R IN 30 HOURS

Develop Achiever's Mindset and Habits, Work Smarter and Still Create Time For Things That Matter

O.S. ABDUL QADIR **M. IBRAMSHA**

Contents

Introduction

- R is a *programming language* and *software* environment for *statistical analysis*, *graphics representation* and *reporting*.
- This language was named **R**, based on 1ST letter of first name of the two authors (***Robert Gentleman*** and ***Ross Ihaka***).
- R is *freely available* under the *General Public License (GNU)* and pre-compiled binary versions are provided for operating systems (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**.
- R allows integration with the procedures written in the ***C, C++, .Net, Python*** or ***FORTRAN*** languages for efficiency.

Evolution of R

- R is an interpreted programming language which was created by ***Ross Ihaka*** and ***Robert Gentleman*** at the ***University of Auckland, New Zealand*** and is currently developed by the ***R Development Core Team***.
- R made its first appearance in ***1993***.
- A large group of individuals contributed to R by sending code and bug reports.
- Since mid-1997 there has been a core group (the "R Core Team") who can modify the R source code archive.

Features of R

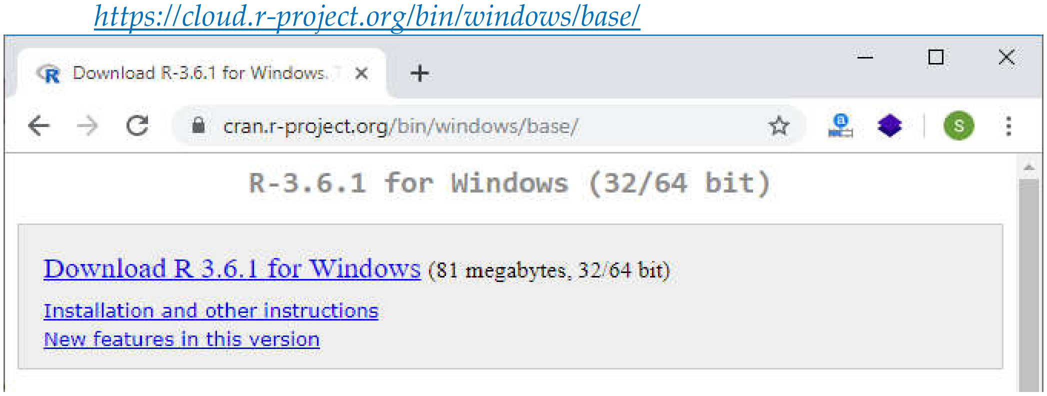
- R is a well-developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and I facilities.
- R has an effective data handling and storage facility,
- R provides a suite of operators for calculations on arrays, lists, vectors and matrices.
- 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.

Environment Setup

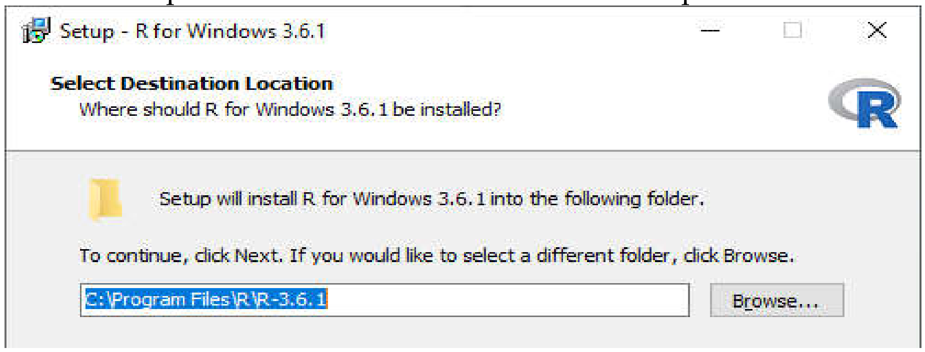
1. ***Windows Installation***

   First, we have to download the R setup from

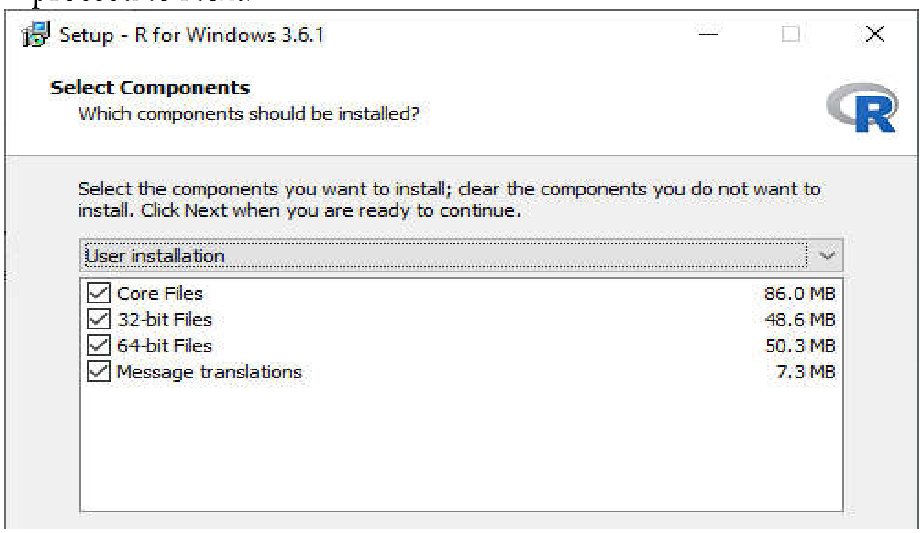
   https://cloud.r-project.org/bin/windows/base/



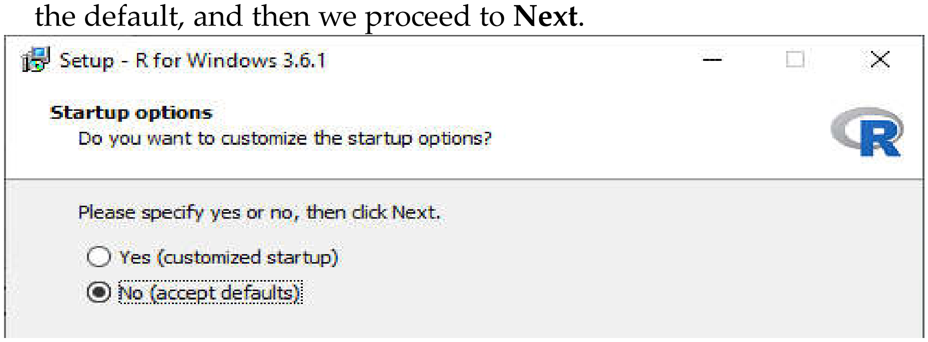
- When we click on Download, our downloading will be started of R setup. Once it is finished, we have to run the setup of R in the following way:
  - Select the path where we want to download and proceed to Next.



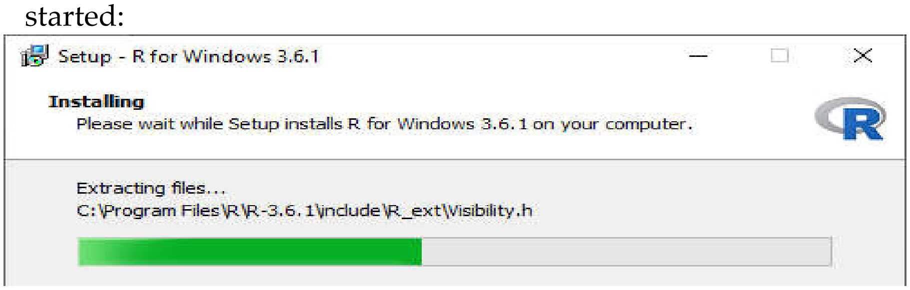
  - Select all components which we want to install, and then we will proceed to **Next**.



