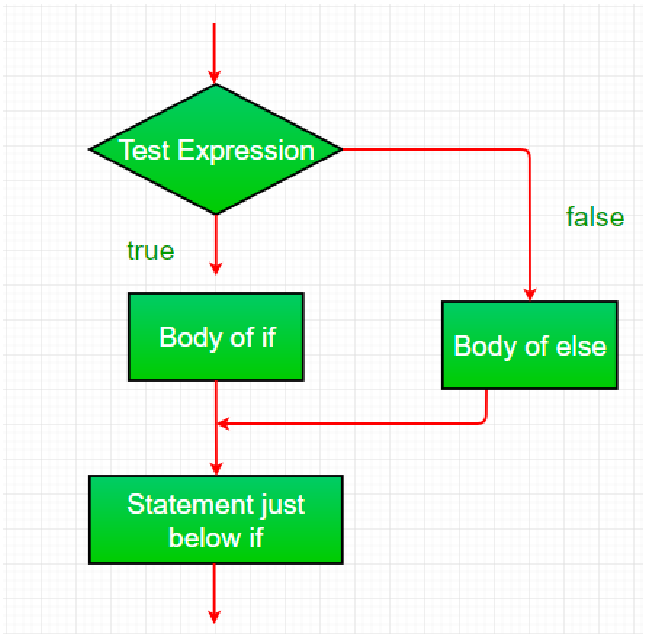
2) if-else statement:

The if-statement in Programming Language alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the R **else statement**. We can use the else statement with the if statement to execute a block of code when the condition is false.

Syntax of if-else statement in R Language:

```
if (condition)
{
   // Executes this block if
   // condition is true
} else
{
   // Executes this block if
   // condition is false
}
```

Flowchart if-else statement in R:



Examples:

1) 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"
```

Here in the above code, Firstly, x is initialized to 5, then if-condition is checked(x > 10), and it yields false. Flow enters the else block and prints the statement “5 is less than 10”.

2) x <- 5

```
  # Check if value is equal to 10
  if(x == 10)
     {
          print(paste(x, "is equal to 10"))
     }
else
    {
         print(paste(x, "is not equal to 10"))
    }
```

Output:

```
[1] "5 is not equal to 10"
```

Loops:

The loop statement are essential to construct systematically block stile programming. Here we can discuss two type of conditional statement

1. for loop
2. while loop

1)for loop:

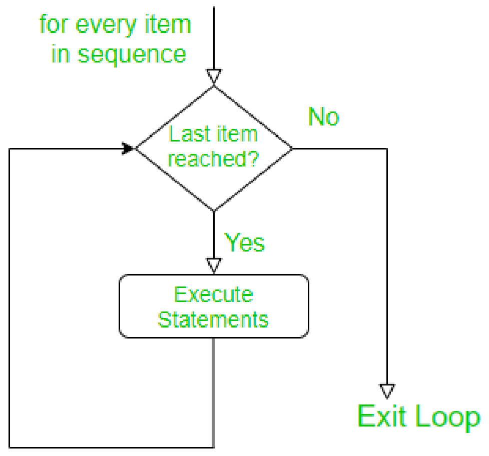
For loop in R Programming Language is useful to iterate over the elements of a list, dataframe, vector, matrix, or any other object. It means, the for loop can be used to execute a group of statements repeatedly depending upon the number of elements in the object. It is an entry controlled loop, in this loop the test condition is tested first, then the body of the loop is executed, the loop body would not be executed if the test condition is false.

For loop in R Syntax:

```
for (var in vector) {
                        statement(s)
                       }
```

Here, var takes on each value of vector during the loop. In each iteration, the statements are evaluated.

Flowchart of For loop in R:



For Loop in R

Examples:

1) # the use of for loop

```
  for (i in 1: 4)
   {
       print(i ^ 2)
   }
```

Output:

```
[1] 1

[1] 4

[1] 9

[1] 16
```

In the above example, we iterated over the range 1 to 4 which was our vector. Now there can be several variations of this general for loop. Instead of using a sequence 1:5, we can use the concatenate function as well.

2)# for loop along with concatenate

