UNIT - IV Conditionals and Control Flow: Relational Operators, Relational Operators and Vectors, Logical Operators, Logical Operators and Vectors, Conditional Statements. Iterative Programming in R: Introduction, While Loop, For Loop, Looping Over List. Functions in R: Introduction, writing a Function in R, Nested Functions, Function Scoping, Recursion, Loading an R Package, Mathematical Functions in R.

- In **computer programming**, an operator is a symbol which represents an action. An operator is a symbol which tells the compiler to perform specific **logical** or **mathematical** manipulations. R programming is very rich in built-in operators.
- In **R programming**, there are different types of operator, and each operator performs a different task. For data manipulation, There are some advance operators also such as model formula and list indexing.

There are the following types of operators used in R:

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

R Arithmetic Operators

These operators are used to carry out mathematical operations like addition and multiplication. Here is a list of arithmetic operators available in R.

Arithmetic Operators in R	
Operator	Description
+	Addition
–	Subtraction
*	Multiplication
/	Division
^	Exponent
%%	Modulus (Remainder from division)

%/%	Integer Division

Note: **cat converts its arguments to character strings, concatenates them, separating them by the given sep= string, and then prints them.**

.

```
# R program to illustrate
# the use of Arithmetic operators
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 ("Power operator :", vec1 ^ vec2)
```

Output:

Addition of vectors : 2 5

Subtraction of vectors : -2 -1

Multiplication of vectors : 0 6

Division of vectors : 0 0.6666667

Power operator : 0 8

Logical Operators

- 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 a 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 is 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)

print(!list1)

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 is True.

Input : list1 <- c(TRUE, 0.1)

list2 <- c(0,4+3i)

print(list1||list2)

Output : TRUE

Relational Operators

R Relational Operators

- Relational operators are used to compare between values. Here is a list of relational operators available in R.

Relational Operators in R	
Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

==	Equal to
!=	Not equal to

EX:

```
> 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
```

Operation on Vectors

- The above mentioned operators work on vectors. The variables used above were in fact single element vectors.
- We can use the function c() (as in concatenate) to make vectors in R.

All operations are carried out in element-wise fashion. Here is an example.

```
> x <- c(2,8,3)

> y <- c(6,4,1)

> x+y

[1]  8 12  4

> x>y

[1] FALSE  TRUE  TRUE
```

- When there is a mismatch in length (number of elements) of operand vectors, the elements in shorter one is recycled in a cyclic manner to match the length of the longer one.
- R will issue a warning if the length of the longer vector is not an integral multiple of the shorter vector.

```
> x <- c(2,1,8,3)

> y <- c(9,4)

> x+y # Element of y is recycled to 9,4,9,4

[1] 11  5 17  7

> x-1 # Scalar 1 is recycled to 1,1,1,1

[1] 1 0 7 2

> x+c(1,2,3)

[1]  3  3 11  4

Warning message:

In x + c(1, 2, 3) :

longer object length is not a multiple of shorter object length
```

R Logical Operators

- Logical operators are used to carry out Boolean operations like AND, OR etc.

Logical Operators in R	
Operator	Description
!	Logical NOT
&	Element-wise logical AND
&&	Logical AND
|	Element-wise logical OR
||	Logical OR

Operators & and | perform element-wise operation producing result having length of the longer operand.

But && and || examines only the first element of the operands resulting into a single length logical vector.

Zero is considered FALSE and non-zero numbers are taken as TRUE.

```
> x <- c(TRUE,FALSE,0,6)

> y <- c(FALSE,TRUE,FALSE,TRUE)

> !x

[1] FALSE  TRUE  TRUE FALSE
```

```
> x&y

[1] FALSE FALSE FALSE  TRUE

> x&&y

[1] FALSE

> x|y

[1]  TRUE  TRUE FALSE  TRUE

> x||y

[1] TRUE
```

R Assignment Operators

- These operators are used to assign values to variables.

Assignment Operators in R	
Operator	Description
<-, <<-, =	Leftwards assignment

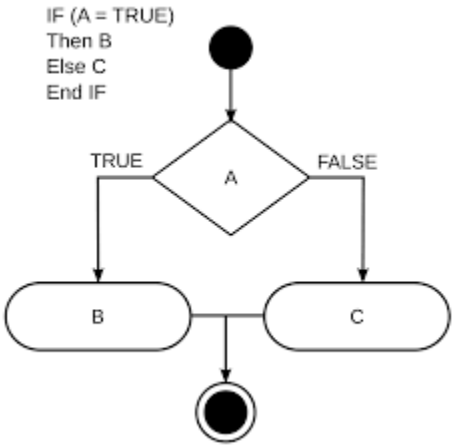
->, ->>	Rightwards assignment

- The operators <- and = can be used, almost interchangeably, to assign to variable in the same environment.
- The <<- operator is used for assigning to variables in the parent environments (more like global assignments). The rightward assignments, although available are rarely used.

```
> x <- 5

> x

[1] 5

> x = 9

> x

[1] 9

> 10 -> x

> x

[1] 10
```

Conditional Statements

- A conditional expression or a conditional statement is a programming construct where a decision is made to execute some code based on a boolean (true or false) condition.
- A more commonly used term for conditional expression in programming is an 'if-else' condition. In plain English, this is stated as 'if this test is true then do this operation; otherwise do this different operation'.



1. if Statement

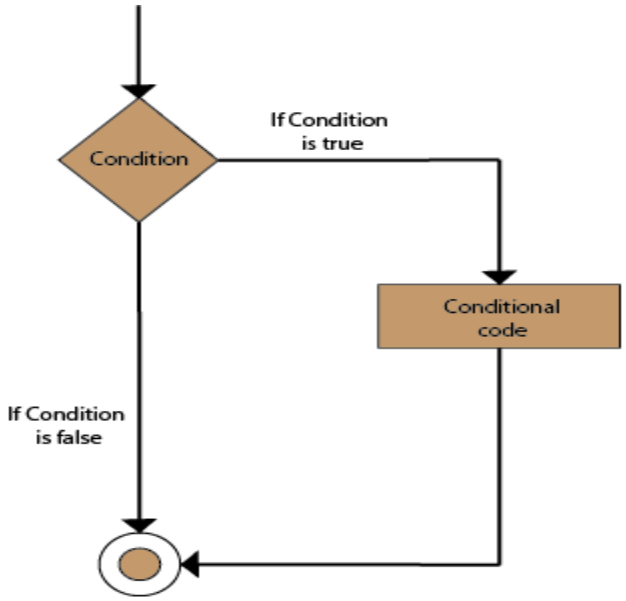
- The if statement consists of the Boolean expressions followed by one or more statements.
- The if statement is the simplest decision-making statement which helps us to take a decision on the basis of the condition.
- The if statement is a conditional programming statement which performs the function and displays the information if it is proved true.

- The block of code inside the if statement will be executed only when the boolean expression evaluates to be true.
- If the statement evaluates false, then the code which is mentioned after the condition will run.