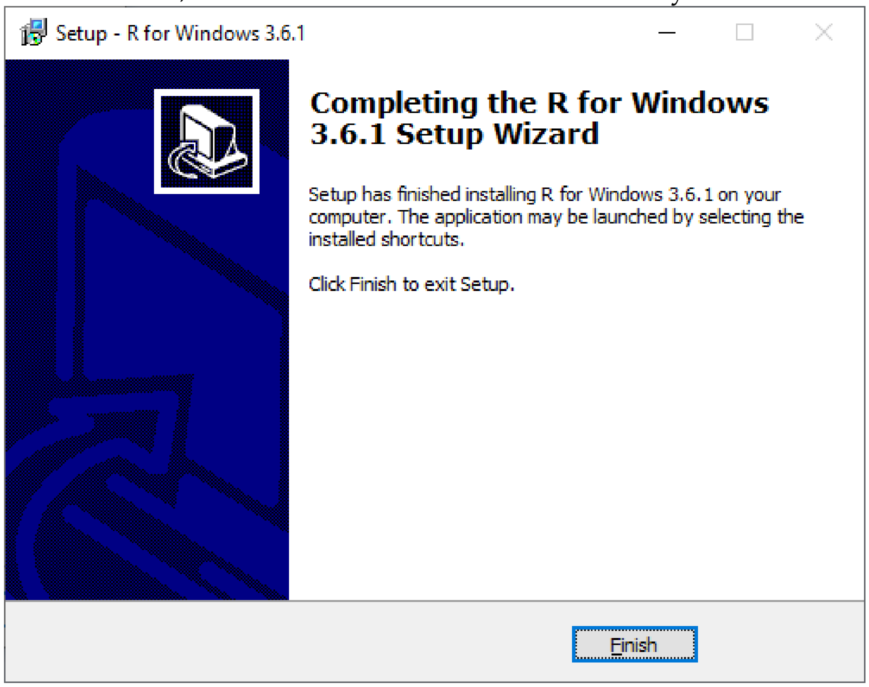
  - In the next step, we have to select either customized startup or accept the default, and then we proceed to **Next**.



  - When we proceed to next, our installation of R in our system will get started:

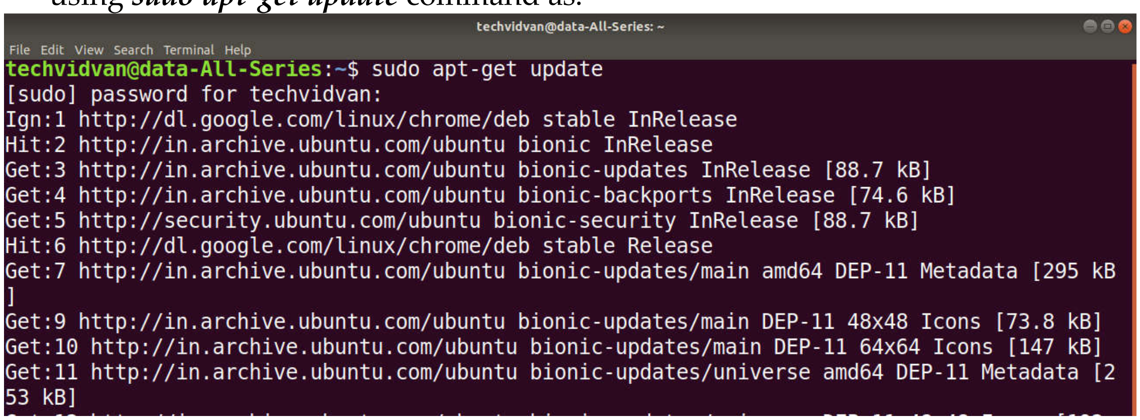


® In the last, we will click on finish to successfully install R.

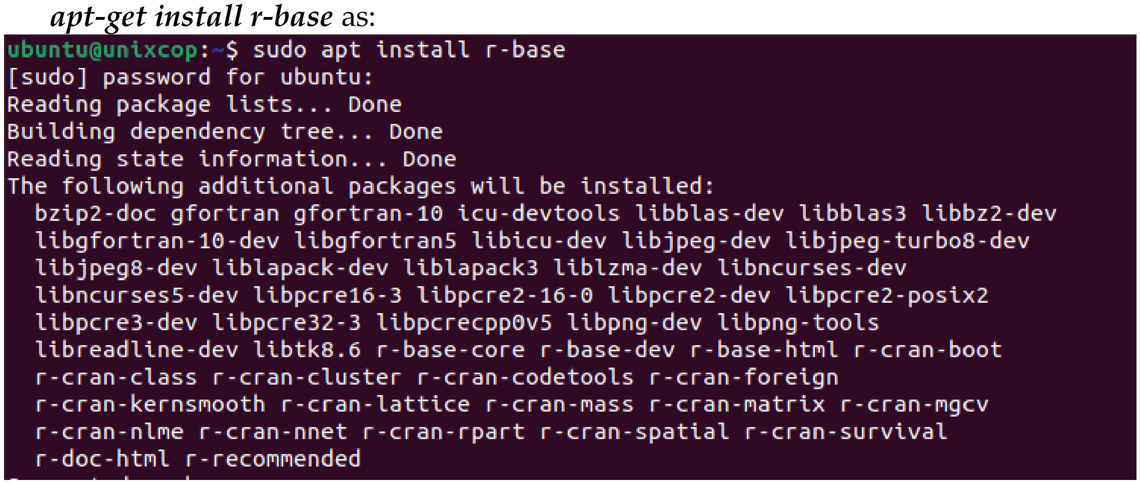


2. *Linux Installation*

® In the first step, we have to update all the required files in our system using ***sudo apt-get update*** command as:



® In the second step, we will install R file in our system with the help of ***sudo apt-get install r-base*** as:



® In the last step, we type R and press enter to work on R editor.

```
ubuntu@ubuntu: ~
File  Edit  View  Search  Terminal  Help
ubuntu@ubuntu:~$ R

R version 3.6.0 (2019-04-26) -- "Planting of a Tree"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

R Basic Syntax

R Command Prompt

Once you have R environment setup, then it's easy to start your R command prompt by just typing the following command at your command prompt –

$ R

This will launch R interpreter and you will get a prompt > where you can start typing your program as follows –

```
> myString <- "Hello, SAQ!"
> print ( myString)

[1] "Hello, SAQ!"
```

® Here first statement defines a string variable myString, where we assign a string "Hello, SAQ!" and then next statement print() is being used to print the value stored in variable myString.

R Script File

Usually, you will do your programming by writing your programs in script files and then you execute those scripts at your command prompt with the help of R interpreter called **Rscript**. So let's start with writing following code in a text file called test.R as under –

```
# My first program in R Programming
myString <- "Hello, SAQ!"
print ( myString)
```

Save the above code in a file test.R and execute it at Linux command prompt as given below. Even if you are using Windows or other system, syntax will remain same.

$ Rscript test.R

When we run the above program, it produces the following result.
[1] "Hello, SAQ!"

Comments

- Comments are like helping text in your R program and they are ignored by the interpreter while executing your actual program.
- Single comment is written using # in the beginning of the statement as follows –

```
# My first program in R Programming
```

- R does not support multi-line comments but you can perform a trick which is something as follows –

```
if(FALSE)
{
   "This is a demo for multi-line comments and it should be put inside either a
      single OR double quote"
}
myString <- "Hello, SAQ!"
print ( myString)
```

  [1] "Hello, SAQ!"

  - Though above comments will be executed by R interpreter, they will not interfere with your actual program.
  - You should put such comments inside, either single or double quote

Datatypes

® In contrast to other programming languages like C and java in R, the variables are not declared as some data type.

® The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable.

® There are many types of R-objects.

® The frequently used ones are

1. Vectors
2. Lists
3. Matrices
4. Arrays
5. Factors
6. Data Frames

The simplest of these objects is the **vector object** and there are six data types of these atomic vectors, also termed as six classes of vectors.

S. No	Data Type	Example	Verify
1	***Logical***	TRUE, FALSE	v <- TRUE print(class(v)) ***OUTPUT*** [1] "logical"
2	***Numeric***	12.3, 5, 999	v <- 23.5 print(class(v)) ***OUTPUT*** [1] "numeric"
3	***Integer***	2L, 34L, 0L	v <- 2L print(class(v)) ***OUTPUT*** [1] "integer"
4	***Complex***	3 + 2i	v <- 2+5i print(class(v)) ***OUTPUT*** [1] "complex"
5	***Character***	'a' , '"good", "TRUE", '23.4'	v <- "TRUE" print(class(v)) ***OUTPUT*** [1] "character"
6	***Raw***	"Hello" is stored as 48 65 6c 6c 6f	v <- charToRaw("Hello") print(class(v)) ***OUTPUT*** [1] "raw"

In R programming, the very basic data types are the R-objects called **vectors** which hold elements of different classes.

1. *Vectors*

When you want to create vector with more than one element, you should use **c()** function which means to combine the elements into a vector.

