**1. Introduction to R**

## **1.1 History of Computer Programming**

*What do you mean by the term programming language?*

● To instruct the computer we need to speak in their language.

● Computers understand binary (Machine Language) but

speaking that is very difficult.

● So generation by generation people tried to go closer to english

and move away from technical details (Principle of abstraction).

|  |  |  |
| --- | --- | --- |
| **Year** | **Generation** | **Languages** |
| 1990’s | Fourth Generation(4GL) | R, Java, Python |
| 1970’s | Third Generation (3GL) | Fortran, Basic , C , C++ |
| 1960’s | Second Generation (2GL) | Assembly Language |
| 1950’s | First Generation (1GL) | Machine Language |

*Computer Programming Through Yearwise Generations*

## **1.2 Definition of R**

*Definition*

R is a general-purpose, multi-platform, object-oriented, interpreted programming language that is used for computational statistics, analysis and the hard sciences, for example, astronomy, chemistry, and genomics to practical applications in business, finance, health care, marketing, medicine etc.

● Created by Ross Ihaka and Robert Gentleman in 1992

● Current version of R is 3.6.2.

*Features of R:*

1. Comprehensive Language. ...
2. Provides the user a Large Array of Packages.
3. Possesses a Number of Graphical Libraries. ...
4. Open-source. ...
5. Cross-Platform Compatibility. ...
6. Facilities for Various Industries. ...
7. No Need for a Compiler. ...
8. Performs Fast Calculations.

### Can Handle all Sorting the Data

### Integration with various Technologies

### R provides the user with various Community

### R is also used in Machine Learning

# *Advantages and Disadvantages of R :*

R is the one of the most popular programming language for statistical model and data analysis. Like various other programming languages, It also has some pros and cons.There are the following pros and cons of R

*Advantages :*

1. Open Source
2. Data Wrangling
3. Array of packages
4. Quality Plotting and Graphing
5. Platform Independent
6. Machine Learning Operations
7. Continuously Growing

*Disadvantages :*

1. Weak Origin
2. Data Handling
3. Basic Security
4. Complicated Language
5. Lesser Speed

## **1.3 Installation of R and R Studio**

1. Open official R website<https://cran.r-project.org/> and click Download R for either windows/mac / linux as per your operating system.
2. The commonly used IDE (Integrated Development Environment) for R is RStudio. Download RStudio from the following website<https://rstudio.com/products/rstudio/download/> and click download RStudio for either windows/mac / Linux as per your operating system.

## **1.4 First Program**

Open RStudio -> File -> New File -> R Script and save it as **hello.R**.

|  |
| --- |
| **print(“Hello World”)**  **print(“Welcome to python”)** |

Select the entire code and click the run button or press Ctrl+Enter.

**2. Variables and Datatypes**

## **2.1 Variables**

*Definition:*

|  |
| --- |
| R is dynamically typed, which means that you don't have to declare what type each variable is. In R, variables are stored in a memory which allots placeholder for texts and numbers. |

1. Variables are references that can be used to refer to a particular data
2. <- or = characters are used to assign a variable
3. Variables changes their value
4. Variables are stored in RAM

*Examples of variables in real world:*

1. Temperature of Room
2. Age
3. Cricket Score
4. Marks in exams

*Declaring a variable in python:*

|  |
| --- |
| name = “itvedant” or name<-“itvedant”  age = 25 or age<-25  percentage = 98.5 or percentage<-98.5 |

*Rules for identifier:*

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

###### Variable must be started with the use of a letter or a period. If it starts with a period, it cannot be followed by a digit.

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

|  |  |  |
| --- | --- | --- |
| **Variable Name** | **Validity** | **Reason** |
| first\_name. | valid | Has letters, numbers, dot and underscore |
| first\_name% | Invalid | Has the character '%'. Only dot(.) and underscore allowed. |
| 2first\_name | invalid | Starts with a number |
| .last\_name,  address.var | valid | Can start with a dot(.) but the dot(.)should not be followed by a number. |
| .2pin\_code | invalid | The starting dot is followed by a number making it invalid. |
| \_pin\_code | invalid | Starts with \_ which is not valid |

## *Deleting Variables*

Variables can be deleted by using the rm() function. Below we can delete the variable a.3. On printing the value of the variable error is shown.

|  |
| --- |
| rm(a.3)  print(a.3)  When we execute the above code, it produces the following result −  [1]”a.3”  Error in print(a.3):object ‘a.3’ not found |

# **2.2 Data Types :**

*Data Types in R - Programming :*

1. R's basic data types are character, numeric, integer, complex, and logical, Raw.
2. There are many basic data structures in R which includes the matrix, list, vector, data frame, and factors, array.
3. R's basic data types :

* Computers store the data in memory using variables but the *amount of storage is decided by its datatype.*
* Unlike compiled languages in python, we do not specify the datatype for the variables
* R decides it intelligently and assigns one of the following data types:

1. *Numbers**(Integer / Decimal)*
2. *Character* *(Single / Double quotes)*
3. *Logical (True / False)*

# **2.3 Numbers(Numeric)**

Numeric data types are needed when you want to perform arithmetic operations like addition or multiplication on variables.