The **syntax of if** statement in R is as follows:

```
1. if(boolean_expression) {
2.    // If the boolean expression is true, then statement(s) will be exe
   cuted.
3. }
```

Flow Chart



EX:1

```
1.       x <-24L
2. if(is.integer(x))
3. {
4.    print("x is an Integer")
5. }
```

Output

```
[1] "x is an Integer"
```

EX:2

```
x <- 100
if(x > 10){
print(paste(x, "is greater than 10"))
}
```

Output:

```
[1] "100 is greater than 10"
```

2.If-else statement

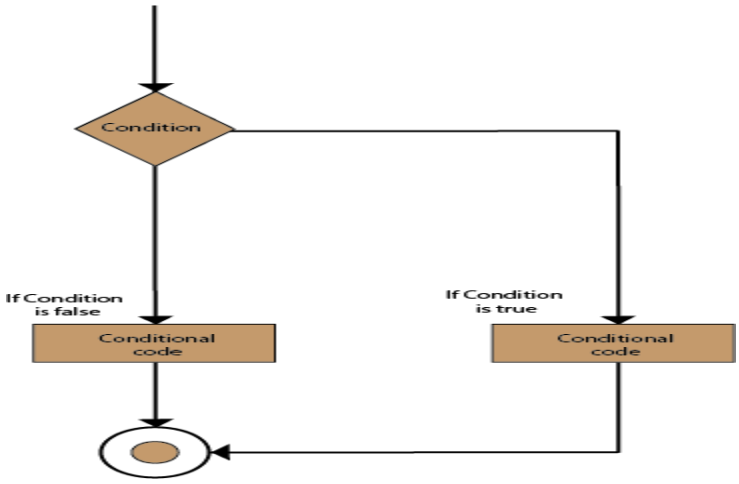
- In the if statement, the inner code is executed when the condition is true. The code which is outside the if block will be executed when the if condition is false.
- There is another type of decision-making statement known as the if-else statement. An if-else statement is the if statement followed by an else statement. An if-else statement, else

statement will be executed when the boolean expression will false. In simple words, If a Boolean expression will have true value, then the if block gets executed otherwise, the else block will get executed.

- R programming treats any non-zero and non-null values as true, and if the value is either zero or null, then it treats them as false.
- The basic syntax of If-else statement

```
1. if(boolean_expression) {
2.    // statement(s) will be executed if the boolean expression is true.

3. } else {
4.    // statement(s) will be executed if the boolean expression is false.

5. }
```

Flow Chart



NOTE: paste() will take multiple elements as inputs and concatenate those inputs into a single string**. The elements will be separated by a space as the default option.**

Example 1

```
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"

3. else if statement

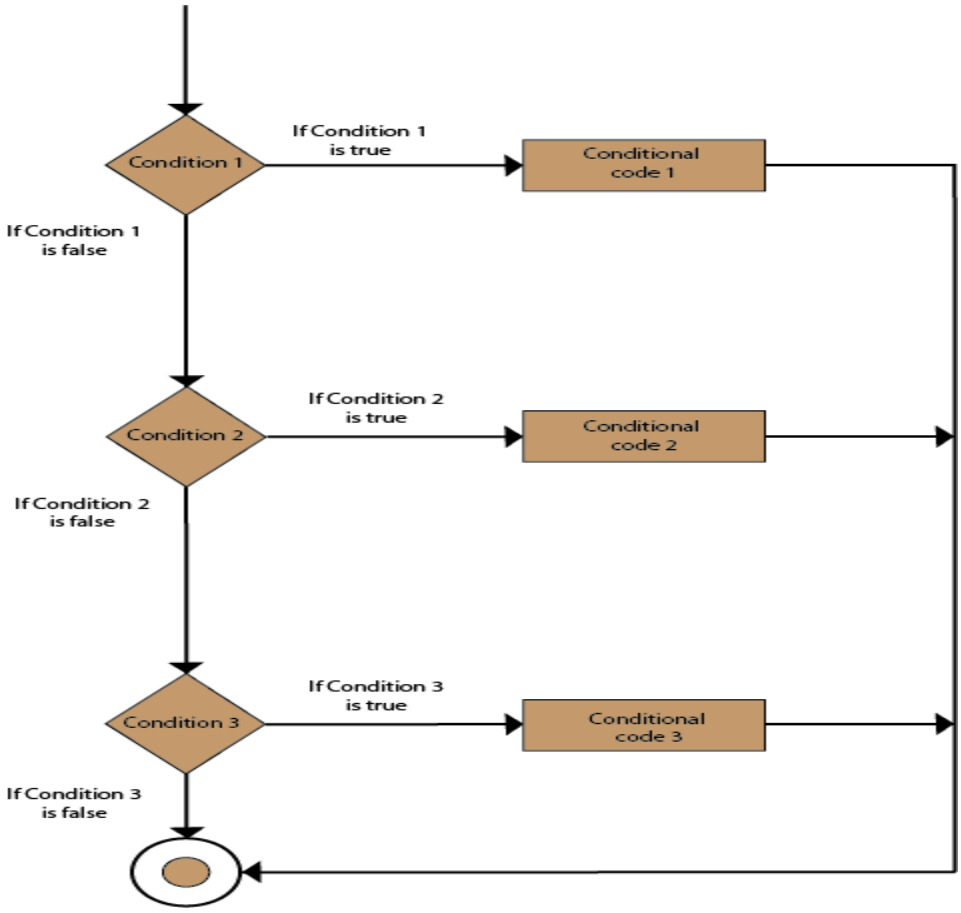
- This statement is also known as nested if-else statement.
- The if statement is followed by an optional else if..... else statement.
- This statement is used to test various condition in a single if.....else if statement.
- There are some key points which are necessary to keep in mind when we are using the if.....else if.....else statement. These points are as follows:

1. **if** statement can have either zero or one **else** statement and it must come after any **else if's** statement.
2. **if** statement can have many **else if's** statement and they come before the else statement.
3. Once an **else if** statement succeeds, none of the remaining **else if's** or **else's** will be tested.

The basic syntax of If-else statement is as follows:

```
1. if(boolean_expression 1) {
2.    // This block executes when the boolean expression 1 is true.
3. } else if( boolean_expression 2) {
4.    // This block executes when the boolean expression 2 is true.
5. } else if( boolean_expression 3) {
6.    // This block executes when the boolean expression 3 is true.
7. } else {
8.    // This block executes when none of the above condition is true.
    }
```

Flow Chart



Example 1

```
marks<-83;

if(marks>75){

    print("First class")

}else if(marks>65){

    print("Second class")

}else if(marks>55){

    print("Third class")

}else{

    print("Fail")
```

```
}
```

Output

```
[1] "First class"
```

EX:2

```
quantity <-  10
# Create multiple condition statement
if (quantity <20) {
      print('Not enough for today')
} else if (quantity > 20  &quantity <= 30) {
     print('Average day')
} else {
      print('What a great day!')
}
```

Output

```
[1] "Not enough for today"
```

Ex: 3

VAT has different rate according to the product purchased. Imagine we have three different kind of products with different VAT applied:

Categories	Products	VAT
A	Book, magazine, newspaper, etc..	8%
B	Vegetable, meat, beverage, etc..	10%
C	Tee-shirt, jean, pant, etc..	20%

We can write a chain to apply the correct VAT rate to the product a customer bought.

