# R - 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. 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

The basic syntax for creating a matrix in R is −

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

Following is the description of the parameters used −

* **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.

# Create a matrix.

a\_matrix = matrix( c('a','b','c','d','e','f','g','h','i'), nrow = 3, ncol = 3, byrow=TRUE)

print(a\_matrix)

print(class(a\_matrix))

When we execute the above code, it produces the following result –

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

[1,] "a" "b" "c"

[2,] "d" "e" "f"

[3,] "g" "h" "i"

[1] "matrix"

b\_matrix = matrix( c('a','b','c','d','e','f','g','h','i'), nrow = 3, ncol = 3, byrow=FALSE)

print(b\_matrix)

print(class(b\_matrix))

When we execute the above code, it produces the following result –

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

[1,] "a" "d" "g"

[2,] "b" "e" "h"

[3,] "c" "f" "i"

[1] "matrix"

b\_matrix <- matrix (1:16, nrow = 4, ncol=4 , byrow = TRUE,

dimnames = list( c("one", "two", "three", "four"), c("East", "West", "North", "South") ) )

print(b\_matrix)

print(class(b\_matrix))

East West North South

one 1 2 3 4

two 5 6 7 8

three 9 10 11 12

four 13 14 15 16

[1] "matrix"

c\_matrix <- matrix(1:16, nrow = 4, ncol = 4, byrow = TRUE,

dimnames = list(NULL, c("North","East","South","West")))

c\_matrix

North East South West

[1,] 1 2 3 4

[2,] 5 6 7 8

[3,] 9 10 11 12

[4,] 13 14 15 16

dimnames(c\_matrix) <- list(c("Row1","Row2","Row3","Row4"))

c\_matrix

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

Row1 1 2 3 4

Row2 5 6 7 8

Row3 9 10 11 12

Row4 13 14 15 16

dimnames(c\_matrix) <- list(c("Row1","Row2","Row3","Row4"),c("North","East","South","West"))

c\_matrix

North East South West

Row1 1 2 3 4

Row2 5 6 7 8

Row3 9 10 11 12

Row4 13 14 15 16

**Use of matrix or array functions**

# Create a matrix.

a\_matrix <- matrix(1:16, nrow = 4, ncol = 4,

dimnames = list( c("one", "two", "three", "four"), c("east", "west", "north","south") ) )

print(a\_matrix)

print(class(a\_matrix))

east west north south

one 1 5 9 13

two 2 6 10 14

three 3 7 11 15

four 4 8 12 16

[1] "matrix"

dim(a\_matrix)

[1] 4 4

nrow(a\_matrix)

[1] 4

ncol(a\_matrix)

[1] 4

length(a\_matrix)

[1] 16

identical(nrow(a\_matrix), ncol(a\_matrix))

[1] TRUE

rownames(a\_matrix)

[1] "one" "two" "three" "four"

colnames(a\_matrix)

[1] "east" "west" "north" "south"

dimnames(a\_matrix)

[[1]]

[1] "one" "two" "three" "four"

[[2]]

[1] "east" "west" "north" "south"

# To change the name of dimentions

dimnames(c\_matrix) = list( c("R1", "R2", "R3", "R4"), c("east", "west", "north","south") )

dimnames(c\_matrix)

[[1]]

[1] "R1" "R2" "R3" "R4"

[[2]]

[1] "east" "west" "north" "south"

**2D array is identical to matrix**

**Example 3.**

a\_matrix <- matrix(1:16, nrow = 4, ncol = 4, **byrow=TRUE,**

dimnames = list( c("one", "two", "three", "four"), c("east", "west", "north","south") ) )

print(a\_matrix)

east west north south

one 1 2 3 4

two 5 6 7 8

three 9 10 11 12

four 13 14 15 16

This matrix could also be created using the array function. The following two-dimensional array is identical to the matrix that we just created above (it even has class matrix)

two\_d\_array <- **array(** 1:16, dim = c(4, 4),

dimnames = list( c("one", "two", "three", "four"), c("East", "West", "North", "South") ) **)**

print(two\_d\_array)

print(class(two\_d\_array))

East West North South

one 1 5 9 13

two 2 6 10 14

three 3 7 11 15

four 4 8 12 16

**[1] "matrix"**

**NOTE: Here class is matrix not array because 2D array is considered as matrix.**

## Combining Matrices

The c function converts matrices to vectors before concatenating them:

a\_matrix <- matrix( 1:16, nrow = 4, ncol = 4,

dimnames = list( c("one", "two", "three", "four"), c("col1", "col2", "col3","Col4") ) )

b\_matrix <- matrix( seq.int(2, 32, 2), nrow = 4, ncol = 4,

dimnames = list( c("five", "six", "seven", "eight"), c("col5", "col6", "col7","Col8") ) )

print(b\_matrix)

col1 col2 col3 Col4

five 2 10 18 26

six 4 12 20 28

seven 6 14 22 30

eight 8 16 24 32

print(a\_matrix)

east west north south

one 1 2 3 4

two 5 6 7 8

three 9 10 11 12

four 13 14 15 16

**concatenating two matrix**

c(a\_matrix, b\_matrix)

[1] 1 5 9 13 2 6 10 14 3 7 11 15 4 8 12 16 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32

More natural combining of matrices can be achieved by using cbind and rbind, which bind matrices together by columns and rows:

cbind(a\_matrix, b\_matrix)

east west north south col1 col2 col3 Col4

one 1 2 3 4 2 10 18 26

two 5 6 7 8 4 12 20 28

three 9 10 11 12 6 14 22 30

four 13 14 15 16 8 16 24 32

rbind(a\_matrix, b\_matrix)

east west north south

one 1 2 3 4

two 5 6 7 8

three 9 10 11 12

four 13 14 15 16

five 2 10 18 26

six 4 12 20 28

seven 6 14 22 30

eight 8 16 24 32

## Arithmetic Operation On Matrices And Array

The standard arithmetic operators (+, - , \*, /) work element-wise on matrices and arrays, just they like they do on vectors:

matrix\_a <- matrix( seq.int(2, 32, 2), nrow = 4,

dimnames = list( c("One", "Two", "Three", "Four"), c("col1", "col2", "col3","Col4") ) )

matrix\_b <- matrix( seq.int(2, 32, 2), nrow = 4,

dimnames = list( c("One", "Two", "Three", "Four"), c("col1", "col2", "col3","Col4") ) )

print(matrix\_a)

col1 col2 col3 Col4

One 2 10 18 26

Two 4 12 20 28

Three 6 14 22 30

Four 8 16 24 32

print(matrix\_b)

col1 col2 col3 Col4

One 2 10 18 26

Two 4 12 20 28

Three 6 14 22 30

Four 8 16 24 32

matrix\_a + matrix\_b

col1 col2 col3 Col4

One 4 20 36 52

Two 8 24 40 56

Three 12 28 44 60

Four 16 32 48 64

matrix\_a - matrix\_b

col1 col2 col3 Col4

One 0 0 0 0

Two 0 0 0 0

Three 0 0 0 0

Four 0 0 0 0

matrix\_a \* matrix\_b

col1 col2 col3 Col4

One 4 100 324 676

Two 16 144 400 784

Three 36 196 484 900

Four 64 256 576 1024

matrix\_a / matrix\_b

col1 col2 col3 Col4

One 1 1 1 1

Two 1 1 1 1

Three 1 1 1 1

Four 1 1 1 1

**Transposes Matrices**

The t function transposes matrices (but not higher-dimensional arrays, where the concept isn’t well defined):

a\_matrix

east west north south

one 1 2 3 4

two 5 6 7 8

three 9 10 11 12

four 13 14 15 16

t(a\_matrix)

one two three four

east 1 5 9 13

west 2 6 10 14

north 3 7 11 15

south 4 8 12 16

## Arrays

Matrices are a special case of two-dimensional array. Arrays can be of any number of dimensions. An Array in R Programming is simply called as the multi-dimensional Data structure. In R Array, data is stored in matrices, row and columns, and we can access the matrix element using the matrix level, row index and column index.