*For Example :*

*Example 1 :*

|  |
| --- |
| a = 10  print(a + 2 ) #prints 12  print(a-2) #prints 8 |

*Example 2:*

|  |
| --- |
| num1 <- 100  num2 <- 40.56  print(num1) #prints 100  print(num2) #prints 40.56 |

## **2.4 Character**

* A character is a sequence of one or more characters (letters, numbers, symbols) means Character data type is used to represent single or group of characters that can be a-z, A-Z, 0-9 or any special characters.
* Characters exist within either *single quotes ' or double quotes “.*

# *For Example :*

# *Example 1 :*

|  |
| --- |
| msg = ‘Hello World’ print(msg) #prints Hello World |

*Example 2 :*

|  |
| --- |
| fname = “Ramesh”  lname = ‘’’Singh’’’  full = fname +’ ‘+ lname  print(“My Name is “ , full ) #My name is Ramesh Singh |

## *Example 3:*

|  |
| --- |
| char1 <- "Hello world" |

|  |
| --- |
| char2 <- 'I am awesome' |

## **2.5 Logical**

* The Logical data type can be one of two values, either True or False.
* The values True and False will be always begin with a capital T and F

*For Example :*

*Example 1:*

|  |
| --- |
| bool1 <- TRUE |

|  |
| --- |
| bool2 <- FALSE |

|  |
| --- |
| # single character T and F can be used as well  bool3 <- T |

|  |
| --- |
| bool4 <- F |

|  |
| --- |
| bool3  # Output :  TRUE |

|  |
| --- |
| bool4  #Output :  FALSE |

*Example 2:*

|  |
| --- |
| state = TRUE  print(state) #prints True  state = FALSE  print(state) #prints False |

**class() Function**

Let's check the data type of the above variables using class function

|  |
| --- |
| num1<-50  class(num1) #output : 'numeric' |

|  |
| --- |
| num2<-56.89  class(num2) #output : ‘numeric’ |

|  |
| --- |
| char1<- "Hello world"  class(char1) #output : ‘character’ |

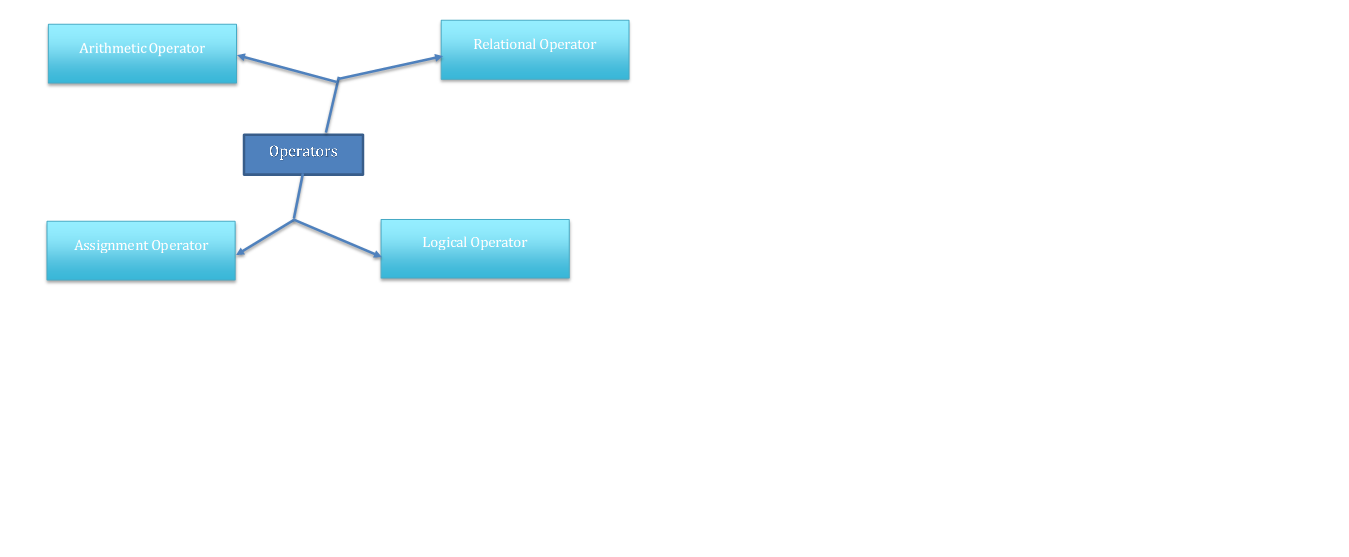
|  |
| --- |
| bool1<-TRUE  class(bool1) #output : 'logical' |

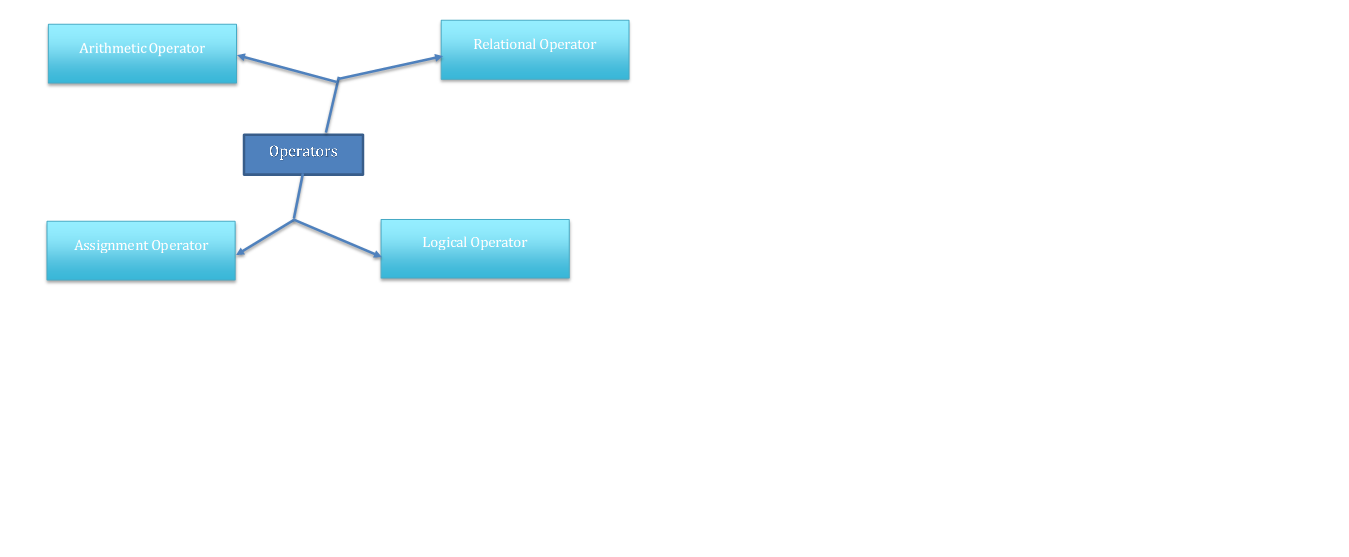
|  |
| --- |
| bool2 <- FALSE  class(bool2) #output : 'logical' |