```
for (i in c(-8, 9, 11, 45))
 {
     print(i)
 }
```

Output:

```
[1] -8

[1] 9

[1] 11

[1] 45
```

Instead of writing our vector inside the loop, we can also define it beforehand.

3) # for loop with vector

```
x <- c(-8, 9, 11, 45)
for (i in x)
 {
    print(i)
  }
```

Output:

```
[1] -8

[1] 9

[1] 11

[1] 45
```

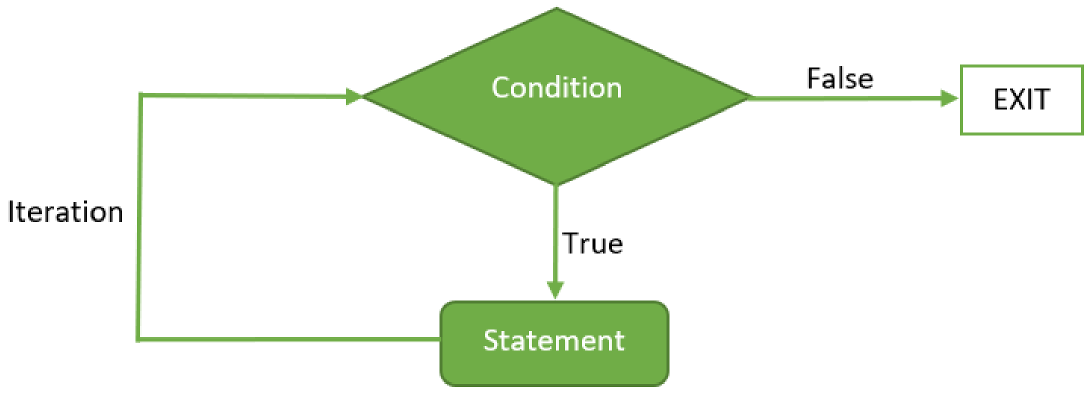
2)While Loop:

It is a type of control statement which will run a statement or a set of statements repeatedly unless the given condition becomes false. It is also an entry controlled loop, in this loop the test condition is tested first, then the body of the loop is executed, the loop body would not be executed if the test condition is false.

R – While loop Syntax:

```
while ( condition )
{
  statement
}
```

While loop Flow Diagram:



Examples:

Example 1: Program to display numbers from 1 to 5 using while loop in R.

```
# R program to demonstrate the use of while loop

val = 1

# using while loop
while (val <= 5)
{
    # statements
    print(val)
    val = val + 1
}
```

Output:

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

Initially, the variable value is initialized to 1. In each iteration of the while loop the condition is checked and the value of val is displayed and then it is incremented until it becomes 5 and the condition becomes false, the loop is terminated.

Example 2: Program to calculate factorial of a number.

```
n < - 5
factorial < - 1
i < - 1
while (i <= n)
{
   factorial = factorial * i
   i = i + 1
}
print(factorial)
```

Output:

```
[1] 120
```

Unconditional Statement

In R programming, we require a control structure to run a block of code multiple times. Loops come in the class of the most fundamental and strong programming concepts. A loop is a control statement that allows multiple executions of a statement or a set of statements. The word 'looping' means cycling or iterating.

Jump statements are used in loops to terminate the loop at a particular iteration or to skip a particular iteration in the loop.The two most commonly used jump statements in loops are:

- Break Statement
- Next Statement

Note: In R language continue statement is referred to as the next statement.

Break Statement

The break keyword is a jump statement that is used to terminate the loop at a particular iteration.

Syntax:

```
if (test_expression) {
                              break
                           }
```

Examples:

Example 1: Using break in For-loop

```
# R program for break statement in For-loop

no<- 1:10

for (val in no)
                  {
```

```
                  if (val == 5)
                {
                 print(paste("Coming out from for loop Where i = ", val))
                 break
                }
                  print(paste("Values are: ", val))
                }
```

Output:

```
[1] "Values are:  1"

[1] "Values are:  2"

[1] "Values are:  3"

[1] "Values are:  4"

[1] "Coming out from for loop Where i=  5"
```

Example 2: Using break statement in While-loop