```
category <- 'A'

price <- 10

if (category =='A'){

  cat('A vat rate of 8% is applied.','The total price is',price *1.08)

} else if (category =='B'){

   cat('B vat rate of 10% is applied.','The total price is',price *1.10)

} else {

   cat('C vat rate of 20% is applied.','The total price is',price *1.20)

}
```

Output

```
A vat rate of 8% is applied. The total price is 10.8
```

Iterative Programming in R

- 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.
- Loops are used to repeat the process until the expression (condition) is TRUE. R uses three keywords **for, while and repeat** for looping purpose. **Next and break**, provide additional control over the loop.
- The break statement exits the control from the innermost loop.
- The next statement immediately transfers control to return to the start of the loop and statement after next is skipped.
- The value returned by a loop statement is always NULL and is returned invisibly.

There are three types of loop in R programming:

- For Loop
- While Loop
- Repeat Loop

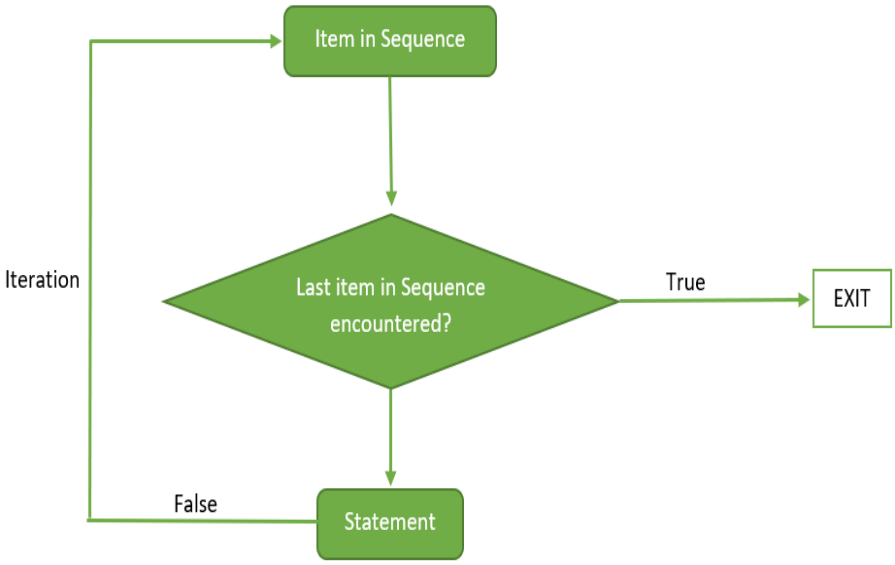
1. For Loop in R

- It is a type of control statement that enables one to easily construct a loop that has to run statements or a set of statements multiple times.
- For loop is commonly used to iterate over items of a sequence. 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.

R – For loop Syntax:

```
for (value in sequence)
{
  statement
}
```

For Loop Flow Diagram:



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

R program to demonstrate the use of for loop

```
 # using for loop

for (val in 1: 5)

{

    # statement

    print(val)

}
```

Output:

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

Example 2: Program to display days of a week.

```
# R program to illustrate

# application of for loop

 # assigning strings to the vector

week < - c('Sunday',  'Monday', Tuesday',
```

```
        'Wednesday', 'Thursday',
        'Friday', 'Saturday')
 # using for loop to iterate
# over each string in the vector
for (day in week)
{
    # displaying each string in the vector
    print(day)
}
```

Output:

```
1] "Sunday"
[1] "Monday"
[1] "Tuesday"
[1] "Wednesday"
[1] "Thusrday"
[1] "Friday"
[1] "Saturday"
```

2.repeat statement

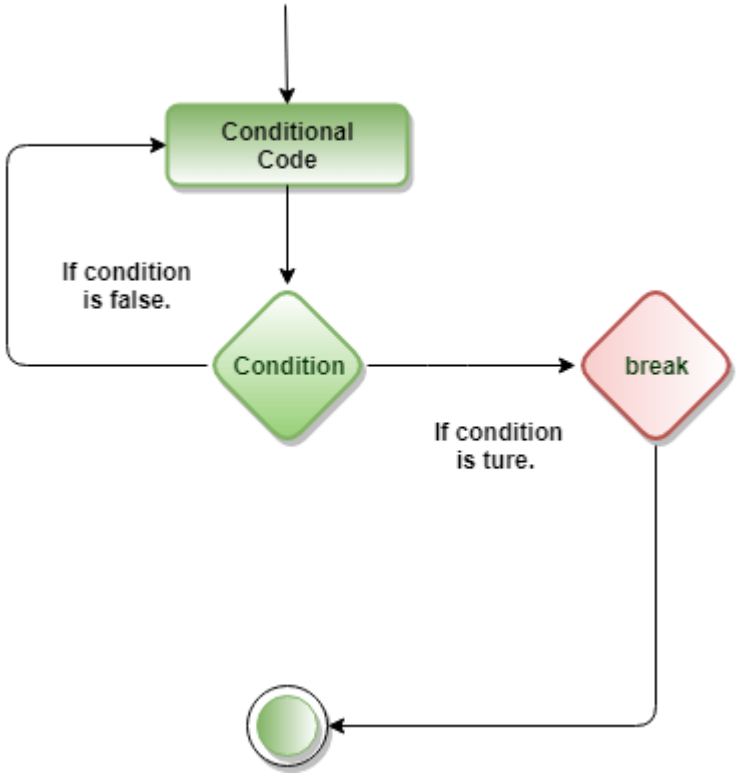
- The repeat statement executes the body of the loop until a break occurs. Be careful while using it because of the infinite nature of the loop.
- We must use the **break statement** to terminate the loop. The syntax of the repeat loop is :

For exiting, we include a **break statement** with a user-defined condition. This property of the loop makes it different from the other loops.

A repeat loop constructs with the help of the repeat keyword in R. It is very easy to construct an infinite loop in R.

The basic syntax of the repeat loop is as follows:

```
1. repeat {
2.    commands
3.    if(condition) {
4.       break
5.    }
6. }
```



Flowchart

1. First, we have to initialize our variables than it will enter into the Repeat loop.
2. This loop will execute the group of statements inside the loop.
3. After that, we have to use any expression inside the loop to exit.
4. It will check for the condition. It will execute a break statement to exit from the loop
5. If the condition is true.
6. The statements inside the repeat loop will be executed again if the condition is false.

Example 1:

```
1. v <- c("Hello","repeat","loop")
2. cnt <- 2
3. repeat {
4.    print(v)
5.    cnt <- cnt+1
6.         if(cnt > 4) {
7.       break
8.    }
9.}
```

Output