Arrays 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.

**R Array Syntax**

The basic syntax of the Array in R Programming language is as shown below:

array\_name <- array(data, dim = (row\_Size, column\_Size, matrices, dimnames)

Where

**Data**: Data is a vector of values which is to be placed in matrix.

**Row\_Size**: Please specify the number of Row elements an array can store. For example, Row\_Size = 4 then each matrix in an array will have 4 rows.

**Column\_Size**: Number of Column elements an array can store. For example, Column\_Size = 3 then each matrix in an array will have 3 Columns.

**Matrices**: It will decide the number of Matrices an array can accept. Array in R is always a Multi Dimensional, it means more than 1 matrix with rows and columns.

**Dimnames**: It is used to change the default Row, Column, Matrices names to more meaningful names.

**Create an Array in R**

To create an array, you call the array function, passing in a vector of values and a vector of dimensions. Optionally, you can also provide names for each dimension:

**Example 1.**

# Create an array.

md\_array <- array( 1:24, dim = c(4,3,2), dimnames = list( c("one","Two","Three","Four"), c("Col1","Col2","Col3"), c("Matrix1","Matrix2") ) )

print(md\_array)

print(class(md\_array))

Above statement will create an Array of elements from 1 to 24 arranged in two matrices and each matrices contains four rows and three columns.

When we execute the above code, it produces the following result −

, , Matrix1

Col1 Col2 Col3

one 1 5 9

Two 2 6 10

Three 3 7 11

Four 4 8 12

, , Matrix2

Col1 Col2 Col3

one 13 17 21

Two 14 18 22

Three 15 19 23

Four 16 20 24

[1] "array"

**Example 2.**

md\_array <- array( **1:24**, dim = c(4,3,4), dimnames = list( c("one","Two","Three","Four"), c("Col1","Col2","Col3"), c("Matrix1"," Matrix2"," Matrix3"," Matrix4") ) )

print(md\_array)

print(class(md\_array))

When we execute the above code, it produces the following result −

, , Matrix1

Col1 Col2 Col3

one 1 5 9

Two 2 6 10

Three 3 7 11

Four 4 8 12

, , Matrix2

Col1 Col2 Col3

one 13 17 21

Two 14 18 22

Three 15 19 23

Four 16 20 24

, , Matrix3

Col1 Col2 Col3

one 1 5 9

Two 2 6 10

Three 3 7 11

Four 4 8 12

, , Matrix4

Col1 Col2 Col3

one 13 17 21

Two 14 18 22

Three 15 19 23

Four 16 20 24

[1] "array"

**Example 3.**

md\_array <- array( **1:48**, dim = c(4,3,4), dimnames = list( c("one","Two","Three","Four"), c("Col1","Col2","Col3"), c("First","Second","Third","Fourth") ) )

print(md\_array)

print(class(md\_array))

When we execute the above code, it produces the following result −

, , First

Col1 Col2 Col3

one 1 5 9

Two 2 6 10

Three 3 7 11

Four 4 8 12

, , Second

Col1 Col2 Col3

one 13 17 21

Two 14 18 22

Three 15 19 23

Four 16 20 24

, , Third

Col1 Col2 Col3

one 25 29 33

Two 26 30 34

Three 27 31 35

Four 28 32 36

, , Fourth

Col1 Col2 Col3

one 37 41 45

Two 38 42 46

Three 39 43 47

Four 40 44 48

> print(class(md\_array))

[1] "array"

>

**Example 4.**

# Defining Row names, Column names and matrix name in R array.

row.names <- c("Row1", "Row2", "Row3", "Row4")

column.names <- c("Col1", "Col2", "Col3")

matrix.names <- c("Matrix1", "Matrix2")

md\_array <- array(1:24, dim = c(4, 3, 2), dimnames = list(row.names, column.names, matrix.names))

print(md\_array)

When we execute the above code, it produces the following result −

, , Matrix1

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Matrix2

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

**Example 5.**

# Defining length names, bredth names, hight names and matrix name in R array.

length.names <- c("Row1", "Row2", "Row3", "Row4")

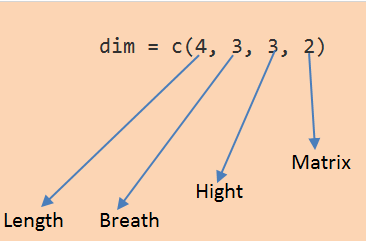
breath.names <- c("Col1", "Col2", "Col3")

hight.names <- c("Hight1","Hight2","Hight3")

matrix.names <- c("Matrix1", "Matrix2")

md\_array5 <- array(1:24, dim = c(4, 3, 3, 2), dimnames = list(length.names, breath.names, hight.names, matrix.names))

print(md\_array5)



When we execute the above code, it produces the following result −

, , Hight1, Matrix1

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Hight2, Matrix1

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

, , Hight3, Matrix1

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Hight1, Matrix2

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

, , Hight2, Matrix2

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Hight3, Matrix2

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

**Example 6.**

# Defining length names, bredth names, hight names, direction names and matrix name in R array.

length.names <- c("Row1", "Row2", "Row3", "Row4")

breath.names <- c("Col1", "Col2", "Col3")

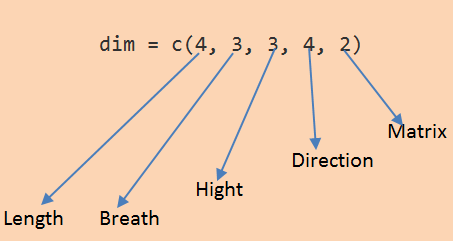
hight.names <- c("Hight1","Hight2","Hight3")

direction.names <- c("East","West","North","South")

matrix.names <-c("Matrix1", "Matrix2")

md\_array6 <- array(1:24, dim = c(4, 3, 3, 4, 2), dimnames = list(length.names, breath.names, hight.names, direction.names, matrix.names))

print(md\_array6)



When we execute the above code, it produces the following result −

, , Hight1, East, Matrix1

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Hight2, East, Matrix1

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

, , Hight3, East, Matrix1

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Hight1, West, Matrix1

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

, , Hight2, West, Matrix1

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Hight3, West, Matrix1

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

, , Hight1, North, Matrix1

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Hight2, North, Matrix1

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

, , Hight3, North, Matrix1

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Hight1, South, Matrix1

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

, , Hight2, South, Matrix1

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Hight3, South, Matrix1

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

, , Hight1, East, Matrix2

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Hight2, East, Matrix2

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

, , Hight3, East, Matrix2

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Hight1, West, Matrix2

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

, , Hight2, West, Matrix2

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Hight3, West, Matrix2

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

, , Hight1, North, Matrix2

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Hight2, North, Matrix2

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

, , Hight3, North, Matrix2

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Hight1, South, Matrix2

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

, , Hight2, South, Matrix2

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Hight3, South, Matrix2

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

## Accessing R Array Elements

In R programming, We can use the index position to access the array elements. Using index we can access or alter/change each and every individual element present in an array. Index value starts at 1 and end at n where n is the size of a matrix, row or column. The syntax behind Array accessing is: **Array\_Name[row\_position, Column\_Position, Matrix\_Level].** For example, we declared an array of two matrices of size 6 rows \* 4 columns. To access or alter 1st value use Array\_name[1, 1, 1], to access or alter 2nd row 3rd column value at 1stMatrix level then use Array\_name[2, 3, 1] and to access the 6th row 4th column in 2nd matrix level then use Array\_name[6, 4, 2]. Lets see the example for better understanding:

# Accessing R Array Elements

print(md\_array)

, , Matrixl1

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Matrix2

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

# Access the element of 1st row and 2nd column in Matrix 1.

print(md\_array[1, 2, 1])

[1] 5

# Access the element of 3rd row and 1st column in in Matrix 2.

print(md\_array[3, 1, 2])

[1] 15

# Access only the 3rd row in First Matrix.

print(md\_array[3, , 1])

Col1 Col2 Col3

3 7 11

# Access only the 3rd column in Second Matrix.

print(md\_array[, 3, 2])

Row1 Row2 Row3 Row4

21 22 23 24

# Access the Complete First Matrix.

print(md\_array[ , , 1])

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

# Access the Complete Second Matrix.

print(md\_array[ , , 2])

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

**Concatenating matrix value with literals.**

md\_array

, , Matrixl1

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Matrix2

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

#Print all elements of 1st row.

print(paste("Element at md\_array[1, 1, 1] = ", md\_array[1, 1, 1]))

print(paste("Element at md\_array[1, 2, 1] = ", md\_array[1, 2, 1]))

print(paste("Element at md\_array[1, 3, 1] = ", md\_array[1, 3, 1]))

[1] "Element at md\_array[1, 1, 1] = 1"

[1] "Element at md\_array[1, 2, 1] = 5"

[1] "Element at md\_array[1, 3, 1] = 9"

print(paste("Element at md\_array[4, 2, 1] = ", md\_array[4, 2, 1]))

[1] "Element at md\_array[4, 2, 1] = 8"

print(paste("Element at md\_array[3, 1, 2] = ", md\_array[3, 1, 2]))

[1] "Element at md\_array[3, 1, 2] = 15"

print(paste("Element at md\_array[4, 3, 2] = ", md\_array[4, 3, 2]))

[1] "Element at md\_array[4, 3, 2] = 24"

print(paste("Element at md\_array[, 3, 2] = ", md\_array[, 3, 2]))

[1] "Element at md\_array[, 3, 2] = 21" "Element at md\_array[, 3, 2] = 22" "Element at

md\_array[, 3, 2] = 23" "Element at md\_array[, 3, 2] = 24"

print(paste("Element at md\_array[3, , 2] = ", md\_array[3, , 2]))

[1] "Element at md\_array[3, , 2] = 15" "Element at md\_array[3, , 2] = 19" "Element at md\_array[3, , 2] = 23"

### Accessing Subset of a R Array Elements

Access subset of multiple items from the Array. To achieve the same we use the R Vector.

**NOTE:**  Negative index position are used to omit those values from Array.

# Accessing R Array Elements

print(md\_array)

, , Matrixl1

Col1 Col2 Col3

Row1 1 5 9

Row2 2 6 10

Row3 3 7 11

Row4 4 8 12

, , Matrix2

Col1 Col2 Col3

Row1 13 17 21

Row2 14 18 22

Row3 15 19 23

Row4 16 20 24

# Access the elements of 1st, 2nd row and 3rd, 1st column in Matrix 1.

print(md\_array[c(1, 2), c(3, 1), 1])

Col3 Col1

Row1 9 1

Row2 10 2

# Access All the element of 2nd and 3rd row in Matrix 2.

print(md\_array[c(2, 3), , 2])

Col1 Col2 Col3

Row2 14 18 22

Row3 15 19 23

# Access All the element of 1st and 3rd Column in Matrix 1.

print(md\_array[ , c(1, 3), 1])

Col1 Col3

Row1 1 9

Row2 2 10

Row3 3 11

Row4 4 12

# Access All the element except 2nd row and 3rd Columm in Matrix 2.

print(md\_array[-2, -3, 2])

Col1 Col2

Row1 13 17

Row3 15 19

Row4 16 20

# 1st column from 1matrix, 2nd column from 2nd matrix and 3column from 3rd matrix

print(c(md\_array[,1,1],md\_array[,2,2],md\_array[,3,3]))

# another workaround

finalMatrix <- c(md\_array[,1,1],md\_array[,2,2],md\_array[,3,3])

dim(finalMatrix) <- c(4,3)

finalMatrix

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

[1,] 1 17 33

[2,] 2 18 34

[3,] 3 19 35

[4,] 4 20 36

>

### R Array Addition and Subtraction

How to use Arithmetic Operators on Matrices to perform arithmetic Operations on Array in R programming.

print(a\_matrix)

east west north south

one 1 2 3 4

two 5 6 7 8

three 9 10 11 12

four 13 14 15 16

print(b\_matrix)

col1 col2 col3 Col4

five 2 10 18 26

six 4 12 20 28

seven 6 14 22 30

eight 8 16 24 32

print(a\_matrix + b\_matrix)

east west north south

one 3 12 21 30

two 9 18 27 36

three 15 24 33 42

four 21 30 39 48

print(a\_matrix - b\_matrix)

east west north south

one -1 -8 -15 -22

two 1 -6 -13 -20

three 3 -4 -11 -18

four 5 -2 -9 -16

print(b\_matrix - a\_matrix)

col1 col2 col3 Col4

five 1 8 15 22

six -1 6 13 20

seven -3 4 11 18

eight -5 2 9 16

print(a\_matrix \* b\_matrix)

east west north south

one 2 20 54 104

two 20 72 140 224

three 54 140 242 360

four 104 224 360 512

print(a\_matrix / b\_matrix)

east west north south

one 0.500 0.2000000 0.1666667 0.1538462

two 1.250 0.5000000 0.3500000 0.2857143

three 1.500 0.7142857 0.5000000 0.4000000

four 1.625 0.8750000 0.6250000 0.5000000

## CHANGE THE DIMENSIONS OF A VECTOR IN R

### Note

If you just want to convert a vector to a matrix.

x <- 1:12 # numeric vactor

x

[1] 1 2 3 4 5 6 7 8 9 10 11 12

class(x)

[1] "integer"

dim(x) <- c(4, 3)

x

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

[1,] 1 5 9

[2,] 2 6 10

[3,] 3 7 11

[4,] 4 8 12

class(x)

[1] "matrix"

dimnames(x) <- list( c("row1","row2","row3","row4"), c("Col1","Col2","Col3"))

x

Col1 Col2 Col3

row1 1 5 9

row2 2 6 10

row3 3 7 11

row4 4 8 12

class(x)

[1] "matrix"

Converted to array

dim(x) <- c(2,2,3)

x

class(x)

[1] "array"

## 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 arrays and print their values..

array1 <- array(1:18,dim = c(3,3,2))

array2 <- array(21:38,dim = c(3,3,2))

print(array1)

print(array2)

# create matrices from these arrays and print their values.

matrix1 <- array1[,,2]

matrix2 <- array2[,,2]

print(matrix1)

print(matrix2)