```
# R Break Statement Example
a<-1
while (a < 10)
{
   print(a)
   if(a==5)
      break
   a = a + 1
}
```

Output:

```
[1] 1

[1] 2

[1] 3

[1] 4

[1] 5
```

Next Statement

The next statement is used to skip the current iteration in the loop and move to the next iteration without exiting from the loop itself.

Syntax:

```
if (test_condition)

          {

              next

          }
```

Example 1: Using next statement in For-loop

```
# R Next Statement Example

no<- 1:10

for (val in no)
{
   if (val == 6)
   {
      print(paste("Skipping for loop Where i =  ", val))
      next
   }
   print(paste("Values are:  ", val))
}
```

Output:

```
[1] "Values are:   1"

[1] "Values are:   2"

[1] "Values are:   3"

[1] "Values are:   4"

[1] "Values are:   5"

[1] "Skipping for loop Where i =   6"

[1] "Values are:   7"

[1] "Values are:   8"

[1] "Values are:   9"

[1] "Values are:   10"
```

Example 2: Using next statement in While-loop

```
# R Next Statement Example
x <- 1
while(x < 5)
{
   x <- x + 1;
   if (x == 3)
      next;
   print(x);
}
```

Output:

```
[1] 2

[1] 4

[1] 5
```

goto statement in R Programming

Goto statement in a general programming sense is a command that takes the code to the specified line or block of code provided to it. This is helpful when the need is to jump from one programming section to the other without the use of functions and without creating an abnormal shift.

Unfortunately, R doesn't support goto but its algorithm can be easily converted to depict its application. By using following methods this can be carried out more smoothly:

- Use of if and else
- Using break, next and return

Functions in R Programming

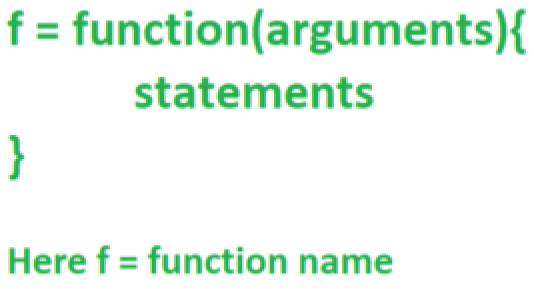
Functions are useful when you want to perform a certain task multiple times. A function accepts input arguments and produces the output by executing valid R commands that are inside the function. In R Programming Language when you are creating a function the function name and the file in which you are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

Types of function in R Language

- **Built-in Function:** Built function R is sq(), mean(), max(), these function are directly call in the program by users.
- **User-defile Function:** R language allow us to write our own function.

Functions in R Language

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



```
f = function(arguments){
        statements
}

Here f = function name
```

Note: In the above syntax f is the function name, this means that you are creating a function with name f which takes certain arguments and executes the following statements
.

Single Input Single Output

Now create a function in R that will take a single input and gives us a single output.

Example-1:

# A simple R function to calculate

```
# area of a circle

areaOfCircle = function(radius){
                                                    area = pi*radius^2
                                                    print(area)
                                                  }
 areaOfCircle(2)
```

Following is an example to create a function that calculates the area of a circle which takes in the arguments the radius. So, to create a function, name the function as “areaOfCircle” and the arguments that are needed to be passed are the “radius” of the circle.

Example-2:

```
# A simple R function to check
# whether x is even or odd

evenOdd = function(x){
                                      if(x %% 2 == 0)
                                      print("even")
                                       else
                                      print("odd")
                                    }
evenOdd(4)
evenOdd(3)
```

Output:

```
[1] "even"

[1] "odd"
```

Multiple Input Multiple Output

Now create a function in R Language that will take multiple inputs and gives us multiple outputs using a list.

Example:

```
# A simple R function to calculate
# area of a rectangle

Rectangle = function(length, width){
                                      area = length * width
                                      print(area)
                                    }

 Rectangle(2, 3)
```

Output:

```
6
```

Graphical Representation of R programming

R language is mostly used for statistics and data analytics purposes to represent the data graphically in the software. To represent those data graphically, charts and graphs are used in R.

We can discuss here three type of graphical Representation.

- **Histogram**
- **Frequency polygon**
- **Ogive curve**

Histogram

Histogram is a graphical representation used to create a graph with bars representing the frequency of grouped data in vector. Histogram is same as bar chart but only difference between them is histogram represents frequency of grouped data rather than data itself.

We can create **histogram in R Programming Language using hist() function.**

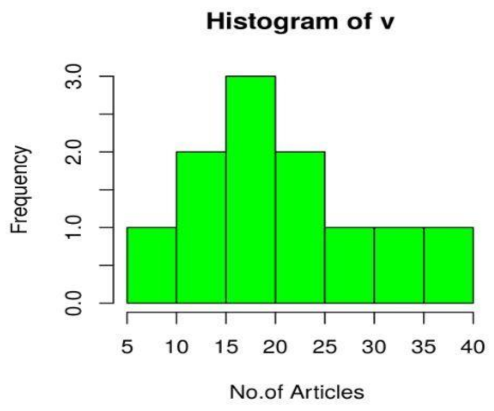
Syntax: *hist(v, main, xlab, xlim, ylim, breaks, col, border)*
Parameters:

- ***v:*** *This parameter contains numerical values used in histogram.*
- ***main:*** *This parameter main is the title of the chart.*
- ***col:*** *This parameter is used to set color of the bars.*
- ***xlab:*** *This parameter is the label for horizontal axis.*
- ***border:*** *This parameter is used to set border color of each bar.*
- ***xlim:*** *This parameter is used for plotting values of x-axis.*
- ***ylim:*** *This parameter is used for plotting values of y-axis.*
- ***breaks:*** *This parameter is used as width of each bar.*

Example-1:

```
# Create data for the graph.
v <- c(19, 23, 11, 5, 16, 21, 32,
       14, 19, 27, 39)

# Create the histogram.
hist(v, xlab = "No.of Articles ",
     col = "green", border = "black")
```

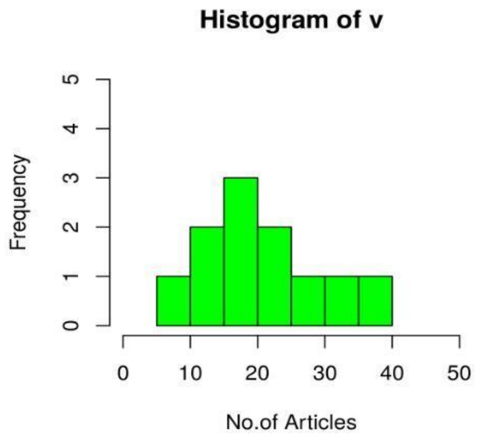
Output:



Example-2:

```
# Create data for the graph.
v <- c(19, 23, 11, 5, 16, 21, 32, 14, 19, 27, 39)

# Create the histogram.
hist(v, xlab = "No.of Articles", col = "green",
    border = "black", xlim = c(0, 50),
    ylim = c(0, 5), breaks = 5)
```

Output:



Frequency polygon

Frequency polygons are the plots of the values in a data frame to visualize the shape of the distribution of the values. It helps us in comparing different data frames and visualizing the cumulative frequency distribution of the data frames. The frequency polygon indicates the number of occurrences for each distinct class in the data frame.

To create a basic frequency polygon in the R Language, we first create a line plot for the variables under construction. Then we use the polygon() function to create the frequency polygon.

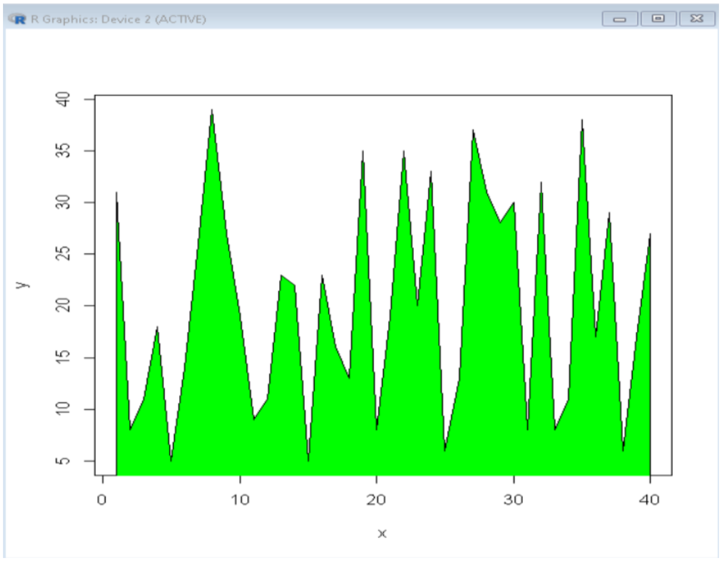
Syntax: *plot(x, y) polygon(c(xmin, x, xmax), c(ymin, y, ymax), col)*
where,

x and y: *determines the data vector for x and y axes data.*
xmin and ymin: *determines the minimum limit of x and y axis.*
xmax and ymax: *determines the maximum limit of x and y axis.*
col: *determines the color of frequency polygon.*

Example-1:

```
x<-1:40

y<-sample(5:40,40,replace=TRUE)
plot(x,y,type="l")
polygon(c(1,x,40),c(0,y,0),col="green")
```

output:



Digrametic Representation of R programming

There are hundreds of charts and graphs present in R. For example, bar plot, box plot, mosaic plot, dot chart, histogram, pie chart, scatter graph, etc.

We can discuss here following type of digramatical Representation.

- Simple Bar Diagram
- subdivided bar diagram
- pie diagram

Bar Plot or Bar Chart

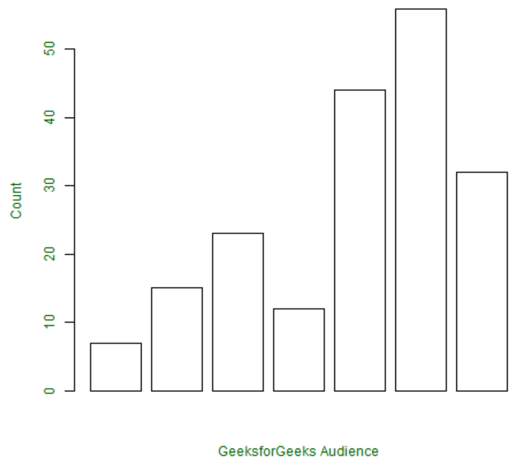
Bar plot or Bar Chart in R is used to represent the values in data vector as height of the bars. The data vector passed to the function is represented over y-axis of the graph. Bar chart can behave like histogram by using **table()** function instead of data vector.
Note: To know about more optional parameters in **barplot()** function, use the below command in R console:

Syntax: *barplot(data, xlab, ylab)*
where:
data *is the data vector to be represented on y-axis*
xlab *is the label given to x-axis*
ylab *is the label given to y-axis*

Example-1:

```
x <- c(7, 15, 23, 12, 44, 56, 32)

# plotting vector
barplot(x,xlab = "GeeksforGeeks Audience",
        ylab = "Count", col = "white",
        col.axis = "darkgreen",
        col.lab = "darkgreen")
```

output:



Pie Diagram or Pie Chart

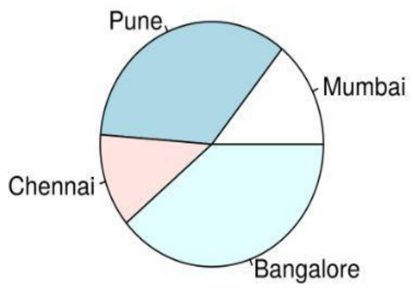
Pie chart is a circular chart divided into different segments according to the ratio of data provided. The total value of the pie is 100 and the segments tell the fraction of the whole pie. It is another method to represent statistical data in graphical form and **pie()** function is used to perform the same.
Note: To know about more optional parameters in **pie()** function, use the below command in the R console:

Syntax: *pie(x, labels, col, main, radius)*
where,
x *is data vector*
labels *shows names given to slices*
col *fills the color in the slices as given parameter*
main *shows title name of the pie chart*
radius *indicates radius of the pie chart. It can be between -1 to +1*

Example-1:

```
# Create data for the graph.
geeks<- c(23, 56, 20, 63)
labels <- c("Mumbai", "Pune", "Chennai", "Bangalore")

# Plot the chart.
pie(geeks, labels)
```

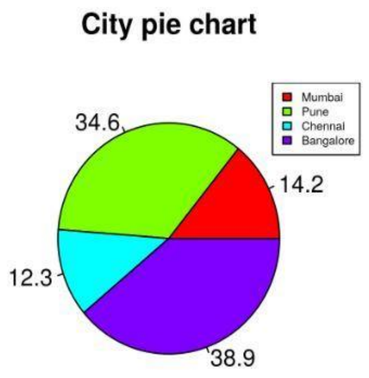
output:



Example-2:

```
geeks <- c(23, 56, 20, 63)

labels <- c("Mumbai", "Pune", "Chennai", "Bangalore")

piepercent<- round(100 * geeks / sum(geeks), 1)

# Plot the chart.

pie(geeks, labels = piepercent,

    main = "City pie chart", col = rainbow(length(geeks)))

legend("topright", c("Mumbai", "Pune", "Chennai", "Bangalore"),

                    cex = 0.5, fill = rainbow(length(geeks)))
```

output:



Programmes:

1)Finding Area of circle:

```
 radius=4

pi=3.14

area = pi*radius^2

area
```

2) To check whether the given integer is positive or negative:

```
a = as.integer(readline(prompt = "Enter the number :"))

if(a > 0)

  {

  print(paste("The number", a ,"is positive"))

    }else

    {

      print(paste("The number", a ,"is negative"))

    }
```

3) Reverse a given number:

```
n = as.integer(readline(prompt = "Enter given number :"))

rev = 0

while (n > 0) {

  r = n %% 10

  rev = rev * 10 + r

  n = n %/% 10

}

print(paste("Reverse number is :", rev))
```

4)To find greatest of three numbers:

```
x <- as.integer(readline(prompt = "Enter first number :"))

y <- as.integer(readline(prompt = "Enter second number :"))

z <- as.integer(readline(prompt = "Enter third number :"))

if (x > y && x > z)

 {

    print(paste("Greatest is :", x))

 } else

   if (y > z)

    {

      print(paste("Greatest is :", y))

    } else

      {

        print(paste("Greatest is :", z))

      }

 }
```

5)Find Prime numbers in a given range:

Individual number

```
n = as.integer(readline(prompt = "Enter a number :"))
  f = 1
  i = 2
  while (i <= n / 2) {
    if (n %% i == 0) {
      f = 0

      break
    }
    i = i + 1
  }

  if (f == 1) {
    print(paste("Number is prime :", n))
  } else{
    print(paste("Number is not prime :", n))

  }
```

Given Range

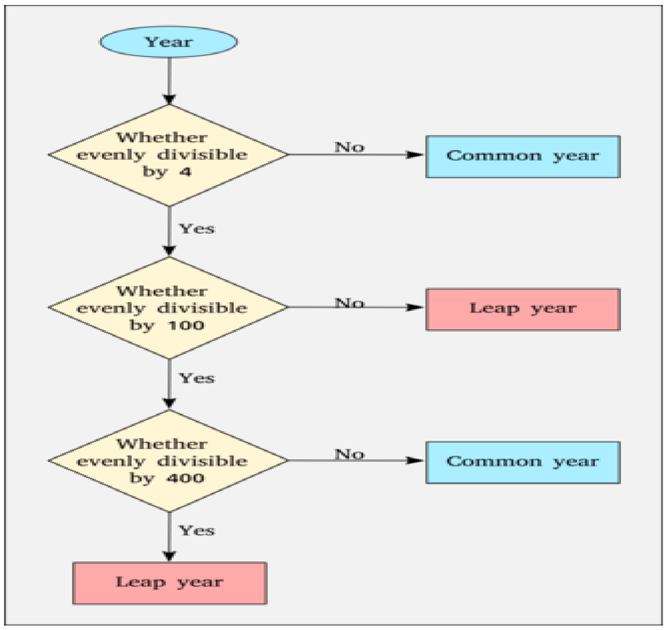
```
n = as.integer(readline(prompt = "Enter a number :"))
for (j in 2:n)
 {
  f = 1
  i = 2
  n = j
  while (i <= n / 2)
    {
      if (n %% i == 0)
        {
          f = 0
          break
        }
        i = i + 1
    }
    if (f == 1)
      {
         print(paste("Number is prime :", n))
      }
  }
```

6)To check if number is odd or even:

```
a = as.integer(readline(prompt = "Enter a number :"))
a
 if(a %% 2 == 0)
  {
    print(paste("The number", a ,"is even"))
  }else
   {
       print(paste("The number", a ,"is odd"))
   }
```

7)To check leap year:

```
 year = as.integer(readline(prompt="Enter a year: "))
if((year %% 4) == 0)
 {
   if((year %% 100) == 0)
    {
       if((year %% 400) == 0)
        {
           print(paste(year,"is a leap year"))
         } else
          {
             print(paste(year,"is not a leap year"))
          }
     } else
        {
           print(paste(year,"is a leap year"))
        }
 } else
   {
      print(paste(year,"is not a leap year"))
   }
```


Year
Whether evenly divisible by 4
No
Common year
Yes
Whether evenly divisible by 100
No
Leap year
Yes
Whether evenly divisible by 400
No
Common year
Yes
Leap year

8) To find sum of first n natural numbers:

```
num = as.integer(readline(prompt = "Enter a number: "))

if(num < 0) {

  print("Enter a positive number")

} else {

sum = 0

# use while loop to iterate until zero

while(num > 0) {

sum = sum + num

num = num - 1

}

print(paste("The sum is", sum))

}
```

9) To find AM, GM, and HM for ungrouped data:

Example-1:

Monthly sales of 10 small shops are given below

100,190, 210, 160, 150, 160, 190, 200, 170, 152

Calculate A.M. , G.M., H.M. of the above data and also calculate median, mode and quartiles.

Sol:

```
x=c(100,190, 210, 160, 150, 160, 190, 200, 170, 152)

n=length(x)

am=mean(x)

lx=log10(x)

gm=10^mean(lx)

hm=n/sum(1/x)

tx=table(x); m=which(tx==max(tx)); stx=sort(unique(x)); mo=stx[m]

me=median(x)

q1=quantile(x,0.25); q2=quantile(x,0.50); q3=quantile(x,0.75)
```

Example-2:

For the following frequency distribution

x: 1   2   3   4   5

f: 7   11   9   8   3

Calculate A.M.,G.M. and H.M.

sol:

x=1:5

f=c(7, 11, 9, 8, 3)

n=sum(f)

y=rep(x,f)

am=mean(y)

ly=log10(y)

gm=10^mean(ly)

hm=n/sum(f/x)

10) To find Mean deviation, Variance, Standard deviation for ungrouped data:

Example-1:

The number of mistakes in a page recorded for 20 pages are as follows.

2, 5, 9, 7, 11, 6, 5, 2, 7, 9, 3, 2, 8, 12, 14, 6, 3, 9, 8, 7

Calculate find mean deviation about mean, variance and standard deviation.

Sol:

X=c(, 5, 9, 7, 11, 6, 5, 2, 7, 9, 3, 2, 8, 12, 14, 6, 3, 9, 8, 7)

n=length(x)

mx=mean(x)

md=sum(abs(x-mx))/n

v1=var(x)

v=((n-1)/n)*v1

sd=sqrt(v)

cv=sd*100/abs(mx)

Example-2:

Calculate mean deviation about median, variance, standard deviation and also calculate quartile deviation and its coefficient.

Match score:	0	1	2	3	4
No. of matches:	27	9	8	5	4

Sol:

```
x=1:4
f=c(27,9,8,5,4)
n=sum(f)
y=rep(x,f)
mx=sum(f*x)/n
q1=quantile(y,0.25); q2=quantile(y,0.50); q3=quantile(y,0.75)
md=sum(f*abs(x-q2))/n
v=sum(f*(x-mx)^2)/n
sd=sqrt(v)
qd=(q3-q1)/2
cqd=(q3-q1)/(q3+q1)
```