```
[1] "Hello"  "repeat" "loop"
[1] "Hello"  "repeat" "loop"
[1] "Hello"  "repeat" "loop"
```

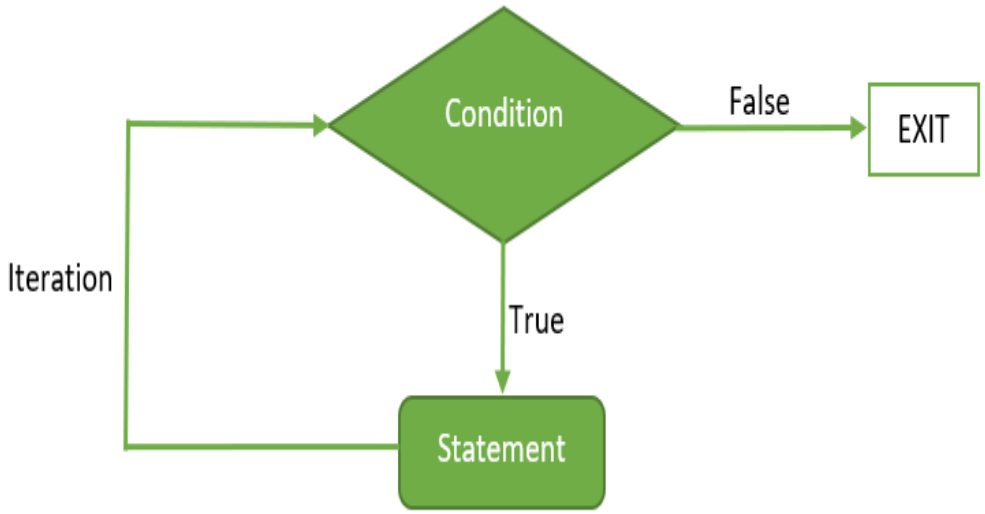
3.While Loop

- 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:



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.

```
# R program to illustrate
# application of while loop
 # assigning value to the variable
# whose factorial will be calculated
n <- 5
 # assigning the factorial variable
# and iteration variable to 1
factorial <- 1
i <- 1
 # using while loop
while (i <= n)
{
```

```
    # multiplying the factorial variable
    # with the iteration variable
    factorial = factorial * i
    # incrementing the iteration variable
    i = i + 1
}
# displaying the factorial
print(factorial)
```

Output:

```
[1] 120
```

- Here, at first, the variable n is assigned to 5 whose factorial is going to be calculated, then variable i and factorial are assigned to 1. i will be used for iterating over the loop, and factorial will be used for calculating the factorial.
- In each iteration of the loop, the condition is checked i.e. i should be less than or equal to 5, and after that factorial is multiplied with the value of i, then i is incremented.
- When i becomes 5, the loop is terminated and the factorial of 5 i.e. 120 is displayed beyond the scope of the loop.

4.Jump Statements in Loop

- We use a jump statement 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:** The break keyword is a jump statement that is used to terminate the loop at a particular iteration.

Example:

```
# R program to illustrate
# the use of break statement
 # using for loop
# to iterate over a sequence
for (val in 1: 5)
{
    # checking condition
    if (val == 3)
    {
        # using break keyword
```

```
        break
    }
     # displaying items in the sequence
    print(val)
}
```

Output:

```
[1] 1
[1] 2
```

In the above program, if the value of val becomes 3 then the break statement will be executed and the loop will terminate.

Next Statement: The next keyword is a jump statement which is used to skip a particular iteration in the loop.

Example:

```
# R program to illustrate
# the use of next statement
 # using for loop
# to iterate over the sequence
for (val in 1: 5)
{
```

```
    # checking condition
    if (val == 3)
    {
        # using next keyword
        next
    }
     # displaying items in the sequence
    print(val)
}
```

Output:

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

- In the above program, if the value of Val becomes 3 then the next statement will be executed hence the current iteration of the loop will be skipped. So 3 is not displayed in the output.
- As we can conclude from the above two programs the basic difference between the two jump statements is that the **break** statement terminates the loop and the **next** statement skips a particular iteration of the loop.

Loop over a list

- Looping over a list is just as easy and convenient as looping over a vector. There are again two different approaches here:

```
primes_list <- list(2, 3, 5, 7, 11, 13)

# loop version 1
for (p in primes_list) {
  print(p)
}

# loop version 2
for (i in 1:length(primes_list)) {
  print(primes_list[[i]])
}
```

Notice that you need double square brackets - [[ ]] - to select the list elements in loop version 2.

Output

```
[1] 2
[1] 3
[1] 5
[1] 7
[1] 11
[1] 13
[1] 2
[1] 3
[1] 5
[1] 7
[1] 11
[1] 13
```

Functions in R Programming

- A set of statements which are organized together to perform a specific task is known as a function.
- R provides a series of in-built functions, and it allows the user to create their own functions.
- Functions are used to perform tasks in the modular approach.
- Functions are used to avoid repeating the same task and to reduce complexity.
- To understand and maintain our code, we logically break it into smaller parts using the function.
- A function should be

- Written to carry out a specified task.
- May or may not have arguments
- Contain a body in which our code is written.
- May or may not return one or more output values.

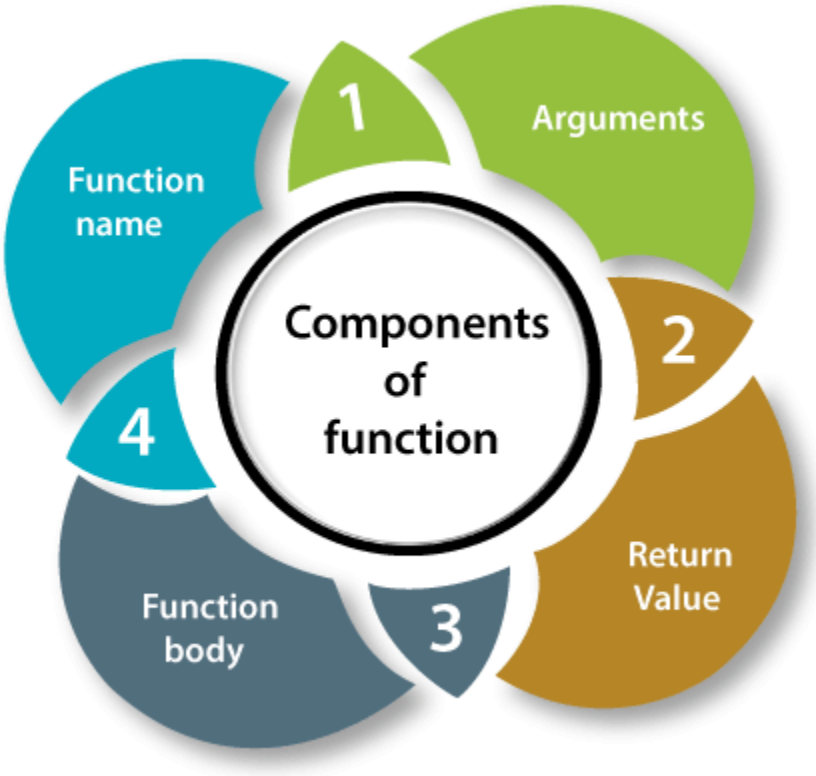
"**An R function is created by using the keyword function.**"

✓ There is the following syntax of R function:

```
1. func_name <- function(arg_1, arg_2, ...) {
2.     Function body
3. }
```

Components of Functions

There are four components of function, which are as follows:



Function Name

- ✓ The function name is the actual name of the function.
- ✓ In R, the function is stored as an object with its name.

Arguments

- ✓ In function, arguments are optional means a function may or may not contain arguments, and these arguments can have default values also.
- ✓ We pass a value to the argument when a function is invoked.

Function Body

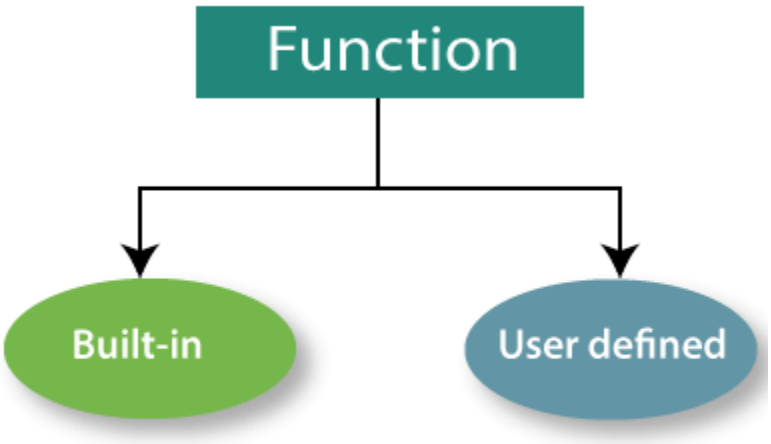
- ✓ The function body contains a set of statements which defines what the function does.

Return value

- ✓ It is the last expression in the function body which is to be evaluated.

Function Types

- ✓ Similar to the other languages, R also has two types of function, i.e. **Built-in Function** and **User-defined Function**.
- ✓ In R, there are lots of built-in functions which we can directly call in the program without defining them. R also allows us to create our own functions.



Built-in function

- ✓ The functions which are already created or defined in the programming framework are known as built-in functions.
- ✓ User doesn't need to create these types of functions, and these functions are built into an application.
- ✓ End-users can access these functions by simply calling it.
- ✓ R have different types of built-in functions such as seq(), mean(), max(), and sum(x) etc.

1.# Creating sequence of numbers from 32 to 46.

2.print(seq(32,46))

3.# Finding the mean of numbers from 22 to 80.

4.print(mean(22:80))

5.# Finding the sum of numbers from 41 to 70.

6.print(sum(41:70))

Output

```
 [1] 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
[1] 51
[1] 1665
```

EX: 2

sequence of number from 44 to 55 both including incremented by 1

```
x_vector <- seq(45,55, by = 1)
#logarithm
log(x_vector)
```

Output:

```
##  [1] 3.806662 3.828641 3.850148 3.871201 3.891820 3.912023 3.931826
##  [8] 3.951244 3.970292 3.988984 4.007333
```

Math functions

R has an array of mathematical functions.

Operator	Description
abs(x)	Takes the absolute value of x
log(x,base=y)	Takes the logarithm of x with base y; if base is not specified, returns the natural logarithm
exp(x)	Returns the exponential of x

sqrt(x)	Returns the square root of x
factorial(x)	Returns the factorial of x (x!)

Basic statistic functions

Operator	**Description**
mean(x)	Mean of x
median(x)	Median of x
var(x)	Variance of x
sd(x)	Standard deviation of x
quantile(x)	The quartiles of x
summary(x)	Summary of x: mean, min, max etc..

EX: 1

```
speed <- dt$speed
speed
# Mean speed of cars dataset
mean(speed)
```

User-defined function

- ✓ R allows us to create our own function in our program.
- ✓ A user defines a user-define function to fulfill the requirement of user.
- ✓ Once these functions are created, we can use these functions like in-built function.

Call a Function

- ✓ To call a function, use the function name followed by parenthesis, like **my_function()**:

Example

```
my_function <- function()

{
  print("Hello World!")
}

my_function() # call the function named my_function
```

O/P:

```
[1] "Hello World!"
```

Arguments

- ✓ Information can be passed into functions as arguments.
- ✓ Arguments are specified after the function name, inside the parentheses.
- ✓ You can add as many arguments as you want, just separate them with a comma.
- ✓ The following example has a function with one argument (fname).
- ✓ When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
my_function <- function(fname)
{
  paste(fname, "Griffin")
}
my_function("Peter")
my_function("Lois")
my_function("Stewie")
```

O/P;

```
[1] "Peter Griffin"
[1] "Lois Griffin"
[1] "Stewie Griffin"
```

Return Values

✓ To let a function return a result, use the return() function:

Example

```
my_function <- function(x) {
  return (5 * x)
}
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

```
[1] 15
[1] 25
[1] 45
```

EX:

```
# A simple R function to check
# whether x is even or odd
 evenOdd <- function(x){
  if(x %% 2 == 0)
    return("even")
  else
    return("odd")
}
 print(evenOdd(4))
print(evenOdd(3))
```

Output:

[1] "even"

[1] "odd"

Single Input Single Output

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

```
# A simple R function to calculate

# area of a circle

 areaOfCircle = function(radius){

  area = pi*radius^2

  return(area)

}

 print(areaOfCircle(2))
```

Output:

12.56637

Nested Functions

- ✓ A **nested function** or the **enclosing function** is a function that is defined within another function. In simpler words, a nested function is a function in another function.

- ✓ There are two ways to create a nested function in the R programming language:

1. **Calling a function within another function we created.**
2. **Writing a function within another function**.

Calling a function within another function

- ✓ For this process, we have to create the function with two or more required parameters. After that, we can call the function that we created whenever we want to

```
# creating a function with two parameters
myFunction <- function(x, y) {
  # passing a command to the function
  a <- x * y
  return(a)
}
# creating a nested function
myFunction(myFunction(2,2), myFunction(3,3))
```

Code explanation

- **Line 2**: We creatd a function with two parameter values, x and y.

- **Line 4**: The function we created tells `x` to multiply `y` and assign the output to a variable `a`.
- **Line 5**: We return the value of `a`.
- **Line 9**: We make the first input `myFunction(2,2)` represent the primary function's `x` parameter. Likewise, we make the second input `myFunction(3,3)` represent the `y` parameter of the main function.

Hence, we get the following expected output: `(2 * 2) * (3 * 3) = 36`

Writing a function within another function

✓ In this process, we can't directly call the function because there is an inner function defined inside an outer function. Therefore, we will call the external function to call the function inside.