|  |
| --- |
| bool3 <- T  class(bool3) #output : 'logical' |

**3. Data Operators In R :**

Operations : Operators are the constructs which can manipulate the value of operands.





|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **3.1 Assignment Operator**  The assignment operator takes the value or an expression and assigns it to a variable.  Leftward assignment =, <-, <<-  Rightward assignment ->, ->>  **3.2 Arithmetic Operator**  The following operators are valid arithmetic operators:   |  | | --- | | 1. Addition + ⇒ a+b 2. Subtraction - ⇒ a-b 3. Multiplication \* ⇒ a\*b 4. Divide / ⇒ a/b 5. Floor Division %/% ⇒ a%/%b 6. Exponential ^ or \*\* ⇒ a^b or a\*\*b 7. Modulus Operator %% ⇒ a%%b |   **3.3 Comparison Operator**  The following operators are valid Comparison operators:   |  | | --- | | 1. Equal To ⇒ == 2. Not Equal To ⇒ != 3. Greater Than ⇒ > 4. Greater Than or Equal To ⇒ >= 5. Less Than ⇒ < 6. Less Than or Equal To ⇒ <= |     **3.4 Logical Operator**  The following operators are valid Logical operators:   |  | | --- | | 1. And ⇒ && or & 2. Or ⇒ || or | 3. Not ⇒ ! |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **AND** | | |  | **OR** | | | | True | True | True |  | True | True | True | | True | False | False |  | True | False | True | | False | True | False |  | False | True | True | | False | False | False |  | False | False | False |  |  |  | | --- | --- | | **NOT** | | | True | False | | False | True |   *For Example :*  *1) Arithmetic operators :*  *Addition :*   |  | | --- | | 10 + 20  #output : 30 |   Subtraction :   |  | | --- | | 50-20  #output : 30 |   Multiplication :   |  | | --- | | 5\*2  #output : 10 |   Division :   |  | | --- | | 5/2  #output : 2.5 |   Floor Division or Integer Division   |  | | --- | | 5%/%2  #output : 2 | |
|  | Modulus Operator : To return remainder value   |  | | --- | | 5%%2  #output : 1 |   Exponential Operator: To return the power of given base .   |  | | --- | | 5^2  #output : 25 |   *2) Comparison operators*  Greater Than :   |  | | --- | | 10>5  #output: TRUE  10<5  #output : FALSE |   Greater Than or Equal To :   |  | | --- | | 10>=10  #output : TRUE  10<=5  #output : FALSE | |

Less Than :

|  |
| --- |
| 5<10  #output : TRUE  10<5  #output : FALSE |

Less Than or Equal To :

|  |
| --- |
| 5<=5  #output : TRUE  5<=6  #outptu : FALSE |

Equal To :

|  |
| --- |
| 5==5  #output : TRUE  5!=5  output : FALSE |

Not Equal To :

|  |
| --- |
| 5!=9  output : TRUE  5!=5  Output : FALSE |

*3) Logical operators*

AND :

|  |
| --- |
| 6>5 && 10>5  #output : TRUE  4 > 2 && 3 > 6  #output : FALSE |

OR :

|  |
| --- |
| 6>5 || 4>5  #output : TRUE  4 > 12 || 13 > 6  #output : FALSE |

NOT:

|  |
| --- |
| !(4 > 2)  #output :  FALSE  !(2 > 4)  #output :  TRUE |

*0 is considered as FALSE and any number greater than 0 is TRUE*

|  |
| --- |
| 4 && 5  #output : TRUE |

|  |
| --- |
| 4 && 0  #output : FALSE |

|  |
| --- |
| 4 || 5  #output : TRUE |

|  |
| --- |
| 4 || 0  #output : FALSE |

*4) Assignment operators*

Leftwards assignment :

|  |
| --- |
| num1 = 10  num1  #output : 10 |

|  |
| --- |
| num2 <- 20  num2  #output : 20 |

|  |
| --- |
| num3 <<-30  Num3  #output : 30 |

Rightward assignment:

|  |
| --- |
| 40 -> num4  num4  #output : 40 |

|  |
| --- |
| 50 ->> num5  num5  #output : 50 |

->> and <<- are used for global assignment

**4.** **Vectors In R**

**4.1 Introduction**

Vectors are one of the most important data structures in R. Vectors are one dimensional array that can store homogeneous (same data type) data.