# Add the matrices.

result <- matrix1+matrix2

print(result)

When we execute the above code, it produces the following result −

# Create two arrays and print their values.

, , 1

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

[1,] 1 4 7

[2,] 2 5 8

[3,] 3 6 9

, , 2

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

[1,] 10 13 16

[2,] 11 14 17

[3,] 12 15 18

, , 1

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

[1,] 21 24 27

[2,] 22 25 28

[3,] 23 26 29

, , 2

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

[1,] 30 33 36

[2,] 31 34 37

[3,] 32 35 38

# create matrices from these arrays and print their values.

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

[1,] 10 13 16

[2,] 11 14 17

[3,] 12 15 18

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

[1,] 30 33 36

[2,] 31 34 37

[3,] 32 35 38

# Add the matrices.

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

[1,] 40 46 52

[2,] 42 48 54

[3,] 44 50 56

## Calculations Across Array Elements

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

### Syntax

apply(x, margin, fun)

Following is the description of the parameters used −

* **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 one array and print their values.

new.array <- array(1:18,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, 1, sum)

print(result)

# Use apply to calculate the sum of the columns across all the matrices.

result <- apply(new.array, 2, sum)

print(result)

# Use apply to calculate the sum of the matrices.

result <- apply(new.array, 3, sum)

print(result)

When we execute the above code, it produces the following result −

# Create one array and print their values.

, , 1

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

[1,] 1 4 7

[2,] 2 5 8

[3,] 3 6 9

, , 2

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

[1,] 10 13 16

[2,] 11 14 17

[3,] 12 15 18

# Use apply to calculate the sum of the rows across all the matrices.

[1] 51 57 63

# Use apply to calculate the sum of the columns across all the matrices.

[1] 39 57 75

# Use apply to calculate the sum of the matrices.

[1] 45 126

If we do calculation on the above matrix in excel sheet, it would look like this.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 4 | 7 | **12** |
| 2 | 5 | 8 | **15** |
| 3 | 6 | 9 | **18** |
| **6** | **15** | **24** | **45** |

|  |  |  |  |
| --- | --- | --- | --- |
| 10 | 13 | 16 | **39** |
| 11 | 14 | 17 | **42** |
| 12 | 15 | 18 | **45** |
| **33** | **42** | **51** | **126** |

# Factors

Factors are the r-objects which are created using a vector. It stores the vector along with the distinct values of the elements 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 modeling.

Factors are created using the **factor()** function.The **nlevels** functions gives the count of levels.

R factors variable is a vector of categorical data. factor() function creates a factor variable, and calculates the categorical distribution of a vector data.

**factor(x = character(), levels, labels = levels, exclude = NA, ordered = is.ordered(x))**

The first three arguments of factor() warrant some exploration:

x: The input vector that you want to turn into a factor.

levels: An optional vector of the values that x might have taken. The default is lexicographically sorted, unique values of x.

labels: Another optional vector that, by default, takes the same values as levels. You can use this argument to rename your 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(class(factor\_apple))

print(nlevels(factor\_apple))

When we execute the above code, it produces the following result −

[1] green green yellow red red red green

Levels: green red yellow

[1] "factor"

# applying the nlevels function we can know the number of distinct values

[1] 3

The fact that you can supply both levels and labels to factor can lead to confusion. Just remember that levels refers to the input values of x, while labels refers to the output values of the new factor.

# Notice that this vector contains the value "South" twice and lacks the value "West". First, convert directions to a factor:

directions <- c("North", "East", "South", "South")

direction.factor <- factor(directions)

print(direction.factor)

print(paste("nlevel is ", nlevels(direction.factor)))

[1] North East South South

Levels: East North South

[1] "nlevel is 3"

# The values are still the same but this time the levels also contain "West".

directions <- c("North", "East", "South", "South")

factor(directions, levels= c("North", "East", "South", "West"))

[1] North East South South

Levels: North East South West

directions <- c("North", "East", "South", "South")

factor(directions, levels= c("North", "East", "South", "West"), labels= c("North", "East", "South", "West"))

[1] North East South South

Levels: North East South West

Now imagine that you actually prefer to have abbreviated names for the levels.

directions <- c("North", "East", "South", "South")

factor(directions, levels= c("North", "East", "South", "West"), labels= c("E", "W", "N", "S"))

[1] E W N N

Levels: E W N S

mons <- c("March","April","January","November","January",

"September","October","September","November","August",

"January","November","November","February","May","August",

"July","December","August","August","September","November",

"February","April")

str <- factor(mons)

print(str)

[1] March April January November January September October September November August

[11] January November November February May August July December August August

[21] September November February April

Levels: April August December February January July March May November October September

fert = c(10,20,20,50,10,20,10,50,20)

fert = factor(fert,levels=c(10,20,50),ordered=TRUE)

fert

[1] 10 20 20 50 10 20 10 50 20

**Levels: 10 < 20 < 50**

fert = c(10,20,20,50,10,20,10,50,20)

fert = factor(fert,levels=c(10,20,50),ordered=FALSE)

fert

[1] 10 20 20 50 10 20 10 50 20

**Levels: 10 20 50**

x <- factor(c("single", "married", "married", "single"), levels = c("single", "married", "divorced"));

x

# Access 2nd element.

x[2]

# Modify 2nd element.

x[2] <- "divorced"

x

Output as like:

[1] single married married single

Levels: single married divorced

[1] married

Levels: single married divorced

[1] single divorced married single

Levels: single married divorced

## Factors in Data Frame

On creating any data frame with a column of text data, R treats the text column as categorical data and creates factors on it.

# Create the vectors for data frame.

height <- c(132,151,162,139,166,147,122)

weight <- c(48,49,66,53,67,52,40)

gender <- c("male","male","female","female","male","female","male")

# Create the data frame.

input\_data <- data.frame(height,weight,gender)

print(input\_data)

# Test if the gender column is a factor.

print(is.factor(input\_data$gender))

# Print the gender column so see the levels.

print(input\_data$gender)

When we execute the above code, it produces the following result −

height weight gender

1 132 48 male

2 151 49 male

3 162 66 female

4 139 53 female

5 166 67 male

6 147 52 female

7 122 40 male

[1] TRUE

[1] male male female female male female male

Levels: female male

## Changing the Order of Levels

The order of the levels in a factor can be changed by applying the factor function again with new order of the levels.

data <- c("East","West","East","North","North","East","West","West","West","East","North")

# Create the factors

factor\_data <- factor(data)

print(factor\_data)

# Apply the factor function with required order of the level.

new\_order\_data <- factor(data, levels = c("East","West","North"))

print(new\_order\_data)

When we execute the above code, it produces the following result −

[1] East West East North North East West West West East North

Levels: East North West

[1] East West East North North East West West West East North

Levels: East West North

## Generating Factor Levels

We can generate factor levels by using the **gl()** function. It takes two integers as input which indicates how many levels and how many times each level.

### Syntax

gl(n, k, labels)

Following is the description of the parameters used −

* **n** is a integer giving the number of levels.
* **k** is a integer giving the number of replications.
* **labels** is a vector of labels for the resulting factor levels.