```
# creating an outer function
outerFunction <- function(x) {
  # creating an inner function
  innerFunction <- function(y) {
    # passing a command to the function
    a <- x * y
    return(a)
  }
  return (innerFunction)
}
```

```
# To call the outer functionn
output <- outerFunction(3)
output(5)
```

Code explanation

- **Line 2**: We create an outer function, outerFunction, and pass a parameter value x to the function.
- **Line 4**: We create an inner function, innerFunction, and pass a parameter value y to the function.
- **Line 6**: We pass a command to the inner function, which tells x to multiply y and assign the output to a variable a.
- **Line 7**: We return the value of a.
- **Line 9**: We return the innerFunction.
- **Line 12**: To call the outer function, we create a variable output and give it a value of 3.
- **Line 13**: We print the output with the desired value of 5 as the value of the y parameter.

Therefore, the output is: (3 * 5 = 15)

Scopes

✓ The scope of a variable is nothing more than the place in the code where it is referenced and visible.

- ✓ There are two basic concepts of scoping, ***lexical(static)* scoping and is *dynamic* scoping.**
- ✓ In R, there is a concept of ***free variables***, which add some spice to the scoping.
- ✓ R uses lexical scoping, which says the values of such variables are searched for in the **environment** in which the function was defined.

```
f <- function(a, b) {
2       (a * b) / z
3 }
```

- ✓ In this function, you have two formal arguments, a and b.
- ✓ You have another symbol, z, in the body of the function, which is a free variable.
- ✓ The scoping rules of the language define how value is assigned to free variables.
- ✓ R uses lexical scoping, which says the value for z is searched for in the environment where the function was defined.
- ✓ **'Z' is called free variable**

Note: Lexical scoping is also referred to as statical scoping.

- ✓ With **dynamic scoping**, the variable is bound to the most recent value assigned to that variable.
- ✓ R provides some escape routes to bypass the shortcomings of lexical scoping. The <- operator is called a variable assignment operator.

- ✓ **Given the expression a <- 3.14, the value is assigned to the variable in the current environment. If you already had an assignment for the variable before in the same environment, this one will overwrite it.**
- ✓ Variable assignments only update in the current environment, and they never create a new scope. When R is looking for a value of a given variable, it will start searching from the bottom.
- ✓ This means the **current environment(The top level environment available to us at the R command prompt is the global environment called R_GlobalEnv . Global environment can be referred to as . GlobalEnv in R codes as well. We can use the ls() function to show what variables and functions are defined in the current environment.)** is inspected first, then its enclosing environment(**The enclosing environment is the environment where the function was created. Every function has one and only one enclosing environment**). The search goes until either the value is found or the empty environment is reached.

```
a <- 3.14
add <- function(x,y)
{
x * y / a
```

```
}
add(10,11)
```

The output is the following:

```
1[1] 35.03185
```

- ✓ When the function is called, only the two arguments are passed. R tries to look up the a variable's value and first looks at the **scope of the function**.
- ✓ Since it cannot be found there, it look for the value in the enclosing scope, where it finally finds it. If you had not defined the a variable, it would give you the following error: Error in b(10, 11) : object 'a' not found, stating that the lookup has failed.
- ✓ This brings us to the concept of **environment**. Environments in R are basically mappings from variables to values.
- ✓ Every function has a local environment and a reference to the enclosing environment.
- ✓ This helps scoping and lookup. You have the option to add, remove, or modify variable mappings and can even change the reference to the enclosing environment.

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.	round(x, digits=n)	It returns round value of input x.	x<- -4 print(abs(x)) **Output** 4

7.	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
8.	log(x)	It returns natural logarithm of input x.	x<- 4 print(log(x)) **Output** [1] 1.386294
9.	log10(x)	It returns common logarithm of input x.	x<- 4 print(log10(x)) **Output** [1] 0.60206
10.	exp(x)	It returns exponent.	x<- 4 print(exp(x))

			Output [1] 54.59815

Recursive Functions in R Programming

- Recursion, in the simplest terms, is a type of looping technique. It exploits the basic working of functions in R.
- **Recursion is when the function calls itself**. **This forms a loop, where every time the function is called, it calls itself again and again and this technique is known as recursion.**
- Since the loops increase the memory we use the recursion. The recursive function uses the concept of recursion **to perform iterative tasks they call themselves, again and again, which acts as a loop.**
- These kinds of functions need a stopping condition so that they can stop looping continuously.
- **Recursive functions call themselves**. They break down the problem into smaller components.
- The function() calls itself within the original function() on each of the smaller components. After this, the results will be put together to solve the original problem.

Example: Factorial using Recursion in R

```
rec_fac <- function(x){
    if(x==0 || x==1)
    {
        return(1)
    }
    else
    {
        return(x*rec_fac(x-1))
    }
}
```

Output:

[1] 120

- Here, rec_fac(5) calls rec_fac(4), which then calls rec_fac(3), and so on until the input argument x, has reached 1.
- The function returns 1 and is destroyed. The return value is multiplied with argument value and returned.
- This process continues until the first function call returns its output, giving us the final result.

Key Features of R Recursion

- The use of recursion, often, makes the code shorter and it also looks clean.
- It is a simple solution for a few cases.
- It expresses in a function that calls itself.

Applications of Recursion in R

- Recursive functions are used in many efficient programming techniques like dynamic programming language(DSL) or divide and conquer algorithms.
- In dynamic programming, for both top-down as well as bottom-up approaches, recursion is vital for performance.
- In divide and conquer algorithms, we divide a problem into smaller sub-problems that are easier to solve. The output is then built back up to the top. Recursion has a similar process, which is why it is used to implement such algorithms.
- In its essence, recursion is the process of breaking down a problem into many smaller problems, these smaller problems are further broken down until the problem left is trivial. The solution is then built back up piece by piece.

Note: **we can define looping or iteration as the process where the same set of instructions is repeated multiple times in a single call. In contrast, we can enumerate recursion as the**

process where the output of one iteration from a function call becomes the input of the next in a separate function call

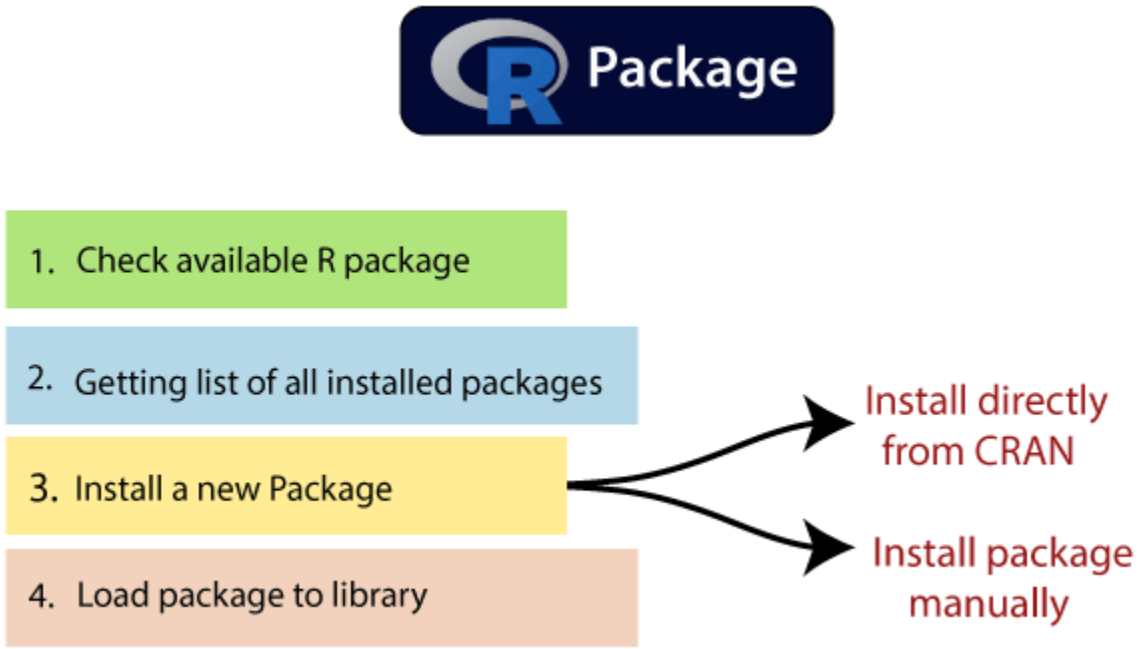
Packages in R

- Packages in R Programming language are a **set of R functions, compiled code, and sample data.**
- These are stored under a directory called "library" within the R environment. By default, R installs a group of packages during installation.
- Once we start the R console, only the default packages are available by default.
- Other packages that are already installed need to be loaded explicitly to be utilized by the R program that's getting to use them.

What are Repositories?

- A repository is a place where packages are located and stored so you can install packages from it.
- Organizations and Developers have a local repository, typically they are online and accessible to everyone.
- Some of the most popular repositories for R packages are:

- **CRAN: Comprehensive R Archive Network(CRAN)** is the official repository, it is a network of ftp and web servers maintained by the R community around the world.
- The R community coordinates it, and for a package to be published in CRAN, the Package needs to pass several tests to ensure that the package is following CRAN policies.
- **Bioconductor:** Bioconductor is a topic-specific repository, intended for open source software for bioinformatics. Similar to CRAN, it has its own submission and review processes, and its community is very active having several conferences and meetings per year in order to maintain quality.
- **Github:** Github is the most popular repository for open source projects. It's popular as it comes from the unlimited space for open source, the integration with git, a version control software, and its ease to share and collaborate with others.



1.Check Available R Packages

- To check the available R Packages, we have to find the library location in which R packages are contained. R provides libPaths() function to find the library locations.