```
# Create a vector.
color <- c('red','green',"yellow")
print(color)

# Get the class of the vector.
print(class(color))
```

OUTPUT

```
[1] "red"    "green"  "yellow"
[1] "character"
```

2. *Lists*

A list is an R-object which can contain many different types of elements inside it like vectors, functions and even another list inside it.

```
# Create a list.
list1 <- list(c(2,5,3),21.3,sin)

# Print the list.
print(list1)
```

OUTPUT

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

[[2]]
[1] 21.3

[[3]]
function (x)  .Primitive("sin")
```

3. *Matrices*

A matrix is a two-dimensional rectangular data set. It can be created using a vector input to the matrix function.

```
# Create a matrix.
M = matrix( c('a','a','b','c','b','a'), nrow = 2, ncol = 3, byrow = TRUE)
print(M)
```

OUTPUT

```
     [,1]  [,2]  [,3]
[1,]  "a"   "a"  "b"
[2,]  "c"   "b"  "a"
```

4. *Arrays*

® While matrices are confined to two dimensions, arrays can be of any number of dimensions.

- The array function takes a dim attribute which creates the required number of dimension.
- In the below example we create an array with two elements which are 3x3 matrices each.

```
# Create an array.
a <- array(c('green','yellow'),dim = c(3,3,2))
print(a)
```

OUTPUT

```
, , 1
     [,1]     [,2]     [,3]
[1,] "green"  "yellow" "green"
[2,] "yellow" "green"  "yellow"
[3,] "green"  "yellow" "green"

, , 2

     [,1]     [,2]     [,3]
[1,] "yellow" "green"  "yellow"
[2,] "green"  "yellow" "green"
[3,] "yellow" "green"  "yellow"
```

5. *Factors*

- Factors are the r-objects which are created using a vector.
- It stores the vector along with the distinct values in the vector as labels.
- The labels are always character irrespective of whether it is numeric or character or Boolean etc. in the input vector. They are useful in statistical modelling.
- Factors are created using the ***factor()*** function. The ***nlevels*** functions gives the count of levels.

```
# Create a vector.
apple_colors <- c('green','green','yellow','red','red','red','green')

# Create a factor object.
factor_apple <- factor(apple_colors)

# Print the factor.
print(factor_apple)
print(nlevels(factor_apple))
```

OUTPUT

```
[1] green  green  yellow red    red    red    green
Levels: green red yellow
[1] 3
```

6. *Data Frames*

- Data frames are tabular data objects. Unlike a matrix, here each column can contain different modes of data. The first column can be numeric while the second column can be character and third column can be logical.
- It is a list of vectors of equal length.
- Data Frames are created using the ***data.frame()*** function.

```
# Create the data frame.
  BMI <- data.frame(
  gender = c("Male", "Male","Female"),
  height = c(152, 171.5, 165),
  weight = c(81,93, 78),
  Age = c(46,28,16)
)
print(BMI)
```

OUTPUT

```
      gender    height    weight    Age
1     Male      152.0     81        46
2     Male      171.5     93        28
3     Female    165.0     78        16
```

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 Robjects.
- A variable
    1. Valid – Has letters, numbers, underscore and dot.
    2. Invalid
        a. Starts with a number or an underscore (_).
        b. The starting dot is followed by a number.
        c. Using special characters other than dot(.) and underscore(_).

Variable Name Example	Validity		Variable Name Example	Validity
var_name2.	Valid		var_name%	Invalid
.var_name,	Valid		2var_name .2var_name	Invalid
var.name	Valid			
			_var_name	Invalid

Variable Assignment

- The variables can be assigned values using leftward, rightward and equal to operator. The values of the variables can be printed using ***print()*** or ***cat()*** *function*.
- The **cat()** function combines multiple items into a continuous print output.

```
# Assignment using equal operator.
var.1 = c(0,1,2,3)
```

```
# Assignment using leftward operator.
var.2 <- c("SAQ","MIS")

# Assignment using rightward operator.
c(TRUE,1) -> var.3
print(var.1)
cat ("var.1 is ", var.1 ,"\n")
cat ("var.2 is ", var.2 ,"\n")
cat ("var.3 is ", var.3 ,"\n")
```

OUTPUT

```
[1] 0 1 2 3
var.1 is  0 1 2 3
var.2 is  SAQ MIS
var.3 is  1 1
```

Note − The vector c(TRUE,1) has a mix of logical and numeric class. So logical class is coerced to numeric class making TRUE as 1.

Data Type of a Variable

In R, a variable *itself is not declared* of any data type, rather it gets the data type of the R - object assigned to it.

So R is called a dynamically typed language, which means that *we can change a variable's data type of the same variable again* and again when using it in a program.

```
var_x <- "Hello"
cat("The class of var_x is ",class(var_x),"\n")

var_x <- 34.5
cat("  Now the class of var_x is ",class(var_x),"\n")

var_x <- 27L
cat("   Next the class of var_x becomes ",class(var_x),"\n")
```

OUTPUT

```
The class of var_x is character
  Now the class of var_x is numeric
    Next the class of var_x becomes integer
```

Finding Variables

- To know all the variables currently available in the workspace we use ***ls()*** function.
- Also the ***ls()*** function can use patterns to match the variable names.

```
print(ls())
```

OUTPUT

```
[1] "my var"     "my_new_var" "my_var"     "var.1"
[5] "var.2"      "var.3"      "var.name"   "var_name2."
[9] "var_x"      "varname"
```

Note – It is a sample output depending on what variables are declared in your environment.

® The ls() function can use patterns to match the variable names.

```
# List the variables starting with the pattern "var".
print(ls(pattern = "var"))
```

OUTPUT

```
[1] "my var"     "my_new_var" "my_var"     "var.1"
[5] "var.2"      "var.3"      "var.name"   "var_name2."
[9] "var_x"      "varname"
```

- The variables starting with **dot(.)** are hidden, they can be listed using "all.names = TRUE" argument to ls() function.

```
print(ls(all.name = TRUE))
```

OUTPUT

```
[1] ".cars"        ".Random.seed" ".var_name"    ".varname"     ".varname2"
[6] "my var"       "my_new_var"   "my_var"       "var.1"        "var.2"
[11]"var.3"        "var.name"     "var_name2."   "var_x"
```

Deleting Variables

® Variables can be deleted by using the **rm()** function.

® Below we delete the variable var.3.

® On printing the value of the variable error is thrown.

```
rm(var.3)
print(var.3)
```

OUTPUT

```
[1] "var.3"
Error in print(var.3) : object 'var.3' not found
```

All the variables can be deleted by using the **rm()** and **ls()** function together.

```
rm(list = ls())
print(ls())
```

OUTPUT

```
character(0)
```

Operators

® An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. R language is rich in built-in operators.

® ***Types of Operators***

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

Arithmetic Operators

Following table shows the arithmetic operators supported by R language.
The operators act on each element of the vector.