### Example

v <- gl(3, 4, labels = c("Tampa", "Seattle","Boston"))

print(v)

x <- gl(3, 4, labels = c("Tampa", "Seattle","Boston","Las Vegas"))

print(x)

When we execute the above code, it produces the following result −

Tampa Tampa Tampa Tampa Seattle Seattle Seattle Seattle Boston Boston Boston Boston

Levels: Tampa Seattle Boston

[1] Tampa Tampa Tampa Tampa Seattle Seattle Seattle Seattle Boston Boston Boston Boston

Levels: Tampa Seattle Boston Las Vegas

## Data Frames

**Data frames are tabular data objects.** Unlike a matrix in data frame 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.**

A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.

Following are the characteristics of a data frame.

* The column names should be non-empty.
* The row names should be unique.
* The data stored in a data frame can be of numeric, factor or character type.
* Each column should contain same number of data items.

Data Frames are created using the **data.frame()** function.

# Create the data frame.

df1 <- data.frame(

name = c("Dev", "Mark","Gita","Sonia"),

gender = c("Male", "Male","Female","Female"),

height = c(152, 171.5, 165, 170),

weight = c(81,93, 78, 80),

Age = c(42,38,26,35)

)

print(df1)

When we execute the above code, it produces the following result −

name gender height weight Age

1 Dev Male 152.0 81 42

2 Mark Male 171.5 93 38

3 Gita Female 165.0 78 26

4 Sonia Female 170.0 80 35

#### Built-in Data Frame

We can use built-in data frame in R, called **mtcars**.

> mtcars

mpg cyl disp hp drat wt qsec vs am gear carb

Mazda RX4 21.0 6 160.0 110 3.90 2.620 16.46 0 1 4 4

Mazda RX4 Wag 21.0 6 160.0 110 3.90 2.875 17.02 0 1 4 4

Datsun 710 22.8 4 108.0 93 3.85 2.320 18.61 1 1 4 1

Hornet 4 Drive 21.4 6 258.0 110 3.08 3.215 19.44 1 0 3 1

Hornet Sportabout 18.7 8 360.0 175 3.15 3.440 17.02 0 0 3 2

Valiant 18.1 6 225.0 105 2.76 3.460 20.22 1 0 3 1

Duster 360 14.3 8 360.0 245 3.21 3.570 15.84 0 0 3 4

Merc 240D 24.4 4 146.7 62 3.69 3.190 20.00 1 0 4 2

The top line of the table, called the header, contains the column names. Each horizontal line afterward denotes a data row, which begins with the name of the row, and then followed by the actual data. Each data member of a row is called a cell.

To retrieve data in a cell, we would enter its row and column coordinates in the single square bracket "[]" operator. The two coordinates are separated by a comma.

Here is the cell value from the fifth row, second column of mtcars.

mtcars

mpg cyl disp hp drat wt qsec vs am gear carb

Mazda RX4 21.0 6 160.0 110 3.90 2.620 16.46 0 1 4 4

Mazda RX4 Wag 21.0 6 160.0 110 3.90 2.875 17.02 0 1 4 4

Datsun 710 22.8 4 108.0 93 3.85 2.320 18.61 1 1 4 1

Hornet 4 Drive 21.4 6 258.0 110 3.08 3.215 19.44 1 0 3 1

Hornet Sportabout 18.7 **8** 360.0 175 3.15 3.440 17.02 0 0 3 2

Valiant 18.1 6 225.0 105 2.76 3.460 20.22 1 0 3 1

Duster 360 14.3 8 360.0 245 3.21 3.570 15.84 0 0 3 4

Merc 240D 24.4 4 146.7 62 3.69 3.190 20.00 1 0 4 2

Merc 230 22.8 4 140.8 95 3.92 3.150 22.90 1 0 4 2

Merc 280 19.2 6 167.6 123 3.92 3.440 18.30 1 0 4 4

mtcars[5, 2]

[1] **8**

Moreover, we can use the row and column names instead of the numeric coordinates.

mtcars["Ferrari Dino", "cyl"]

[1] 6

mtcars["Mazda RX4","disp"]

[1] 160

The number of data rows in the data frame is given by the nrow function.

nrow(mtcars)

[1] 32

The number of columns of a data frame is given by the ncol function.

ncol(mtcars)

[1] 11

Further details of the mtcars data set is available in the R documentation.

help(mtcars)

#### Preview

Instead of printing out the entire data frame, it is often desirable to preview it with the headfunction beforehand.

head(mtcars)

mpg cyl disp hp drat wt qsec vs am gear carb

Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4

Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4

Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1

Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1

Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2

Valiant 18.1 6 225 105 2.76 3.460 20.22 1 0 3 1

# Data Frame Column Vector

NOTE: Only column value is retrived without rows names.

We reference a data frame column with the *double square bracket*"[[]]" operator will retrieve character data as factor. Single [] will retrieve column values as it is.

mtcars[[3]]

[1] 160.0 160.0 108.0 258.0 360.0 225.0 360.0 146.7 140.8 167.6 167.6 275.8 275.8 275.8 472.0 460.0 440.0 78.7 75.7

[20] 71.1 120.1 318.0 304.0 350.0 400.0 79.0 120.3 95.1 351.0 145.0 301.0 121.0

We can retrieve the same column vector by its name.

mtcars[["disp"]]

[1] 160.0 160.0 108.0 258.0 360.0 225.0 360.0 146.7 140.8 167.6 167.6 275.8 275.8 275.8 472.0 460.0 440.0 78.7 75.7

[20] 71.1 120.1 318.0 304.0 350.0 400.0 79.0 120.3 95.1 351.0 145.0 301.0 121.0

We can also retrieve with the "$" operator in lieu of the double square bracket operator.

mtcars$disp

[1] 160.0 160.0 108.0 258.0 360.0 225.0 360.0 146.7 140.8 167.6 167.6 275.8 275.8 275.8 472.0 460.0 440.0 78.7 75.7

[20] 71.1 120.1 318.0 304.0 350.0 400.0 79.0 120.3 95.1 351.0 145.0 301.0 121.0

Yet another way to retrieve the same column vector is to use the single square bracket "[]" operator. We prepend the column name with a comma character, which signals a wildcard match for the row position.

mtcars[,"disp"]

[1] 160.0 160.0 108.0 258.0 360.0 225.0 360.0 146.7 140.8 167.6 167.6 275.8 275.8 275.8 472.0 460.0 440.0 78.7 75.7

[20] 71.1 120.1 318.0 304.0 350.0 400.0 79.0 120.3 95.1 351.0 145.0 301.0 121.0

mtcars["","disp"]

[1] NA

mtcars[3]

disp

Mazda RX4 160.0

Mazda RX4 Wag 160.0

Datsun 710 108.0

Hornet 4 Drive 258.0

Hornet Sportabout 360.0

Valiant 225.0

Duster 360 360.0

Merc 240D 146.7

Merc 230 140.8

Merc 280 167.6

Merc 280C 167.6

Merc 450SE 275.8

Merc 450SL 275.8

# Data Frame Column Slice

NOTE: Along with column values, rows name also dispalyed.

We retrieve a data frame column slice with the *single square bracket*"[]" operator.