In R vectors are created using combine function c().

Since vectors can store only homogeneous data, R will convert the other elements in the array to force everything to be of the same data type.

In R, a sequence of elements which share the same data type is known as vector. A vector supports logical, integer, double, character, complex, or raw data type. The elements which are contained in a vector known as *components* of the vector. We can check the type of vector with the help of the *typeof()* function.

The length is an important property of a vector. A vector length is basically the number of elements in the vector, and it is calculated with the help of the length() function.

Vector is classified into two parts, i.e., *Atomic vectors and Lists.* They have three common properties, i.e., *function type, function length, and attribute function.*

There is only one difference between atomic vectors and lists.

|  |  |
| --- | --- |
| **Atomic vectors** | **Lists** |
| In an atomic vector, all the elements are of the same type | in the list, the elements are of different data types. |

**4.2 Create a vector in R, which are as follows:**

In R, we use c() function to create a vector. This function returns a one-dimensional array or simply vector. The c() function is a generic function which combines its argument. All arguments are restricted with a common data type which is the type of the returned value. There are various other ways to create a vector in R, which are as follows:

### *1) Using the colon(:) operator*

We can create a vector with the help of the colon operator. There is the following syntax to use colon operator:

|  |
| --- |
| 1. z<-x:y |

This operator creates a vector with elements from x to y and assigns it to z.

Example:

|  |
| --- |
| 1. a**<-4:-10** 2. a   Output  [1] 4 3 2 1 0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 |

### *2) Using the seq() function*

In R, we can create a vector with the help of the seq() function. A sequence function creates a sequence of elements as a vector. The seq() function is used in two ways, i.e., by setting step size with ?by' parameter or specifying the length of the vector with the 'length.out' feature.

Example:

|  |
| --- |
| seq\_vec<-seq(1,4,by=0.5)  seq\_vec  class(seq\_vec)  #Output  [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 |

Example:

|  |
| --- |
| 1. seq\_vec**<-**seq(1,4,length.out=6) 2. seq\_vec 3. class(seq\_vec)   #Output  [1] 1.0 1.6 2.2 2.8 3.4 4.0  [1] "numeric" |

### *3) Using the c() function*

### *1.Integer vector :* A vector which contains integer elements is known as an integer vector.

Example:

|  |
| --- |
| a<-c(10,20,45,77,80)  a  class(a)  #output :  [1] 10 20 45 77 80  [1] "numeric" |

*3.Character vector :* A vector which contains character elements is known as a Character vector.

Example:

|  |
| --- |
| char\_vec1**<-c**("shubham","arpita","nishka","vaishali")  char\_vec1  class(char\_vec1)  #output :  [1] "shubham" "arpita" "nishka" "vaishali"  [1] "character" |

### *4.Logical vector:* The logical data types have only two values i.e., True or False. These values are based on which condition is satisfied. A vector which contains Boolean values is known as the logical vector.

### Example :

|  |
| --- |
| log\_vec**<-**c(d<e, d<f, e<d,e<f,f<d,f<e)  log\_vec  class(log\_vec)  #output :  [1] TRUE TRUE FALSE TRUE FALSE FALSE  [1] "logical" |

Since vectors can store only homogeneous data, R will convert the other elements in the array to force everything to be of the same data type.

If a vector is created in this fashion

|  |
| --- |
| vector = c(“A”, 12, 12.5, TRUE)  Output vector will be -  “A” “12” “12.5” “TRUE” |

If you check the data type of the above vector using class() function, the output will be character.

|  |
| --- |
| class(vector)  Output : character |

*Creating vector :*

|  |
| --- |
| v1 <- c(1,2,3,4)  v1  #output :  1 2 3 4 |

|  |
| --- |
| v2 <- c('A','B','C')  v2  #output :  'A' 'B' 'C' |

*Checking class of the above vectors*

|  |
| --- |
| class(v1)  #output :  'numeric' |

|  |
| --- |
| class(v2)  #output :  'character' |

Since vector can store only homogeneous data, R will convert the other elements in the array to force everything to be of the same data type.

|  |
| --- |
| v3 <- c(1,2,TRUE,FALSE)  v3  class(v3)  #output  1 2 1 0  'numeric' |

|  |
| --- |
| v4 <- c(4,5,'Hello','World')  v4  class(v4)  #output  '4' '5' 'Hello' 'World'  'character' |

|  |
| --- |
| v5 <- c(TRUE,FALSE,'Hello','World')  v5  class(v5)  #output :  'TRUE' 'FALSE' 'Hello' 'World'  'character' |

*Vector Names :*

Name function name() is used to assign names to each elements of a vector

Example :

|  |
| --- |
| marks <- c(75,81,69,86)  #Define Name of vector marks  names(marks) <- c('Math','Science','English','GK')  marks |

|  |
| --- |
| Output :  Math 75  Science 81  English 69  GK 86 |

*Creating a sequence vector using :*

|  |
| --- |
| 1:10  #output  1 2 3 4 5 6 7 8 9 10 |

|  |
| --- |
| v <- 5:10  v  #output :  5 6 7 8 9 10 |

**4.3 Vector operations**

*1) Arithmetic operations*

|  |
| --- |
| v1 <- c(1,2,3)  v2 <- c(4,5,6) |

|  |
| --- |
| v1  #output  1 2 3 |

|  |
| --- |
| v2  #output  4 5 6 |

*Addition:*

|  |
| --- |
| v1+v 2  #output  5 7 9 |

Subtraction:

|  |
| --- |
| v1-v2  #output  -3 -3 -3 |

Multiplication

|  |
| --- |
| v1 \* v2  #output  4 10 18 |

Division

|  |
| --- |
| v1/v2  #output  0.25 0.4 0.5 |

*Vector and Scalar operation*

|  |
| --- |
| v1 + 3  #output  4 5 6 |

|  |
| --- |
| v2 \* 4  #output  16 20 24 |

*2) Comparison operations*

|  |
| --- |
| v1 < v2  #output  TRUE TRUE TRUE |

|  |
| --- |
| v1 == v2  #output  FALSE FALSE FALSE |

|  |
| --- |
| # Vector and scalar  v1 > 2  #output  FALSE FALSE TRUE |

*3) Logical operations*

*Logical AND & OR*

|  |
| --- |
| v1 && v2  #output  TRUE |

|  |
| --- |
| v1 || v2  #output  TRUE |

The above operators works only for first element of the vector

*Element wise AND & OR*

|  |
| --- |
| v1 & v2  #output  TRUE TRUE TRUE |

|  |
| --- |
| v1 | v2  #output  TRUE TRUE TRUE |

**4.4) Vector functions**

Let's see some of the inbuilt functions that can be used with vectors

*1.sum*

|  |
| --- |
| v <- c(10,20,25,34,56,78)  sum(v)  #output  223 |

*2.mean*

|  |
| --- |
| mean(v)  #output  37.1666666666667 |

*3.Product*

|  |
| --- |
| prod(v)  #output  742560000 |

*4.Variance*

|  |
| --- |
| var(v)  #output  642.566666666667 |

*5.Standard deviation*

|  |
| --- |
| sd(v)  #output  25.3488987269007 |

*6.Max*

|  |
| --- |
| max(v)  #output  78 |

*6.min*

|  |
| --- |
| min(v)  #output  10 |

*7.Length*

|  |
| --- |
| length(v)  #output  6 |

**5) Indexing and Slicing**

*Indexing*

|  |
| --- |
| v <- c(10,20,25,34,56,78) |

|  |
| --- |
| # Index value starts with 1  v[1]  #output  10 |

|  |
| --- |
| v[4]  #output  34 |

|  |
| --- |
| # For index more than length of vector  v[7]  #output  <NA> |

*Multiple Indexing*

|  |
| --- |
| v[c(1,4)]  #output  10 34 |

|  |
| --- |
| v[c(1,9)]  #output  10 <NA> |

*Slicing*

|  |
| --- |
| v[2:4]  #output  20 25 34 |

|  |
| --- |
| v[3:6]  #output  25 34 56 78 |

*Indexing with names*

|  |
| --- |
| names(v) <- c('a','b','c','d','e','f')  V['a']  #output  a: 10 |

|  |
| --- |
| v['e']  #output  e: 56 |

|  |
| --- |
| v[c('a','e')]  #output  a 10  e 56 |

|  |
| --- |
| V['z']  #output  NA: <NA> |

*Conditional selection*

|  |
| --- |
| v[v>20]  #output  c 25  d 34  e 56  f 78 |

|  |
| --- |
| v[v %% 2 == 0]  #output  a 10  b 20  d 34  e 56  f 78 |

|  |
| --- |
| v[v %% 4 ==0]  #output  b 20  e 56 |

## 

## *Applications of vectors*

1. In machine learning for principal component analysis vectors are used. They are extended to eigenvalues and eigenvector and then used for performing decomposition in vector spaces.
2. The inputs which are provided to the deep learning model are in the form of vectors. These vectors consist of standardized data which is supplied to the input layer of the neural network.
3. In the development of support vector machine algorithms, vectors are used.
4. Vector operations are utilized in neural networks for various operations like image recognition and text processing.

**Conditional statements in R**

* The if control structure will execute or skip executing the code depending on a condition.
* If the condition evaluates to True then it will execute otherwise it does not.
* An optional else part is executed if the condition evaluates to False.

Syntax:

if (boolean expression) {

#statements to be executed when expression is True

} else {

#statements to be executed when expression is False

}

*code executes if condition returns TRUE*

Example 1:

|  |
| --- |
| c = TRUE  if (c) {  print(“Right”)  }else{  print(“Wrong”)  } |

Example 2:

|  |
| --- |
| if (10.0 == 10) {  print(“ Equal “)  } |

A conditional block is a set of instructions that executes once, only if certain condition is satisfied.

*if statement*

Syntax -

if (condition) {

#code

}

*code executes if condition returns TRUE*

|  |
| --- |
| age <- 20  if(age >= 18){  print("Eligible")  }  age <- 12  if(age >= 18){  print("Eligible")  #output  [1] "Eligible" |

|  |
| --- |
| age <- 12  if(age >= 18){  print("Eligible")  } |

if else statement

Syntax -

if (condition) {

#code 1

}else{

#code 2

}

code 1 executes if condition returns TRUE, else code 2 executes

|  |
| --- |
| age <- 12  if(age >= 18){  print("Eligible")  }else{  print("Not Eligible")  }  #output  [1] "Not Eligible" |

*if else-if else statement :*

if else-if else statements are used for handling multiple conditions.