```
libPaths()
```

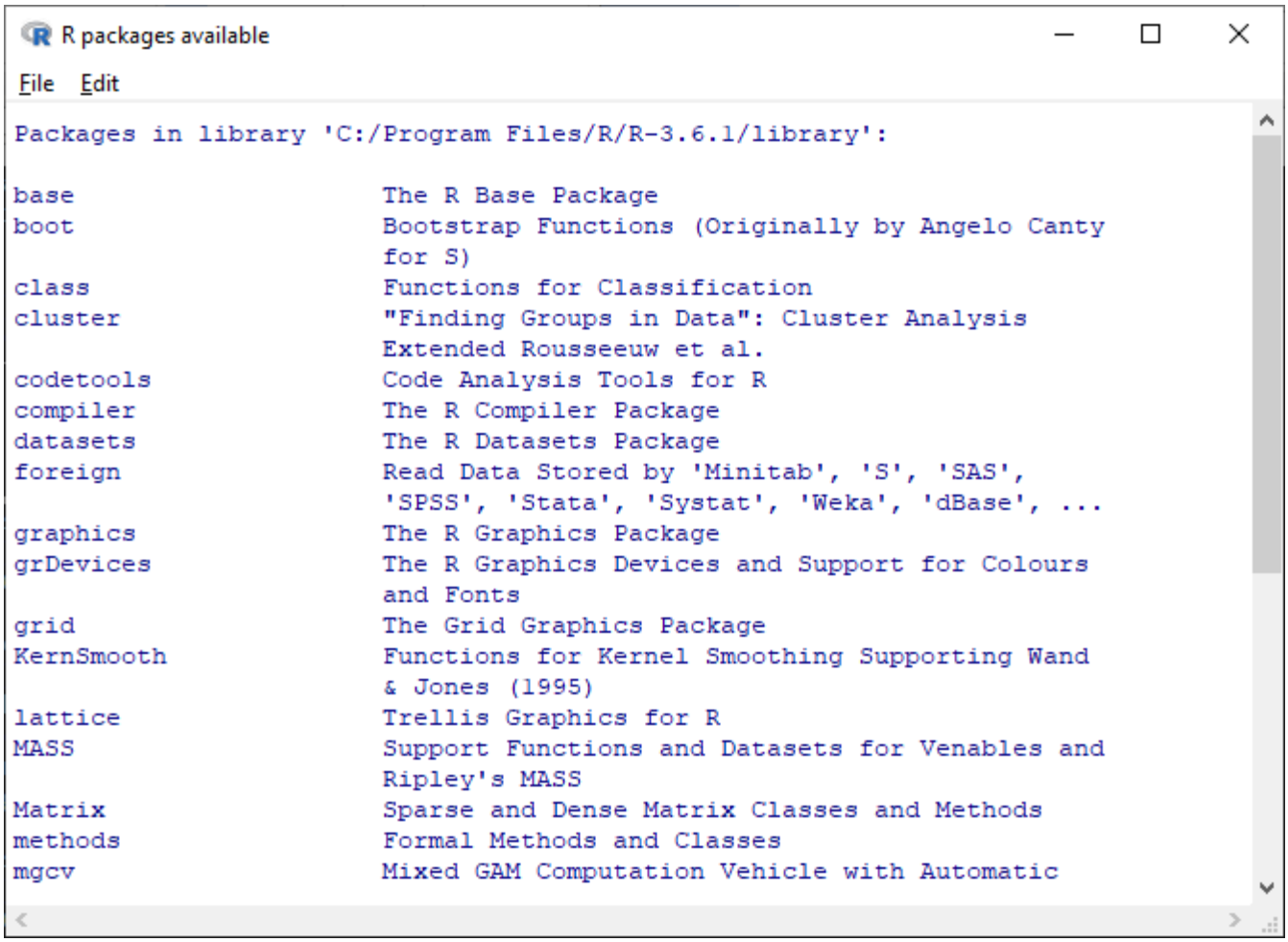
- When the above code executes, it produces the following project, which may vary depending on the local settings of our PCs & Laptops.

2.Getting the list of all the packages installed

- **R provides library() function, which allows us to get the list of all the installed packages.**
- **library()**

When we execute the above function, it produces the following result, which may vary depending on the local settings of our PCs or laptops.

Packages in library 'C:/Program Files/R/R-3.6.1/library':



Like library() function, R provides search() function to get all packages currently loaded in the R environment.

Install a New Package

- In R, there are two techniques to add new R packages. The first technique is installing package directly from the CRAN directory, and the second one is to install it manually after downloading the package to our local system.

Install directly from CRAN

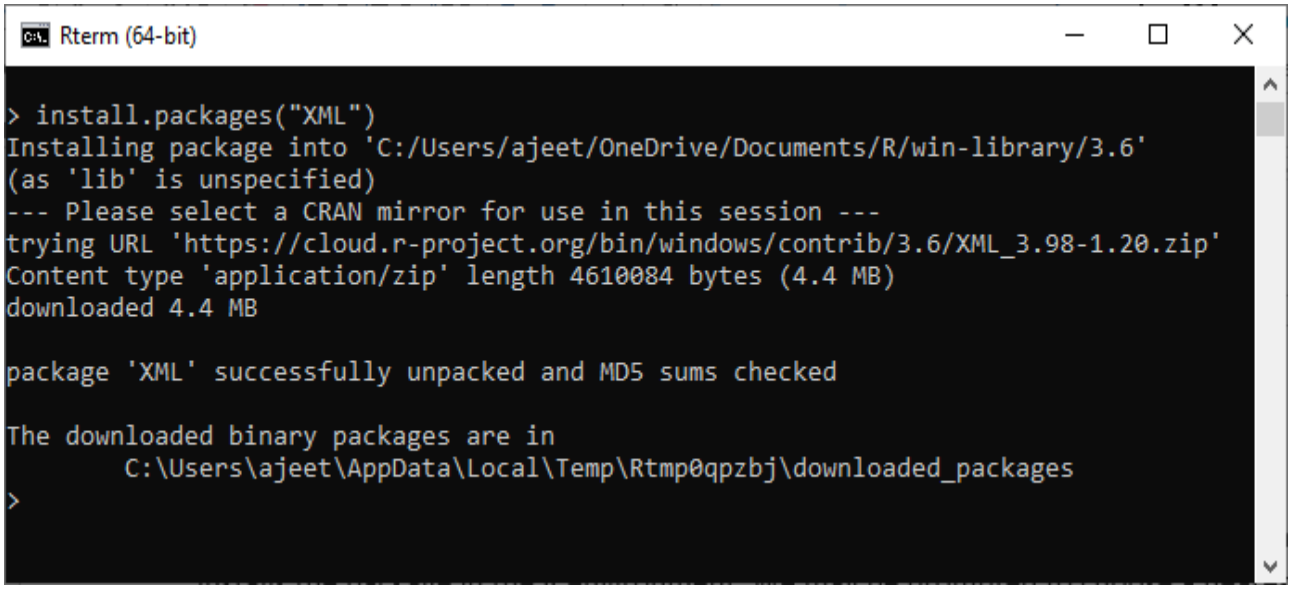
The following command is used to get the packages directly from CRAN webpage and install the package in the R environment. We may be prompted to choose the nearest mirror. Choose the one appropriate to our location.