Operator	Description	Example	
+	Adds two vectors	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v+t)	OUTPUT [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)	OUTPUT [1] -6.0 2.5 2.0
*	Multiplies both vectors	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v*t)	OUTPUT [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)	OUTPUT [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)	OUTPUT [1] 2.0 2.5 2.0
%/%	The result of division of first vector with second (quotient)	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v%/%t)	OUTPUT [1] 0 1 1
^	The first vector raised to the exponent of second vector	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v^t)	OUTPUT [1] 256.000 166.375 1296.000

Relational Operators

- ® Following table shows the relational operators supported by R language.
- ® Each element of 1ST vector is compared with the corresponding element of 2ND vector.
- ® The result of comparison is a *Boolean value.*

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)	OUTPUT [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)	OUTPUT [1] TRUE FALSE TRUE FALSE

Operator	Description	Example	
==	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)	***OUTPUT*** [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)	***OUTPUT*** [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)	***OUTPUT*** [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)	***OUTPUT*** [1] TRUE TRUE TRUE FALSE

Logical Operators

- Following table shows the logical operators supported by R language.
- It is applicable only to vectors of type logical, numeric or complex.
- All numbers greater than 1 are considered as logical value TRUE.
- Each element of 1ST vector is compared with the corresponding element of 2ND vector.
- The result of comparison is a Boolean value.

Operator	Description	Example	
&	It is called Element-wise Logical AND operator. It combines each element of 1ST vector with the corresponding element of the 2ND 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)	***OUTPUT*** [1] TRUE TRUE FALSE TRUE
|	It is called Element-wise Logical OR operator. It combines each element of 1ST vector with corresponding element of 2ND vector and gives an 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)	***OUTPUT*** [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)	***OUTPUT*** [1] FALSE TRUE FALSE FALSE

The logical operator considers only the first element and give single element as output.

<table>
<tr><th>Operator</th><th>Description</th><th>Example</th><th></th></tr>
<tr><td>&&</td><td>Called Logical AND operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE.</td><td>v <- c(3,0,TRUE,2+2i)
t <- c(1,3,TRUE,2+3i)
print(v&&t)</td><td><u>OUTPUT</u>
[1] TRUE</td></tr>
<tr><td>||</td><td>Called Logical OR operator. Takes first element of both the vectors and gives the TRUE if one of them is TRUE.</td><td>v <- c(0,0,TRUE,2+2i)
t <- c(0,3,TRUE,2+3i)
print(v||t)</td><td><u>OUTPUT</u>
[1] FALSE</td></tr>
</table>

Assignment Operators

These operators are used to assign values to vectors.

Operator	Description	Example	
<− = <<−	Called Left Assignment	v1 <- c(3,1,TRUE,2+3i) v2 <<- c(3,1,TRUE,2+3i) print(v1) print(v2)	OUTPUT [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)	OUTPUT [1] 3+0i 1+0i 1+0i 2+3i [1] 3+0i 1+0i 1+0i 2+3i

Miscellaneous Operators

These operators are used to for specific purpose and not general mathematical or logical computation.

Operator	Description	Example	
:	Colon operator. It creates the series of numbers in sequence for a vector.	v <- 2:8 print(v)	OUTPUT [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)	OUTPUT [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)	OUTPUT [,1] [,2] [1,] 65 82 [2,] 82 117

Decision Making

® Decision making specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is **true**, and optionally, other statements to be executed if the condition is **false**.

® R provides the following types of decision making statements.

1. ***if statement***
2. ***if...else statement***
3. ***The if...else if...else Statement***
4. ***switch statement***

1. *if statement*

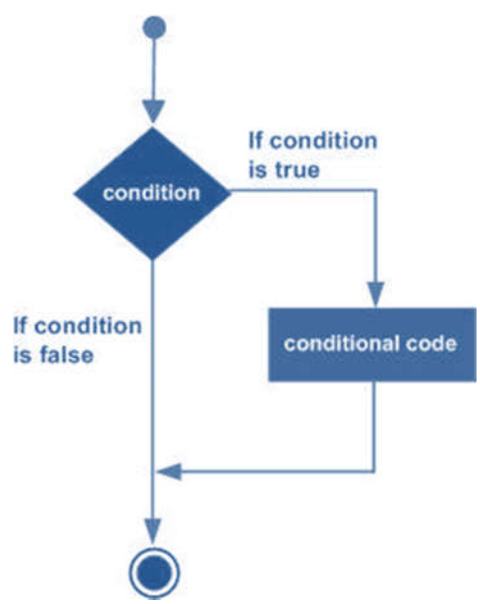
An **if** statement consists of a Boolean expression followed by one or more statements.

Syntax

```
if(boolean_expression)
{
        // statement(s) will execute if the Boolean expression is true.
}
```

- If the expression is ***true***, then '***if block'*** statement will be executed.
- Otherwise, the first set of code after the end of '***if*** 'statement will be executed.

Flow Diagram



Example

```
x <- 30L
if(is.integer(x))
{
        print("X is an Integer")
}
```

OUTPUT

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

2. ***if...else statement***

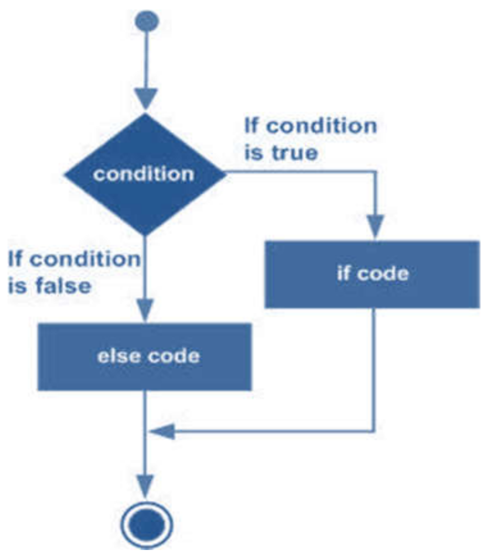
An *'if'* statement can be followed by an optional **else** statement which executes when the Boolean expression is false.

Syntax

```
if(boolean_expression)
{
        // statement(s) will execute if the boolean expression is true.
}
else
{
        // statement(s) will execute if the boolean expression is false.
}
```

- If the Boolean expression evaluates to be ***true***, then the ***'if' block*** of code will be executed, otherwise ***else block*** of code will be executed.

Flow Diagram



Example

```
x <- c("what","is","truth")
if("Truth" %in% x)
{
        print("Truth is found")
}
else
{
        print("Truth is not found")
}
```

OUTPUT

```
[1] "Truth is not found"
```

- Here "Truth" and "truth" are two different strings.

3. *The if...else if...else Statement*

- An **if** statement can be followed by an optional ***else if...else*** statement, which is very useful to test various conditions using single ***if...else if*** statement.
- When using **if**, **else if**, **else** statements there are few points to keep in mind.
    - An **if** can have zero or one **else** and it must come after any **else if**'s.
    - An **if** can have zero to many **else if's** and they must come before the else.
    - Once an **else if** succeeds, none of the remaining **else if**'s or **else**'s will be tested.