Syntax -

if (condition 1) {

#code 1

}else if(condition 2){

#code 2

}else if(condition 3){

#code 3

}else{

#code 4

}

code 1 executes if condition 1 returns TRUE

code 2 executes if condition 2 returns returns TRUE

code 3 executes if condition 3 returns TRUE

else code 4 executes

|  |
| --- |
| percentage <- 78  if(percentage >= 80){  print("Grade A+")  }else if(percentage >= 70){  print("Grade A")  }else if(percentage >=60){  print("Grade B")  }else{  print("Grade C")  }  #output  [1] "Grade A" |

*Nested if else statement*

|  |
| --- |
| Write a program to check the given year is leap year or not   1. year 2. 1 = 2011 3. **if**(year1 %% 4 == 0) { 4. **if**(year1 %% 100 == 0) { 5. **if**(year1 %% 400 == 0) { 6. cat(year,"is a leap year") 7. } **else** { 8. cat(year,"is not a leap year") 9. } 10. } **else** { 11. cat(year,"is a leap year") 12. } 13. } **else** { 14. cat(year,"is not a leap year") 15. } |

**Loops in R**

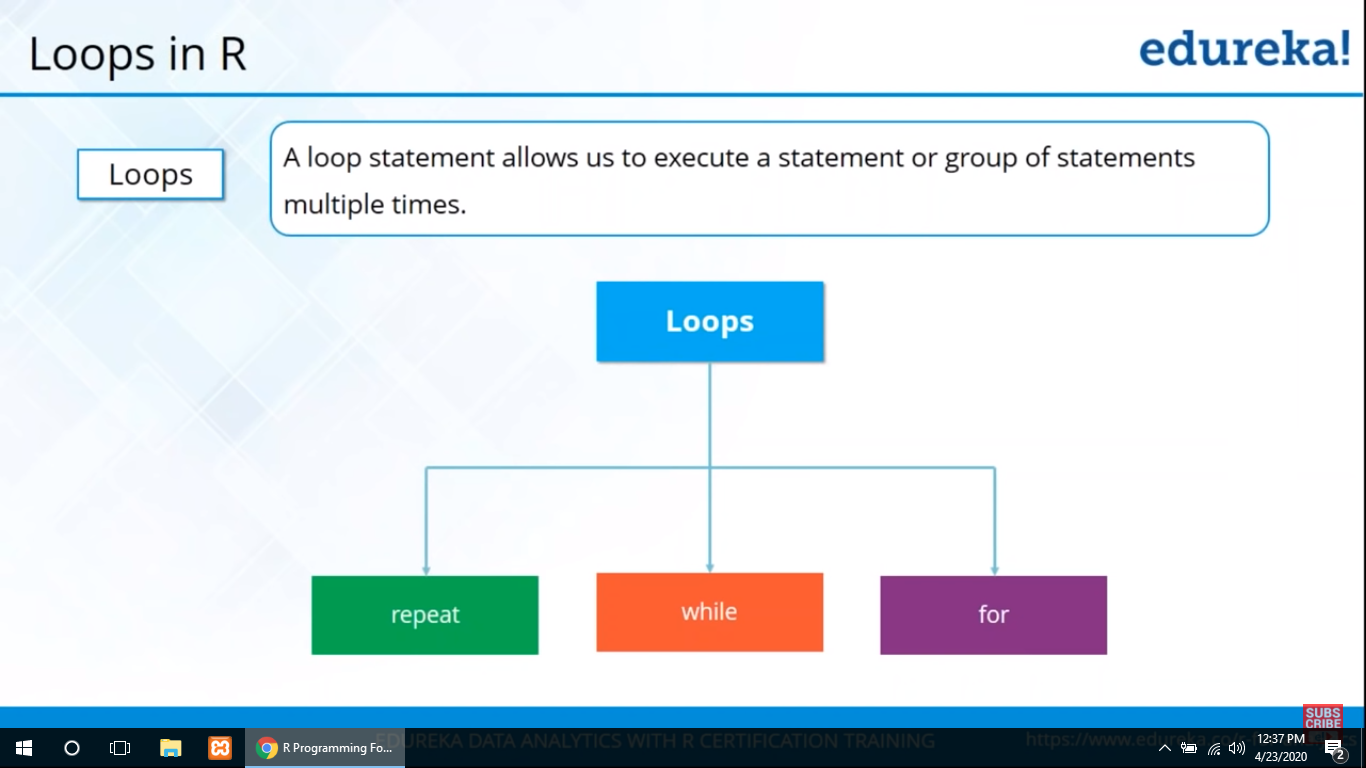
*What are loops and their uses ?*

A loop is repeated execution of code again. Eg. Walking, Drinking water, Fan Rotation

● Every loop has start and an end

● What stops the loop at the end is known as its condition Eg. Arriving at the destination while walking or when the glass is empty else thirst is quenched we stop drinking.

● There are three loops in R: repeat ,while and for loop..



# **R repeat loop**

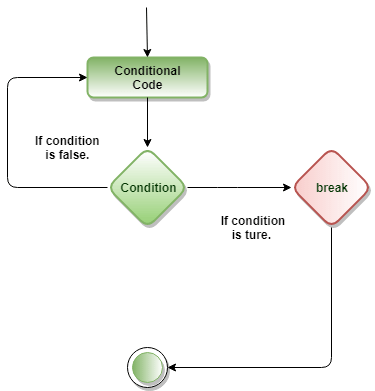
A repeat loop is used to iterate a block of code. It is a special type of loop in which there is no condition to exit from the loop. For exiting, we include a break statement with a user-defined condition. This property of the loop makes it different from the other loops.

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. }

*Flow Chart :*



*Working of Repeat Loop*

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.