1. install.packages("Package Name")

The syntax of installing XML package is as follows:

install.packages("XML")

Output

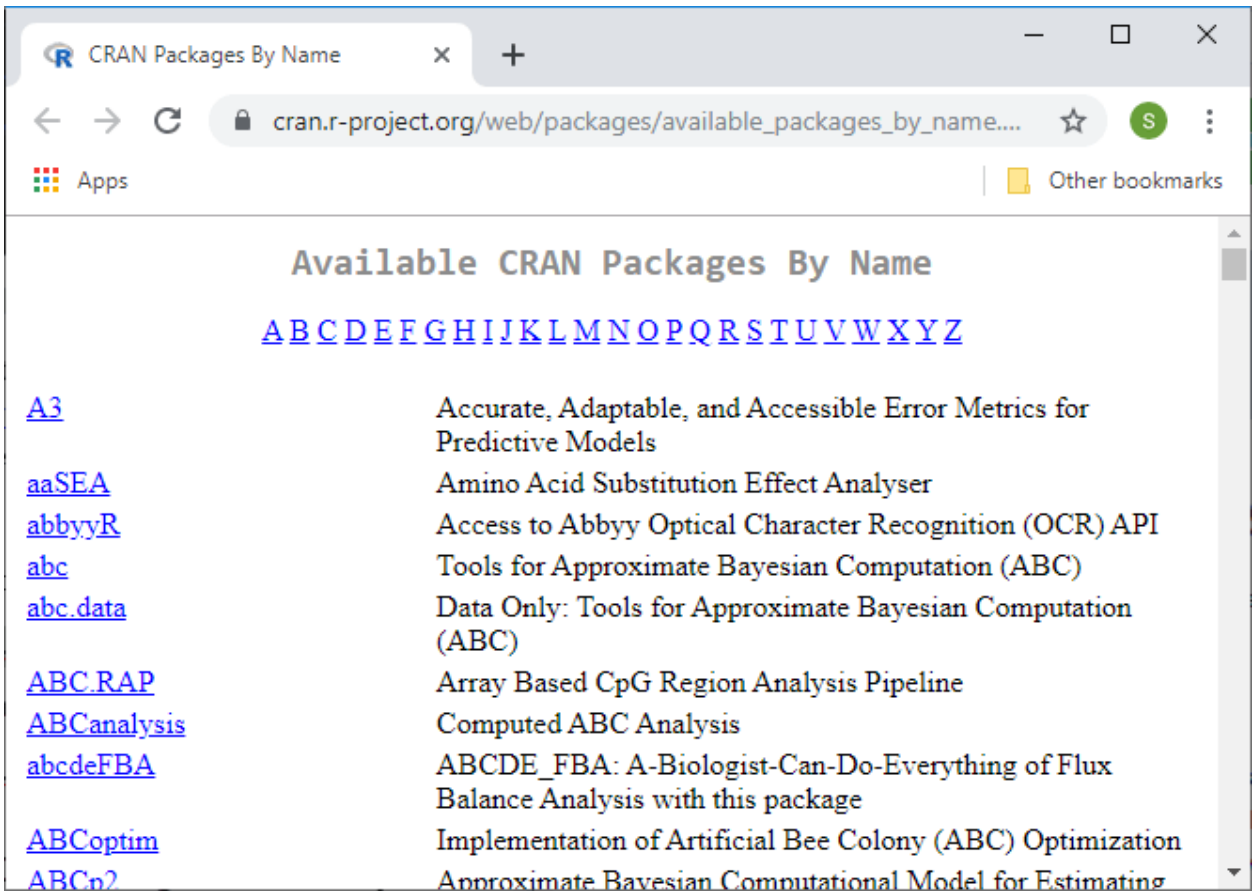


Install package manually

To install a package manually, we first have to download it from

https://cran.r-project.org/web/packages/available_packages_by_name.html

. The required package will be saved as a .zip file in a suitable location in the local system.



Once the downloading has finished, we will use the following command:

1. install.packages(file_name_with_path, repos = NULL, type = "source")

Install the package named "XML"

```
1. install.packages("C:\Users\ajeet\OneDrive\Desktop\graphics\xml2_1.2.2.zip", repos = NULL, type = "source")
```

Load Package to Library

- We cannot use the package in our code until it will not be loaded into the current R environment.
- We also need to load a package which is already installed previously but not available in the current environment.

There is the following command to load a package:

```
1. library("package Name", lib.loc = "path to library")
```

Command to load the XML package

```
1. install.packages("C:\Users\ajeet\OneDrive\Desktop\graphics\xml2_1.2.2.zip", repos = NULL, type = "source")
```