Syntax

```
if(boolean_expression 1)
{
        // Executes when the boolean expression 1 is true.
}
else if( boolean_expression 2)
{
        // Executes when the boolean expression 2 is true.
}
else if( boolean_expression 3)
{
        // Executes when the boolean expression 3 is true.
}
else
{
        // executes when none of the above condition is true.
}
```

Example

```
x <- c("what","is","truth")
if("Truth" %in% x)
{
        print("Truth is found the first time")
}
else if ("truth" %in% x)
{
        print("truth is found the second time")
}
else
{
        print("No truth found")
}
```

OUTPUT

```
[1] "truth is found the second time"
```

4. *switch statement*

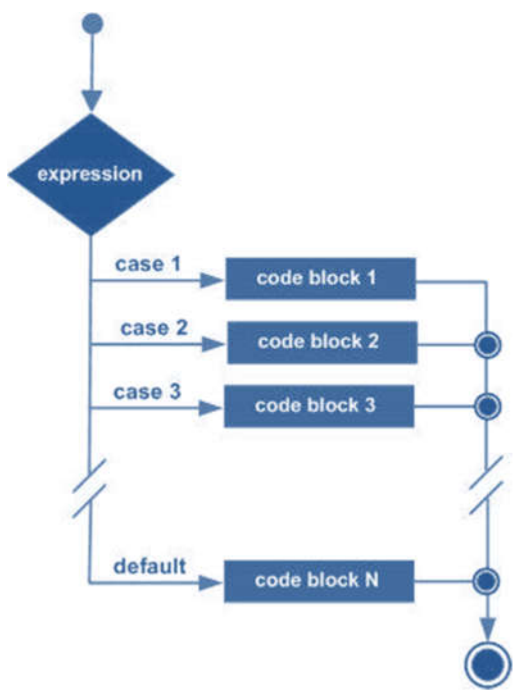
- ® A **switch** statement allows a variable to be tested for equality against a list of values.
- ® Each value is called a case, and the variable being switched on is checked for each case.

Syntax

```
switch(
        expression,
        case1,
        case2,
        case3
        ....)
```

- The following are the rules apply to a switch statement
  - ® If the value of expression is not a character string it is coerced to integer.
  - ® You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
  - ® If the value of the integer is between 1 and nargs()−1 (The max number of arguments) then the corresponding element of case condition is evaluated and the result returned.
  - ® If expression evaluates to a character string then that string is matched (exactly) to the names of the elements.
  - ® If there is more than one match, the first matching element is returned.
  - ® No Default argument is available.
  - ® In the case of no match, if there is a unnamed element of ... its value is returned. (If there is more than one such argument an error is returned.)

Flow Diagram



Example

```
x <- switch(
    3,
    "first",
    "second",
    "third",
    "fourth" )
print(x)
```

OUTPUT

```
[1] "third"
```

Loops

® In general, statements are executed sequentially. There may be a situation when you need to execute a block of code several number of times.
® The 1ST statement in a function is executed first, followed by the second, and so on.
® A loop statement allows us to execute a statement or group of statements multiple times.
® R programming language provides the following kinds of loop to handle looping requirements.
    1. ***repeat loop***
    2. ***while loop***
    3. ***for loop***

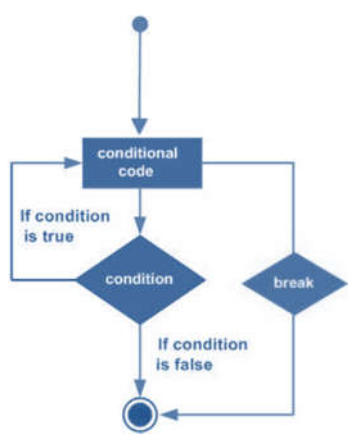
1. *repeat loop*

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

Syntax

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

Flow Diagram



Example

```
v <- c("Hello","loop")
cnt <- 2
repeat
{
        print(v)
        cnt <- cnt+1
        if(cnt > 5)
        {
                break
        }
}
```

OUTPUT

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

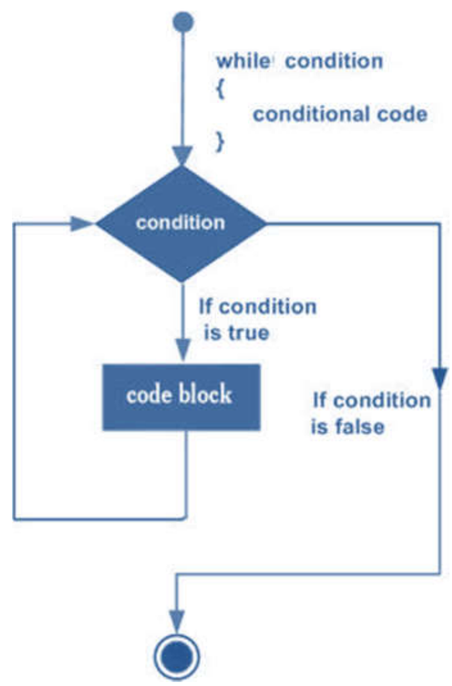
2. *While loop*

Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

Syntax

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

Flow Diagram



- Here key point of the **while** loop is that the loop might not ever run.
- When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
v <- c("Hello","while loop")
cnt <- 2
while (cnt < 7)
{
        print(v)
        cnt = cnt + 1
}
```

OUTPUT

```
[1] "Hello"  "while loop"
[1] "Hello"  "while loop"
[1] "Hello"  "while loop"
[1] "Hello"  "while loop"
[1] "Hello"  "while loop"
```

3. *For loop*

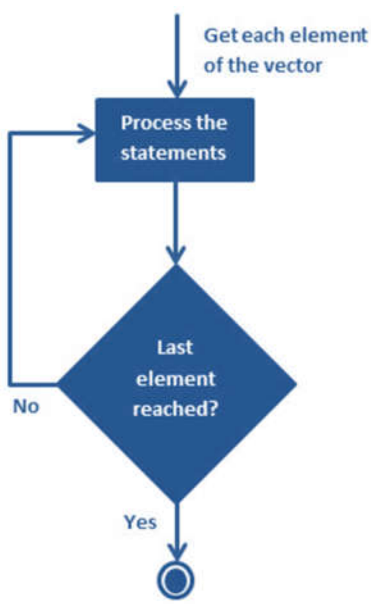
A **For loop** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Like a while statement, except that it tests the condition at the end of the loop body.

Syntax

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

Flow Diagram



- R's for loops are particularly flexible in that they are not limited to integers, or even numbers in the input.
- We can pass character vectors, logical vectors, lists or expressions.

Example

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

OUTPUT

```
[1] "A"
[1] "B"
[1] "C"
[1] "D"
```

Loop Control Statements

® They change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

® R supports the following control statements.

1. *break statement*
2. *Next statement*

1. *break statement*

® Terminates the **loop** statement and transfers execution to the statement immediately following the loop.

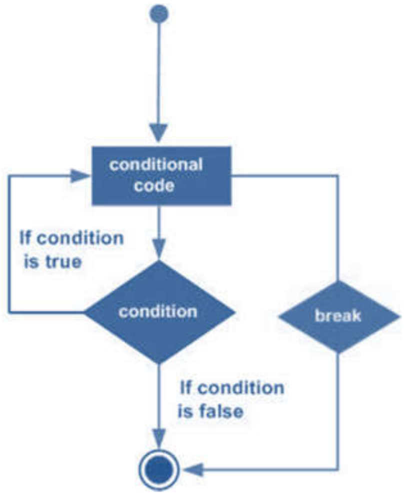
® The break statement has the following two usages

1. When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
2. It can be used to terminate a case in the switch statement.

Syntax

break

Flow Diagram



Example