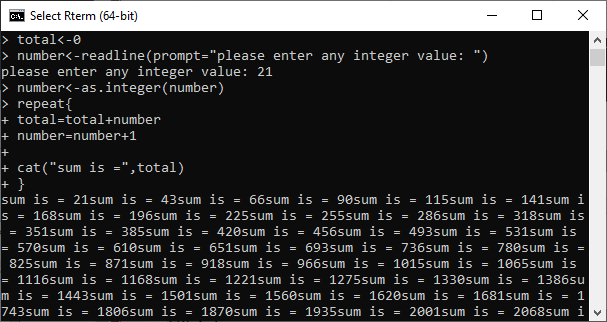
Example :

|  |
| --- |
| v **<-** c("Hello","repeat","loop")  cnt **<-** 2  repeat {  print(v)  cnt **<-** cnt+1  if(cnt **>** 5) {  break  }  }  #output :  [1] "Hello","repeat","loop"  [1]"Hello","repeat","loop"  [1] "Hello","repeat","loop"  [1]"Hello","repeat","loop" |

*Example : Infinity repeat loop*

|  |
| --- |
| total<-0  number<-readline(prompt="please enter any integer value: ")  repeat{  total=total+number  number=number+1  cat("sum is =",total)  } |

Output :

****

*2. while Loop*

Syntax:

Initialize value

while (cond) {

repeating statements

}

Example 1: Write a Program to print first 10 natural numbers

|  |
| --- |
| i = 1 #initial value  while (i<=10) {  print(i)  i =i+1 #increment by 1  }  print(“Loop finished”)  Output :  1  2  3  4  5  6  7  8  9  10 |

*Example : Reverse of a number*

|  |
| --- |
| n = 123  rev=0  while (n > 0) {  rem = num %% 10.  rev = (rev\*10) + rem.  num = num %/% 10  }.  cat(“Reverse No. : “,rev)  #output :  Reverse No. : 321 |

*Example : factorial of number*

|  |
| --- |
| num <- 4  fact <- 1  while (num >= 1) {  fact <- fact\* num  num<-num-1  }  cat(“Factorial of given number : “ ,fact)  #output :  Factorial of given number : 24 |

*For loop*

Syntax:

for ( variable\_reference in object){

repeating statements

}

Example:

|  |
| --- |
| vector = c( 10, 20, 30, 40)  for ( element in vector )  {  print( element)  }  #output  10  20  30  40 |

|  |
| --- |
| vector = c( 10, 20, 30, 40)  for(i in 1:length(vector))  {  print(vector[i])  }  #output  10  20  30  40 |

*Write a program to find the sum of first n odd numbers using for loop*

|  |
| --- |
| sum=0  n=as.integer(readline(“How many times to sum odd number”)  for(i in 1:n)  {  if(i%%2!=0)  {  sum=sum+i  }  }  cat(“sum of first n odd nos. : “,sum) |

Additional Exercise:

1. WAP to add all the digits of a number and print the sum.

2. Check if the number input by the user is an Armstrong number.

Note: An armstrong number is where the sum of cubes of all the digits gives the

original number.

3. WAP to check the given number is palindrom or not.

4. WAP to print the table of any number.

Note : 5\*1=5 , 5\*2=10, …………………….,5\*10=50

**Break in R**

● Break is used to exit the loop before the condition of the loop turns false.

● It prematurely terminates the loop often using a base condition using if.

● break is utmost important to exit the otherwise never ending infinite loop.

● The else part is not executed if the loop exits

Example :

|  |
| --- |
| n <- 1  while(n <= 5){  if(n == 3){  break  }  print(paste("Value of n: ",n))  n <- n+1  }  #output  "Value of n: 1"  "Value of n: 2" |

**Functions in R :**

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

1. Written to carry out a specified task.
2. May or may not have arguments
3. Contain a body in which our code is written.
4. 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 R, an argument is a placeholder. 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 functions,* such as c() to create a vector, print() to print an object to the

terminal etc.

● *User-Defined Functions (UDFs),* which are functions that users create to help

them out.

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

## *1. 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 |

## **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. | substr(x, start=n1,stop=n2) | It is used to extract substrings in a character vector. | a <- "987654321"  substr(a, 3, 3)  Output  [1] "3" |
| 2. | grep(pattern, x , ignore.case=FALSE, fixed=FALSE) | It searches for pattern in x. | st1 <- c('abcd','bdcd','abcdabcd')  pattern<- '^abc'  print(grep(pattern, st1))  Output  [1] 1 3 |
| 3. | 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" |
| 4. | 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 |
| 5. | 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" |
| 6. | tolower(x) | It is used to convert the string into lower case. | st1<- "shuBHAm"  print(tolower(st1))  Output  [1] shubham |
| 7. | 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, trim=0, na.rm=FALSE) | 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. | diff(x, lag=1) | It returns differences with lag indicating which lag to use. | a<-c(0:10, 40)  xm<-diff(a)  print(xm)  Output  [1] 1 1 1 1 1 1 1 1 1 1 30 |
| 8. | min(x) | It returns minimum value. | a<-c(0:10, 40)  xm<-min(a)  print(xm)  Output  [1] 0 |
| 9. | max(x) | It returns maximum value | a<-c(0:10, 40)  xm<-max(a)  print(xm)  Output  [1] 40 |
| 10. | scale(x, center=TRUE, scale=TRUE) | Column center or standardize a matrix. | a <- matrix(1:9,3,3)  scale(x)  Output  [,1]  [1,] -0.747776547  [2,] -0.653320562  [3,] -0.558864577  [4,] -0.464408592  [5,] -0.369952608  [6,] -0.275496623  [7,] -0.181040638  [8,] -0.086584653  [9,] 0.007871332  [10,] 0.102327317  [11,] 0.196783302  [12,] 3.030462849  attr(,"scaled:center")  [1] 7.916667  attr(,"scaled:scale")  [1] 10.58694 |