#### Numeric Indexing

The following is a slice containing the first column of the built-in data set [mtcars](http://www.r-tutor.com/node/10).

mtcars[1]

mpg

Mazda RX4 21.0

Mazda RX4 Wag 21.0

Datsun 710 22.8

Hornet 4 Drive 21.4

Hornet Sportabout 18.7

Valiant 18.1

Duster 360 14.3

#### Name Indexing

We can retrieve the same column slice by its name.

mtcars["mpg"]

mpg

Mazda RX4 21.0

Mazda RX4 Wag 21.0

Datsun 710 22.8

Hornet 4 Drive 21.4

Hornet Sportabout 18.7

Valiant 18.1

Duster 360 14.3

Merc 240D 24.4

To retrieve a data frame slice with the two columns mpg and hp, we pack the column names in an index vector inside the single square bracket operator.

mtcars[c("mpg", "hp")]

mpg hp

Mazda RX4 21.0 110

Mazda RX4 Wag 21.0 110

Datsun 710 22.8 93

Hornet 4 Drive 21.4 110

Hornet Sportabout 18.7 175

Valiant 18.1 105

Duster 360 14.3 245

Merc 240D 24.4 62

Merc 230 22.8 95

Merc 280 19.2 123

Merc 280C 17.8 123

# Data Frame Row Slice

We retrieve rows from a data frame with the single square bracket operator, just like what we did with columns. However, in addition to an index vector of row positions, we append an extra comma character. This is important, as the extra comma signals a wildcard match for the second coordinate for column positions.

#### Numeric Indexing

For example, the following retrieves a row record of the built-in data set [mtcars](http://www.r-tutor.com/node/10). Please notice the extra comma in the square bracket operator, and it is not a typo.

mtcars[24,]

mpg cyl disp hp drat wt qsec vs am gear carb

Camaro Z28 13.3 8 350 245 3.73 3.84 15.41 0 0 3 4

To retrieve more than one rows, we use a numeric index vector.

mtcars[c(3, 24),]

mpg cyl disp hp drat wt qsec vs am gear carb

Datsun 710 22.8 4 108 93 3.85 2.32 18.61 1 1 4 1

Camaro Z28 13.3 8 350 245 3.73 3.84 15.41 0 0 3 4

#### Name Indexing

We can retrieve a row by its name.

mtcars["Camaro Z28",]

mpg cyl disp hp drat wt qsec vs am gear carb

Camaro Z28 13.3 8 350 245 3.73 3.84 15.41 0 0 3 4

To retrieve more than one rows, we use rows name in index vector.

mtcars[c("Datsun 710", "Camaro Z28","Merc 280","Toyota Corolla"),]

mpg cyl disp hp drat wt qsec vs am gear carb

Datsun 710 22.8 4 108.0 93 3.85 2.320 18.61 1 1 4 1

Camaro Z28 13.3 8 350.0 245 3.73 3.840 15.41 0 0 3 4

Merc 280 19.2 6 167.6 123 3.92 3.440 18.30 1 0 4 4

Toyota Corolla 33.9 4 71.1 65 4.22 1.835 19.90 1 1 4 1

To retrieve multiple columns

mtcars[,c("mpg","cyl")]

mpg cyl

Mazda RX4 21.0 6

Mazda RX4 Wag 21.0 6

Datsun 710 22.8 4

Hornet 4 Drive 21.4 6

#### Logical Indexing

Lastly, we can retrieve rows with a logical index vector. In the following vector L, the member value is TRUE if the car has automatic transmission, and FALSE if otherwise.

L = mtcars$am == 0

L

[1] FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE

[20] FALSE TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

Here is the list of vehicles with automatic transmission.

mtcars[L,]

mpg cyl disp hp drat wt qsec vs am gear carb

Hornet 4 Drive 21.4 6 258.0 110 3.08 3.215 19.44 1 0 3 1

Hornet Sportabout 18.7 8 360.0 175 3.15 3.440 17.02 0 0 3 2

Valiant 18.1 6 225.0 105 2.76 3.460 20.22 1 0 3 1

Duster 360 14.3 8 360.0 245 3.21 3.570 15.84 0 0 3 4

Merc 240D 24.4 4 146.7 62 3.69 3.190 20.00 1 0 4 2

Merc 230 22.8 4 140.8 95 3.92 3.150 22.90 1 0 4 2

Merc 280 19.2 6 167.6 123 3.92 3.440 18.30 1 0 4 4

Merc 280C 17.8 6 167.6 123 3.92 3.440 18.90 1 0 4 4

Merc 450SE 16.4 8 275.8 180 3.07 4.070 17.40 0 0 3 3

Merc 450SL 17.3 8 275.8 180 3.07 3.730 17.60 0 0 3 3

Merc 450SLC 15.2 8 275.8 180 3.07 3.780 18.00 0 0 3 3

mtcars[!L,]

mpg cyl disp hp drat wt qsec vs am gear carb

Mazda RX4 21.0 6 160.0 110 3.90 2.620 16.46 0 1 4 4

Mazda RX4 Wag 21.0 6 160.0 110 3.90 2.875 17.02 0 1 4 4

Datsun 710 22.8 4 108.0 93 3.85 2.320 18.61 1 1 4 1

Fiat 128 32.4 4 78.7 66 4.08 2.200 19.47 1 1 4 1

Honda Civic 30.4 4 75.7 52 4.93 1.615 18.52 1 1 4 2

Toyota Corolla 33.9 4 71.1 65 4.22 1.835 19.90 1 1 4 1

Fiat X1-9 27.3 4 79.0 66 4.08 1.935 18.90 1 1 4 1

Porsche 914-2 26.0 4 120.3 91 4.43 2.140 16.70 0 1 5 2

Lotus Europa 30.4 4 95.1 113 3.77 1.513 16.90 1 1 5 2

Ford Pantera L 15.8 8 351.0 264 4.22 3.170 14.50 0 1 5 4

Ferrari Dino 19.7 6 145.0 175 3.62 2.770 15.50 0 1 5 6

And here is the gas mileage data for automatic transmission.

mtcars[L,]$mpg

[1] 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4 10.4 14.7 21.5 15.5 15.2 13.3 19.2

Another Examples

## Create Data Frame

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11","2015-03-27")),

stringsAsFactors = FALSE

)

# Print the data frame.

print(emp.data)

When we execute the above code, it produces the following result −

emp\_id emp\_name salary start\_date

1 1 Rick 623.30 2012-01-01

2 2 Dan 515.20 2013-09-23

3 3 Michelle 611.00 2014-11-15

4 4 Ryan 729.00 2014-05-11

5 5 Gary 843.25 2015-03-27

## Get the Structure of the Data Frame

The structure of the data frame can be seen by using **str()** function.

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11", "2015-03-27")),

stringsAsFactors = FALSE

)

# Get the structure of the data frame.

str(emp.data)

When we execute the above code, it produces the following result −

'data.frame': 5 obs. of 4 variables:

$ emp\_id : int 1 2 3 4 5

$ emp\_name : chr "Rick" "Dan" "Michelle" "Ryan" ...

$ salary : num 623 515 611 729 843

$ start\_date: Date, format: "2012-01-01" "2013-09-23" "2014-11-15" "2014-05-11" ...

## Summary of Data in Data Frame

The statistical summary and nature of the data can be obtained by applying **summary()** function.

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11", "2015-03-27")),

stringsAsFactors = FALSE

)

# Print the summary.

print(summary(emp.data))

When we execute the above code, it produces the following result −

emp\_id emp\_name salary start\_date

Min. :1 Length:5 Min. :515.2 Min. :2012-01-01

1st Qu.:2 Class :character 1st Qu.:611.0 1st Qu.:2013-09-23

Median :3 Mode :character Median :623.3 Median :2014-05-11

Mean :3 Mean :664.4 Mean :2014-01-14

3rd Qu.:4 3rd Qu.:729.0 3rd Qu.:2014-11-15

Max. :5 Max. :843.2 Max. :2015-03-27

## Extract Data from Data Frame

Extract specific column from a data frame using column name.