```
v <- c("Hello","loop")
cnt <- 2
repeat
{
    print(v)
    cnt <- cnt + 1
    if(cnt > 4)
    {
        break
    }
}
```

OUTPUT

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

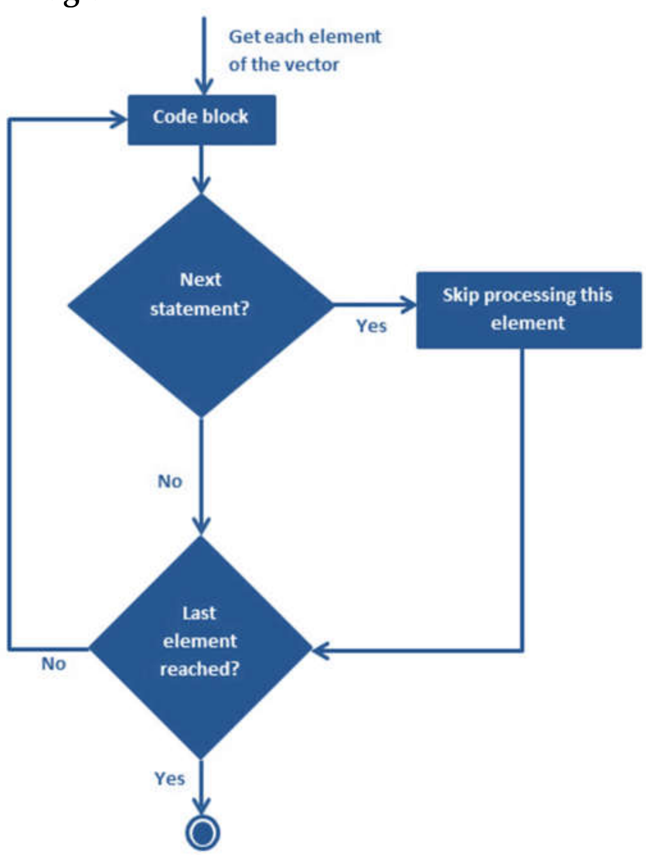
2. ***Next statement***

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

Syntax

next

Flow Diagram



Example

```
v <- LETTERS[1:6]
for ( i in v)
{
        if (i == "D")
        {
                next
        }
        print(i)
}
```

OUTPUT

```
        [1] "A"
        [1] "B"
        [1] "C"
        [1] "E"
        [1] "F"
```

Functions

- A function is a set of statements organized together to perform a specific task.
- R has in-built functions and the user can create their own functions.
- In R, a function is *an object* so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.
- The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

Definition

- An R function is created by using the keyword **function**.
- **Syntax**

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

Components

The different parts of a function are −

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

Built-in Function

- Simple examples of in-built functions are **seq()**, **mean()**, **max()**, **sum(x)** and **paste(...)** etc.
- They are directly called by user written programs.

```
# Create a sequence of numbers from 32 to 44.
print( seq (32,44) )

# Find mean of numbers from 25 to 82.
print( mean (25:82) )

# Find sum of numbers from 41 to 68.
print( sum (41:68) )
```

OUTPUT

```
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44
[1] 53.5
[1] 1526
```

User-defined Function

- They are specific to what a user wants and once created they can be used like the built-in functions.

1. # Create a function to print squares of numbers in sequence.

```
new.function <- function(a)
        {
            for(i in 1:a)
            {
                b <- i^2
                print(b)
            }
        }
# Calling a function with an argument
new.function(4)
```

OUTPUT

```
[1] 1
[1] 4
[1] 9
[1] 16
```

2. # Create a function without an argument

```
new.function <- function()
        {
            for(i in 2:4)
            {
                print(i^2)
            }
        }
# Calling a function without an argument
new.function()
```

OUTPUT

```
[1] 4
[1] 9
[1] 16
```

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

   The arguments to a function call can be supplied as defined or in a different sequence but assigned to the names of the arguments

```
# Create a function with arguments.
new.function <- function(a,b,c)
        {
            result <- a * b + c
            print(result)
        }
```

```
# Call the function by position of arguments
new.function (5, 3, 11)
# Call the function by names of the arguments
new.function (a = 11, b = 5, c = 3)
```

OUTPUT

```
[1] 26
[1] 58
```

4. *Calling a Function with Default Argument*

   We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non default result.

```
# Create a function with arguments
new.function <- function (a = 3, b = 6)
        {
            result <- a * b
            print(result)
        }
# Call the function without giving any argument
new.function ()
# Call the function with giving new values of the argument.
new.function (9, 5)
```

OUTPUT

```
[1] 18
[1] 45
```

5. *Lazy Evaluation of Function*

   Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.

```
# Create a function with arguments.
new.function <- function(a, b)
        {
            print(a^2)
            print(a)
            print(b)
        }
# Evaluate the function without supplying one of the arguments.
new.function(6)
```

OUTPUT

```
[1] 36
[1] 6
Error in print (b): argument "b" is missing, with no default
```

String

® Any value written within a pair of single quote or double quotes is treated as a string. Internally R stores every string within double quotes, even when you create them with single quote.

Rules Applied in String Construction

® The quotes at the beginning and end of a string should be double quotes or both single quote. They cannot be mixed.
® Double quotes can be inserted into a string starting and ending with single quote.
® Single quote can be inserted into a string starting and ending with double quotes.
® Double quotes cannot be inserted into a string starting and ending with double quotes.
® Single quote cannot be inserted into a string starting and ending with single quote.

Examples of Valid Strings

1. a <- 'Start and end with single quote'
   print(a)
2. b <- "Start and end with double quotes"
   print(b)
3. c <- "single quote ' in between double quotes"
   print(c)
4. d <- 'Double quotes " in between single quote'
   print(d)

OUTPUT

[1] "Start and end with single quote"
[1] "Start and end with double quotes"
[1] "single quote ' in between double quote"
[1] "Double quote \" in between single quote"

Examples of Invalid Strings

1. e <- 'Mixed quotes"
   print(e)
2. f <- 'Single quote ' inside single quote'
   print(f)
3. g <- "Double quotes " inside double quotes"
   print(g)
   When we run the script it fails giving below results.
   Error: unexpected symbol in:
4. "print(e)
   f <- 'Single"
   Execution halted

String Manipulation

1. ***Concatenating Strings* - paste() function**

   Many strings in R are combined using the **paste()** function. It can take any number of arguments to be combined together.

   Syntax

   paste(..., sep = " ", collapse = NULL)

   - **...** represents any number of arguments to be combined.
   - ***sep*** represents any separator between the arguments. It is optional.
   - ***collapse*** is used to eliminate the space in between two strings. But not the space within two words of one string.

   Example

```
a <- "Hello"
b <- 'How'
c <- "are you? "
print (paste(a,b,c))
print (paste(a,b,c, sep = "-"))
print (paste(a,b,c, sep = "", collapse = ""))
```

   OUTPUT

```
[1] "Hello How are you? "
[1] "Hello-How-are you? "
[1] "HelloHoware you? "
```

2. ***Formatting numbers & strings* - format() function**

   Numbers and strings can be formatted **format()** function.

   Syntax

   format (x, digits, nsmall, scientific, width, justify = c("left", "right", "centre", "none"))

   - ***x*** is the vector input.
   - ***digits*** is the total number of digits displayed.
   - ***nsmall*** is the minimum number of digits to the right of the decimal point.
   - ***scientific*** is set to TRUE to display scientific notation.
   - ***width*** indicates the minimum width to be displayed by padding blanks in the beginning.
   - ***i*** is the display of the string to left, right or center.

   Example

```
# Total number of digits displayed. Last digit rounded off.
result <- format(23.123456789, digits = 9)
print(result)
# Display numbers in scientific notation.
result <- format(c(6, 13.14521), scientific = TRUE)
print(result)
# The minimum number of digits to the right of the decimal point.
result <- format(23.47, nsmall = 5)
print(result)
```

```
# Format treats everything as a string.
result <- format(6)
print(result)
# Numbers are padded with blank in the beginning for width.
result <- format(13.7, width = 6)
print(result)
# Left justify strings.
result <- format("Hello", width = 8, justify = "l")
print(result)
# Justfy string with center.
result <- format("Hello", width = 8, justify = "c")
print(result)
```

OUTPUT

```
[1] "23.1234568"
[1] "6.000000e+00" "1.314521e+01"
[1] "23.47000"
[1] "6"
[1] "  13.7"
[1] "Hello   "
[1] " Hello  "
```

3. ***Counting number of characters in a string*** **- nchar() function**

   This function counts the number of characters including spaces in a string.

   Syntax

   nchar(x)

   - **x** is the vector input.

   Example

   ```
   result <- nchar("Ibramsha")
   print(result)
   ```

   OUTPUT

   ```
   [1] 8
   ```

4. ***Changing the case*** **- toupper() & tolower() functions**

   These functions change the case of characters of a string.

   Syntax

   toupper(x), tolower(x)

   - **x** is the vector input.

   Example

   1. `result <- toupper("saq")`
      `print(result)`
   2. `result <- tolower("MIS")`
      `print(result)`

      OUTPUT

      ```
      [1] "SAQ"
      [1] "mis"
      ```

5. ***Extracting parts of a string* - substring() function**
   This function extracts parts of a String.

   Syntax

   substring (x, first, last)

   - **x** is the character vector input.
   - **first** is the position of the first character to be extracted.
   - **last** is the position of the last character to be extracted.

   Example

```
# Extract characters from 5th to 7th position.
result <- substring("Extract", 5, 7)
print(result)
```

   OUTPUT

```
[1] "act"
```

Vector

Vectors are the most basic R data objects and there are six types of atomic vectors. They are logical, integer, double, complex, character and raw.

Vector Creation

Single Element Vector

® Even when you write just one value in R, it becomes a vector of length 1 and belongs to one of the above vector types.