## Example :

|  |
| --- |
| # Creating sequence of numbers from 32 to 46.  print(seq(32,46))  # Finding the mean of numbers from 22 to 80.  print(mean(22:80))  # Finding the sum of numbers from 41 to 70.  print(sum(41:70)) |

### *2.User-Defined Functions (UDFs) :*

*Defining & Calling Functions:*

Functions are defined with the function statement:

Syntax:

|  |
| --- |
| function.name = function(parameters) {    “function\_docstring”  return [expression]  } |

Example:

|  |
| --- |
| greet = function(){    print (“Hello world”)  } |

*Types of Function:*

1. *No parameter no return:*

In this type of function in R, while defining, declaring or calling the function, we won’t pass any arguments to the function. This type of function will not return any value when we call the function.

Syntax:

|  |
| --- |
| function.name = function( ) {  “function\_docstring”  }  #function calling  function.name() |

Whenever we are not expecting any return value, we might need some statements to be printed as output. In such a scenario, we can use this type of function.

## *Function calling with no argument and no return*

In R, we can call a function without an argument in the following way

# Creating a function to print squares of numbers in sequence.

|  |
| --- |
| new.function <- function() {  for(i in 1:5) {  a <- i^2  print(a)  }  }  # Calling the function new.function with no argument.  new.function()  #output :  1  4  9  16  25 |

1. *Function with no argument and with Return value*

In this type of function in R, we won’t pass any arguments to the function while defining, declaring or calling the function. When we call the function, this type of function will return some value.

Syntax:

|  |
| --- |
| function.name = function( ) {    “function\_docstring”  return [expression]  }  #function calling  output = function.name() |

1. *Function with argument and No Return value*

In real-time, we mostly deal with dynamic data means we have to allow the user to enter his own values rather than fixed ones.

This type of function in R allows us to pass the arguments to the function while calling the function. But, this type of function in R will not return any value when we call the function.

Syntax:

|  |
| --- |
| function.name = function( parameters ) {    “function\_docstring”    }    #function calling  function\_name( arguments ) |

## 

## *Example : Function calling with an argument and no return value*

We can easily call a function by passing an appropriate argument in the function. Let see an example to see how a function is called.

# Creating a function to print squares of numbers in sequence.

|  |
| --- |
| new.function <- function(a) {  for(i in 1:a) {  b <- i^2  print(b)  }  # Calling the function new.function supplying 10 as an argument. new.function(10)  Output:  1  4  9  16  25  36  49  64  81  100 |

1. *Function with argument and Return value*

This method allows us to pass the arguments to the function while calling the function. This type of function in R will return some value when we call the function. This type of user defined functions are called as fully dynamic function means, it provides maximum control to the end-user.

Syntax:

|  |
| --- |
| function.name = function( parameters ) {    “function\_docstring”  return [expression]  }  #function calling  output = function.name( arguments ) |

## *Example : Function calling with Argument and return Values*

We can supply the arguments to a function call in the same sequence as defined in the function or can supply in a different sequence but assigned them to the names of the arguments.

# Creating a function with arguments.

|  |
| --- |
| new.function <- function(x,y,z) {  result <- x \* y + z  return(result)  }  # Calling the function by position of arguments.  result=new.function(11,13,9)  cat(“Answer : “,result)  # Calling the function by names of the arguments.  result=new.function(x = 2, y = 5, z = 3)  cat(“Answer : “,result)  #output  Answer : 152  Answer : 13 |

## *Function calling with default arguments*

To get the default result, we assign the value to the arguments in the function definition, and then we call the function without supplying argument. If we pass any argument in the function call, then it will get replaced with the default value of the argument in the function definition.

*# Creating a function with arguments.*

|  |
| --- |
| new.function <- function(x = 11, y = 24) {  result <- x \* y  print(result)  }  # Calling the function without giving any argument.  new.function()  # Calling the function with giving new values of the argument.  new.function(4,6)  Output:  264  24 |

**Print Formatting**

|  |  |  |
| --- | --- | --- |
|  | | **1) print()** |
| **print()** function can be used to print out variables or strings |
| In | [1]: | print("Hello world") |
|  |  | [1] "Hello world" |
| In | [2]: | x **<-** 10 |
|  |  |  |
| In | [3]: | print(x) |
|  |  | [1] 10 |
|  |  | **2) paste()** |
|  |  | **paste()** is used along with print() function for formatting |
| In | [1]: | num1 **<-** 10  num2 **<-** 20 |
|  |  |  |
| In | [3]: | print(paste("Sum",num1**+**num2)) |
|  |  | [1] "Sum 30" |

In [14]:

[1] "Sum:30"

**3) sprintf()**

**sprintf()** is a wrapper of C print() function

# %s - character/string

1. %i - integer
2. %f - float

|  |  |  |
| --- | --- | --- |
| In | [21]: | print(sprintf("My name is %s","Batman")) |
|  |  | [1] "My name is Batman" |
| In | [22]: | print(sprintf("I am %i years old",28)) |
|  |  | [1] "I am 28 years old" |
| In | [30]: | print(sprintf("%s has max speed of %f","Ferrari",217.5)) |
|  |  | [1] "Ferrari has max speed of 217.500000" |
| In | [31]: | print(sprintf("%s has max speed of %.2f","Ferrari",217.5)) |
|  |  | [1] "Ferrari has max speed of 217.50" |