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-11", "2015-03-27")),

stringsAsFactors = FALSE

)

# Extract Specific columns.

result <- data.frame(emp.data$emp\_name,emp.data$salary)

print(result)

When we execute the above code, it produces the following result −

emp.data.emp\_name emp.data.salary

1 Rick 623.30

2 Dan 515.20

3 Michelle 611.00

4 Ryan 729.00

5 Gary 843.25

Extract the first two rows and then all columns

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11", "2015-03-27")),

stringsAsFactors = FALSE

)

# Extract first two rows.

result <- emp.data[1:2,]

print(result)

When we execute the above code, it produces the following result −

emp\_id emp\_name salary start\_date

1 1 Rick 623.3 2012-01-01

2 2 Dan 515.2 2013-09-23

Extract 3rd and 5th row with 2nd and 4th column

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11", "2015-03-27")),

stringsAsFactors = FALSE

)

# Extract 3rd and 5th row with 2nd and 4th column.

result <- emp.data[c(3,5),c(2,4)]

print(result)

When we execute the above code, it produces the following result −

emp\_name start\_date

3 Michelle 2014-11-15

5 Gary 2015-03-27

## Expand Data Frame

A data frame can be expanded by adding columns and rows.

### Add Column

Just add the column vector using a new column name.

# Create the data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11","2015-03-27")),

stringsAsFactors = FALSE

)

# Add the "dept" coulmn.

emp.data$dept <- c("IT","Operations","IT","HR","Finance")

v <- emp.data

print(v)

When we execute the above code, it produces the following result −

emp\_id emp\_name salary start\_date dept

1 1 Rick 623.30 2012-01-01 IT

2 2 Dan 515.20 2013-09-23 Operations

3 3 Michelle 611.00 2014-11-15 IT

4 4 Ryan 729.00 2014-05-11 HR

5 5 Gary 843.25 2015-03-27 Finance

### Add Row

To add more rows permanently to an existing data frame, we need to bring in the new rows in the same structure as the existing data frame and use the **rbind()** function.

In the example below we create a data frame with new rows and merge it with the existing data frame to create the final data frame.

# Create the first data frame.

emp.data <- data.frame(

emp\_id = c (1:5),

emp\_name = c("Rick","Dan","Michelle","Ryan","Gary"),

salary = c(623.3,515.2,611.0,729.0,843.25),

start\_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11", "2015-03-27")),

dept = c("IT","Operations","IT","HR","Finance"),

stringsAsFactors = FALSE

)

# Create the second data frame

emp.newdata <- data.frame(

emp\_id = c (6:8),

emp\_name = c("Rasmi","Pranab","Tusar"),

salary = c(578.0,722.5,632.8),

start\_date = as.Date(c("2013-05-21","2013-07-30","2014-06-17")),

dept = c("IT","Operations","Fianance"),

stringsAsFactors = FALSE

)

# Bind the two data frames.

emp.finaldata <- rbind(emp.data,emp.newdata)

print(emp.finaldata)

When we execute the above code, it produces the following result −

emp\_id emp\_name salary start\_date dept

1 1 Rick 623.30 2012-01-01 IT

2 2 Dan 515.20 2013-09-23 Operations

3 3 Michelle 611.00 2014-11-15 IT

4 4 Ryan 729.00 2014-05-11 HR

5 5 Gary 843.25 2015-03-27 Finance

6 6 Rasmi 578.00 2013-05-21 IT

7 7 Pranab 722.50 2013-07-30 Operations

8 8 Tusar 632.80 2014-06-17 Fianance

# R - Variables

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

|  |  |  |
| --- | --- | --- |
| **Variable Name** | **Validity** | **Reason** |
| var\_name2. | valid | Has letters, numbers, dot and underscore |
| var\_name% | Invalid | Has the character '%'. Only dot(.) and underscore allowed. |
| 2var\_name | invalid | Starts with a number |
| .var\_name , var.name | valid | Can start with a dot(.) but the dot(.)should not be followed by a number. |
| .2var\_name | invalid | The starting dot is followed by a number making it invalid. |
| \_var\_name | invalid | Starts with \_ which is not valid |

## 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("learn","R")

# 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")

When we execute the above code, it produces the following result −

[1] 0 1 2 3

var.1 is 0 1 2 3

var.2 is learn R

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 as shown below.

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")

.var\_y <- TRUE

cat("The class of .var\_y becomes ",class(.var\_y),"\n")

.var\_z <- FALSE

cat("The class of .var\_z becomes ",class(.var\_z),"\n")

When we execute the above code, it produces the following result −

The class of var\_x is character

Now the class of var\_x is numeric

Next the class of var\_x becomes integer

The class of .var\_y becomes logical

The class of .var\_z becomes logical

## Finding Variables

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

print(ls())

ls()

When we execute the above code, it produces the following result −

[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"))

When we execute the above code, it produces the following result −

[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))

When we execute the above code, it produces the following result −

[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)

When we execute the above code, it produces the following result −

[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())

When we execute the above code, it produces the following result −

character(0)

rm() and ls() function together doesn’t remove hidden variable.

ls(all.name = TRUE)

[1] ".Random.seed" ".var\_x" ".var\_y" ".var\_z"

rm(.var\_x)

rm(.var\_y)

rm(.var\_z)

ls()

character(0)

ls(all.name = TRUE)

[1] ".Random.seed"

# R - 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 and provides following types of operators.

## Types of Operators

We have the following types of operators in R programming −

* Arithmetic Operators
* Relational Operators
* Logical Operators
* Assignment Operators
* 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)  it produces the following result −  [1] 10.0 8.5 10.0 |
| − | Subtracts second vector from the first | v <- c( 2,5.5,6)  t <- c(8, 3, 4)  print(v-t)  it produces the following result −  [1] -6.0 2.5 2.0 |
| \* | Multiplies both vectors | v <- c( 2,5.5,6)  t <- c(8, 3, 4)  print(v\*t)  it produces the following result −  [1] 16.0 16.5 24.0 |
| / | Divide the first vector with the second | v <- c( 2,5.5,6)  t <- c(8, 3, 4)  print(v/t)  When we execute the above code, it produces the following result −  [1] 0.250000 1.833333 1.500000 |
| %% | Give the remainder of the first vector with the second | v <- c( 2,5.5,6)  t <- c(8, 3, 4)  print(v%%t)  it produces the following result −  [1] 2.0 2.5 2.0 |
| %/% | The result of division of first vector with second (quotient) | v <- c( 2,5.5,6)  t <- c(8, 3, 4)  print(v%/%t)  it produces the following result −  [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)  it produces the following result −  [1] 256.000 166.375 1296.000 |

## Relational Operators

Following table shows the relational operators supported by R language. Each element of the first vector is compared with the corresponding element of the second 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)  it produces the following result −  [1] FALSE TRUE FALSE FALSE |
| < | Checks if each element of the first vector is less than the corresponding element of the second vector. | v <- c(2,5.5,6,9)  t <- c(8,2.5,14,9)  print(v < t)  it produces the following result −  [1] TRUE FALSE TRUE FALSE |
| == | Checks if each element of the first vector is equal to the corresponding element of the second vector. | v <- c(2,5.5,6,9)  t <- c(8,2.5,14,9)  print(v == t)  it produces the following result −  [1] FALSE FALSE FALSE TRUE |
| <= | Checks if each element of the first vector is less than or equal to the corresponding element of the second vector. | v <- c(2,5.5,6,9)  t <- c(8,2.5,14,9)  print(v<=t)  it produces the following result −  [1] TRUE FALSE TRUE TRUE |
| >= | Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector. | v <- c(2,5.5,6,9)  t <- c(8,2.5,14,9)  print(v>=t)  it produces the following result −  [1] FALSE TRUE FALSE TRUE |
| != | Checks if each element of the first vector is unequal to the corresponding element of the second vector. | v <- c(2,5.5,6,9)  t <- c(8,2.5,14,9)  print(v!=t)  it produces the following result −  [1] TRUE TRUE TRUE FALSE |

## 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 the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value (TRUE or FALSE), if both elements are TRUE result is TRUE .