```
# Atomic vector of type character.
print("abc");

# Atomic vector of type double.
print(12.5)

# Atomic vector of type integer.
print(63L)

# Atomic vector of type logical.
print(TRUE)

# Atomic vector of type complex.
print(2+3i)

# Atomic vector of type raw.
print(charToRaw('hello'))
```

OUTPUT

```
[1] "abc"
[1] 12.5
[1] 63
[1] TRUE
[1] 2+3i
[1] 68 65 6c 6c 6f
```

Multiple Elements Vector

1. *Using colon operator with numeric data*

```
# Creating a sequence from 5 to 13.
v <- 5:13
print(v)

# Creating a sequence from 6.6 to 12.6.
v <- 6.6:12.6
print(v)

# If the final element specified does not belong to the sequence then it is discarded.
v <- 3.8:11.4
print(v)
```

OUTPUT

```
[1]  5  6  7  8  9 10 11 12 13
[1]  6.6  7.6  8.6  9.6 10.6 11.6 12.6
[1]  3.8  4.8  5.8  6.8  7.8  8.8  9.8 10.8
```

2. *Using sequence (Seq.) operator*

```
# Create vector with elements from 5 to 9 incrementing by 0.4.
print(seq(5, 9, by = 0.4))
```

OUTPUT

```
[1] 5.0 5.4 5.8 6.2 6.6 7.0 7.4 7.8 8.2 8.6 9.0
```

3. *Using the c() function*

The non-character values are coerced to character type if one of the elements is a character.

```
# The logical and numeric values are converted to characters.
s <- c('apple','red',5,TRUE)
print(s)
```

OUTPUT

```
[1] "apple" "red"   "5"     "TRUE"
```

Accessing Vector Elements

- Elements of a Vector are accessed using indexing.
- The **[ ] brackets** are used for indexing.
- Indexing starts with position 1. Giving a negative value in the index drops that element from result.
- **TRUE**, **FALSE** or **0** and **1** can also be used for indexing.

```
# Accessing vector elements using position.
t <- c("Sun","Mon","Tue","Wed","Thurs","Fri","Sat")
u <- t[c(2,3,6)]
print(u)
```

```
# Accessing vector elements using logical indexing.
v <- t[c(TRUE,FALSE,FALSE,FALSE,FALSE,TRUE,FALSE)]
print(v)

# Accessing vector elements using negative indexing.
x <- t[c(-2,-5)]
print(x)

# Accessing vector elements using 0/1 indexing.
y <- t[c(0,0,0,0,0,0,1)]
print(y)
```

OUTPUT

```
[1] "Mon" "Tue" "Fri"
[1] "Sun" "Fri"
[1] "Sun" "Tue" "Wed" "Fri" "Sat"
[1] "Sun"
```

Vector Manipulation

1. ***Vector arithmetic***

Two vectors of same length can be added, subtracted, multiplied or divided giving the result as a vector output.

```
# Create two vectors.
v1 <- c(3,8,4,5,0,11)
v2 <- c(4,11,0,8,1,2)

# Vector addition.
add.result <- v1+v2
print(add.result)

# Vector subtraction.
sub.result <- v1-v2
print(sub.result)

# Vector multiplication.
multi.result <- v1*v2
print(multi.result)

# Vector division.
divi.result <- v1/v2
print(divi.result)
```

OUTPUT

```
[1]  7 19  4 13  1 13
[1] -1 -3  4 -3 -1  9
[1] 12 88  0 40  0 22
[1] 0.7500000 0.7272727       Inf 0.6250000 0.0000000 5.5000000
```

2. ***Vector Element Recycling***

If we apply arithmetic operations to two vectors of unequal length, then the elements of the shorter vector are recycled to complete the operations.

```
v1 <- c(3,8,4,5,0,11)
v2 <- c(4,11)            # V2 becomes c(4,11,4,11,4,11)
add.result <- v1+v2
print(add.result)
```

OUTPUT

```
[1]  7 19  8 16  4 22
```

3. ***Vector Element Sorting***

Elements in a vector can be sorted using the **sort()** function.

```
v <- c(3,8,4,5,0,11, -9, 304)

# Sort the elements of the vector.
sort.result <- sort(v)
print(sort.result)

# Sort the elements in the reverse order.
revsort.result <- sort(v, decreasing = TRUE)
print(revsort.result)

# Sorting character vectors.
v <- c("Red","Blue","yellow","violet")
sort.result <- sort(v)
print(sort.result)

# Sorting character vectors in reverse order.
revsort.result <- sort(v, decreasing = TRUE)
print(revsort.result)
```

OUTPUT

```
[1]  -9   0   3   4   5   8  11 304
[1] 304  11   8   5   4   3   0  -9
[1] "Blue"   "Red"    "violet" "yellow"
[1] "yellow" "violet" "Red"    "Blue"
```

Lists

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

Creating a List

```
# Create a list containing strings, numbers, vectors and a logical values.
list_data <- list("Red", c(21,32,11), TRUE, 51.23, 119.1)
print(list_data)
```

OUTPUT

```
[[1]]
[1] "Red"
[[2]]
[1] 21 32 11
[[3]]
[1] TRUE
[[4]]
[1] 51.23
```

Naming List Elements

The list elements can be given names and they can be accessed using these names.

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
list("green",12.3))

# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

# Show the list.
print(list_data)
```

OUTPUT

```
$`1st_Quarter`
[1] "Jan" "Feb" "Mar"

$A_Matrix
    [,1] [,2] [,3]
[1,]   3    5   -2
[2,]   9    1    8

$A_Inner_list
$A_Inner_list[[1]]
[1] "green"

$A_Inner_list[[2]]
[1] 12.3
```

Accessing List Elements

Elements of the list can be accessed by the index of the element in the list. In case of named lists it can also be accessed using the names.

We continue to use the list in the above example –

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
list("green",12.3))

# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

# Access the first element of the list.
print(list_data[1])

# Access the thrid element. As it is also a list, all its elements will be printed.
print(list_data[3])

# Access the list element using the name of the element.
print(list_data$A_Matrix)
```

OUTPUT

```
$`1st_Quarter`
[1] "Jan" "Feb" "Mar"

$A_Inner_list
$A_Inner_list[[1]]
[1] "green"

$A_Inner_list[[2]]
[1] 12.3

     [,1] [,2] [,3]
[1,]    3    5   -2
[2,]    9    1    8
```

Manipulating List Elements

We can add and delete elements only at the end of a list.
But we can update any element.

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
list("green",12.3))

# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

# Add element at the end of the list.
list_data[4] <- "New element"
print(list_data[4])

# Remove the last element.
list_data[4] <- NULL

# Print the 4th Element.
print(list_data[4])

# Update the 3rd Element.
list_data[3] <- "updated element"
print(list_data[3])
```

OUTPUT

```
[[1]]
[1] "New element"

$<NA>
NULL

$`A Inner list`
[1] "updated element"
```

Merging Lists

You can merge many lists into one list by placing all the lists inside one list() function.

```
# Create two lists.
list1 <- list(1,2,3)
list2 <- list("Sun","Mon","Tue")
# Merge the two lists.
merged.list <- c(list1,list2)
# Print the merged list.
print(merged.list)
```

OUTPUT

```
[[1]]
[1] 1
[[2]]
[1] 2
[[3]]
[1] 3
[[4]]
[1] "Sun"
[[5]]
[1] "Mon"
[[6]]
[1] "Tue"
```

Converting List to Vector

A list can be converted to a vector so that the elements of the vector can be used for manipulation. All the arithmetic operations on vectors can be applied after the list is converted into vectors ***unlist()*** function. It takes the list as input and produces a vector.

```
# Create lists.
list1 <- list(1:5)
print(list1)
list2 <-list(10:14)
print(list2)
# Convert the lists to vectors.
v1 <- unlist(list1)
v2 <- unlist(list2)
print(v1)
print(v2)
# Now add the vectors
result <- v1+v2
print(result)
```

OUTPUT

```
[[1]]
[1] 1 2 3 4 5
[[1]]
[1] 10 11 12 13 14
[1] 1 2 3 4 5
[1] 10 11 12 13 14
[1] 11 13 15 17 19
```

Matrices

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

® Though we can create a matrix containing only characters or only logical values, they are not of much use. We use matrices containing numeric elements to be used in mathematical calculations.

® A Matrix is created using the **matrix()** function.

Syntax

matrix(data, nrow, ncol, byrow, dimnames)

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

Example

Create a matrix taking a vector of numbers as input.

```
# Elements are arranged sequentially by row.
M <- matrix(c(3:14), nrow = 4, byrow = TRUE)
print(M)

# Elements are arranged sequentially by column.
N <- matrix(c(3:14), nrow = 4, byrow = FALSE)
print(N)

# Define the column and row names.
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")

P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames,
colnames))
print(P)
```

OUTPUT

```
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    7    8
[3,]    9   10   11
[4,]   12   13   14
```

```
     [,1] [,2] [,3]
[1,]    3    7   11
[2,]    4    8   12
[3,]    5    9   13
[4,]    6   10   14
```

```
     col1 col2 col3
row1    3    4    5
row2    6    7    8
row3    9   10   11
row4   12   13   14
```

Accessing Elements of a Matrix

Elements of a matrix can be accessed by using the column and row index of the element. We consider the matrix P above to find the specific elements below.

```
# Define the column and row names.
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")

# Create the matrix.
P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames, colnames))

# Access the element at 3rd column and 1st row.
print(P[1,3])

# Access the element at 2nd column and 4th row.
print(P[4,2])

# Access only the  2nd row.
print(P[2,])

# Access only the 3rd column.
print(P[,3])
```

OUTPUT

```
[1] 5
[1] 13
col1 col2 col3
  6    7    8
row1 row2 row3 row4
  5    8   11   14
```

Matrix Computations

® Various mathematical operations are performed on the matrices using the R operators. The result of the operation is also a matrix.

® The dimensions (number of rows and columns) should be same for the matrices involved in the operation.

Matrix Addition & Subtraction

```
# Create two 2x3 matrices.
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2)
print(matrix1)
matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2)
print(matrix2)
# Add the matrices.
result <- matrix1 + matrix2
cat("Result of addition","\n")
print(result)
```

```
# Subtract the matrices
result <- matrix1 - matrix2
cat("Result of subtraction","\n")
print(result)
```

OUTPUT

```
     [,1] [,2] [,3]
[1,]    3   -1    2
[2,]    9    4    6
Result of addition
     [,1] [,2] [,3]
[1,]    8   -1    5
[2,]   11   13   10
```

```
     [,1] [,2] [,3]
[1,]    5    0    3
[2,]    2    9    4
Result of subtraction
     [,1] [,2] [,3]
[1,]   -2   -1   -1
[2,]    7   -5    2
```

Matrix Multiplication & Division

```
# Create two 2x3 matrices.
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2)
print(matrix1)
matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2)
print(matrix2)

# Multiply the matrices.
result <- matrix1 * matrix2
cat("Result of multiplication","\n")
print(result)

# Divide the matrices
result <- matrix1 / matrix2
cat("Result of division","\n")
print(result)
```

OUTPUT

```
     [,1] [,2] [,3]
[1,]    3   -1    2
[2,]    9    4    6
Result of multiplication
     [,1] [,2] [,3]
[1,]   15    0    6
[2,]   18   36   24
```

```
     [,1] [,2] [,3]
[1,]    5    0    3
[2,]    2    9    4
Result of division
     [,1]      [,2]      [,3]
[1,]  0.6      -Inf 0.6666667
[2,]  4.5 0.4444444 1.5000000
```

Arrays

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

Example

The following example creates an array of two 3x3 matrices each with 3 rows and 3 columns.

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)

# Take these vectors as input to the array.
result <- array(c(vector1,vector2),dim = c(3,3,2))
print(result)
```

OUTPUT

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

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

Naming Columns and Rows

We can give names to the rows, columns and matrices in the array by using the ***dimnames*** parameter.

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
column.names <- c("COL1","COL2","COL3")
row.names <- c("ROW1","ROW2","ROW3")
matrix.names <- c("Matrix1","Matrix2")

# Take these vectors as input to the array.
result <- array(c(vector1,vector2), dim = c(3,3,2),
          dimnames = list(row.names,column.names,matrix.names))
print(result)
```

OUTPUT

```
, , Matrix1
     COL1 COL2 COL3
ROW1    5   10   13
ROW2    9   11   14
ROW3    3   12   15

, , Matrix2
     COL1 COL2 COL3
ROW1    5   10   13
ROW2    9   11   14
ROW3    3   12   15
```

Accessing Array Elements

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
column.names <- c("COL1","COL2","COL3")
row.names <- c("ROW1","ROW2","ROW3")
matrix.names <- c("Matrix1","Matrix2")

# Take these vectors as input to the array.
result <- array(c(vector1,vector2),dim = c(3,3,2),dimnames = list(row.names,
  column.names, matrix.names))

# Print the third row of the second matrix of the array.
print(result[3,,2])

# Print the element in the 1st row and 3rd column of the 1st matrix.
print(result[1,3,1])

# Print the 2nd Matrix.
print(result[,,2])
```

OUTPUT

```
COL1 COL2 COL3
  3   12   15
[1] 13

     COL1 COL2 COL3
ROW1    5   10   13
ROW2    9   11   14
ROW3    3   12   15
```

Manipulating Array Elements

As array is made up matrices in multiple dimensions, the operations on elements of array are carried out by accessing elements of the matrices.

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)

# Take these vectors as input to the array.
array1 <- array(c(vector1,vector2),dim = c(3,3,2))

# Create two vectors of different lengths.
vector3 <- c(9,1,0)
vector4 <- c(6,0,11,3,14,1,2,6,9)
array2 <- array(c(vector1,vector2),dim = c(3,3,2))

# create matrices from these arrays.
matrix1 <- array1[,,2]
matrix2 <- array2[,,2]
```

```
# Add the matrices.
result <- matrix1+matrix2
print(result)
```

OUTPUT

```
     [,1] [,2] [,3]
[1,]   10   20   26
[2,]   18   22   28
[3,]    6   24   30
```

Calculations Across Array Elements

We can do calculations across the elements in an array using the **apply()** function.

Syntax

apply(x, margin, fun)

- **x** is an array.
- **margin** is the name of the data set used.
- **fun** is the function to be applied across the elements of the array.

Example

We use the apply() function below to calculate the sum of the elements in the rows of an array across all the matrices.

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)

# Take these vectors as input to the array.
new.array <- array(c(vector1,vector2),dim = c(3,3,2))
print(new.array)

# Use apply to calculate the sum of the rows across all the matrices.
result <- apply(new.array, c(1), sum)
print(result)
```

OUTPUT

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

[1] 56 68 60
```