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE. | v <- c(3,1,TRUE,2+3i)  t <- c(4,1,FALSE,2+3i)  print(v&t)  v <- c(3,1,TRUE,2+3i)  t <- c(-4,1,FALSE,2+3i)  print(v&t)  v <- c(3,1,TRUE,2+3i)  t <- c(0,1,FALSE,2+3i)  print(v&t)  v <- c(0,1,TRUE,2+3i)  t <- c(4,1,FALSE,2+3i)  print(v&t)  it produces the following result −  [1] TRUE TRUE FALSE TRUE  [1] TRUE TRUE FALSE TRUE  [1] FALSE TRUE FALSE TRUE  [1] FALSE TRUE FALSE TRUE |
| | | It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if one of the elements is TRUE. | v <- c(3,0,TRUE,2+2i)  t <- c(4,0,FALSE,2+3i)  print(v|t)  v <- c(3,0,TRUE,2+2i)  t <- c(4,5,FALSE,2+3i)  print(v|t)  it produces the following result −  [1] TRUE FALSE TRUE TRUE  [1] TRUE TRUE TRUE TRUE |
| ! | It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value. | v <- c(3,0,TRUE,2+2i)  print(!v)  it produces the following result −  [1] FALSE TRUE FALSE FALSE |

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

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

## Assignment Operators

These operators are used to assign values to vectors.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| <−  or  =  or  <<− | Called Left Assignment | v1 <- c(3,1,TRUE,2+3i)  v2 <<- c(3,1,TRUE,2+3i)  v3 = c(3,1,TRUE,2+3i)  v4 = c(3,1,FALSE,2+3i)  print(v1)  print(v2)  print(v3)  print(v4)  it produces the following result −  [1] 3+0i 1+0i 1+0i 2+3i  [1] 3+0i 1+0i 1+0i 2+3i  [1] 3+0i 1+0i 1+0i 2+3i  [1] 3+0i 1+0i 0+0i 2+3i |
| ->  or  ->> | Called Right Assignment | c(3,1,TRUE,2+3i) -> v1  c(3,1,TRUE,2+3i) ->> v2  print(v1)  print(v2)  it produces the following result −  [1] 3+0i 1+0i 1+0i 2+3i  [1] 3+0i 1+0i 1+0i 2+3i |

## Miscellaneous Operators

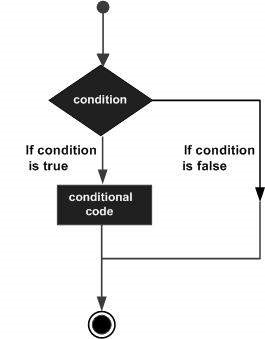
These operators are used for specific purpose and not general mathematical or logical computation.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| : | Colon operator. It creates the series of numbers in sequence for a vector. | v <- 2:8  print(v)  it produces the following result −  [1] 2 3 4 5 6 7 8 |
| %in% | This operator is used to identify if an element belongs to a vector. | v1 <- 8  v2 <- 12  t <- 1:10  print(v1 %in% t)  print(v2 %in% t)  it produces the following result −  [1] TRUE  [1] FALSE |
| %\*% | This operator is used to multiply a matrix with its transpose. | M = matrix( c(2,6,5,1,10,4), nrow = 2,ncol = 3,byrow = TRUE)  t = M %\*% t(M)  print(t)  it produces the following result −  [,1] [,2]  [1,] 65 82  [2,] 82 117 |

# R - Decision making

Decision making structures require the programmer to specify 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 determined to be **true**, and optionally, other statements to be executed if the condition is determined to be **false**.

Following is the general form of a typical decision making structure found in most of the programming languages −



R provides the following types of decision making statements.

|  |  |
| --- | --- |
| **Sr.No.** | **Statement & Description** |
| 1 | [**if statement**](https://www.tutorialspoint.com/r/r_if_statement.htm)  An **if** statement consists of a Boolean expression followed by one or more statements. |
| 2 | [**if...else statement**](https://www.tutorialspoint.com/r/r_if_else_statement.htm)  An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false. |
| 3 | [**switch statement**](https://www.tutorialspoint.com/r/r_switch_statement.htm)  A **switch** statement allows a variable to be tested for equality against a list of values. |

## R if statement

The syntax of if statement is:

if (test\_expression) {

statement

}

If the test\_expression is TRUE, the statement gets executed. But if it's FALSE, nothing happens.

Here, test\_expression can be a logical or numeric vector, but only the first element is taken into consideration.

In the case of numeric vector, zero is taken as FALSE, rest as TRUE.

Note: R script run using source function perfectly.

stud.name <- readline(prompt="Enter Student Name: ")

stud.mark <-scan(n=1)

if(stud.mark >= 50 ) {

cat(stud.name , " has scored ", stud.mark, " marks.", "\n" )

}

stud.name <- readline(prompt="Enter Student Name: ")

stud.mark <-scan(n=1)

if(stud.mark >= 50 )

{

cat(stud.name , " has scored ", stud.mark, " marks"," he is pass.", "\n" )

} else {

cat(stud.name , " has scored ", stud.mark, " marks"," he is fail." , "\n" )

}

stud.name <- readline(prompt="Enter Student Name: ")

stud.mark <-scan(n=1)

if(stud.mark < 50 )

{

cat(stud.name , " scores is ", stud.mark, " marks"," he is fail." , "\n")

} else if ((stud.mark >= 50 ) && (stud.mark < 60 )) {

cat(stud.name , " scores is ", stud.mark, " marks"," he stood 2nd." , "\n" )

} else if ((stud.mark >= 60 ) && (stud.mark < 80 )) {

cat(stud.name , " scores is ", stud.mark, " marks"," he stood 1st." , "\n" )

} else {

cat(stud.name , " has scores ", stud.mark, " marks"," outstanding…...", "\n" )

}

# R switch()

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

The basic syntax for creating a switch statement in R is −

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

If the value evaluated is a number, then that item of the list is returned.

switch(2,"red","green","blue")

[1] "green"

switch(1,"red","green","blue")

[1] "red"

In the above example, "red","green","blue" form a three item list.

The switch() function returns the corresponding item to the numeric value evaluated.

If the numeric value is out of range (greater than the number of items in the list or smaller than 1), then, NULL is returned.

x <- switch(4,"red","green","blue")

x

NULL

x <- switch(0,"red","green","blue")

x

NULL

The result of the statement can be a string as well. In this case, the matching named item's value is returned.

switch("color", "color" = "red", "shape" = "square", "length" = 5)

[1] "red"

switch("length", "color" = "red", "shape" = "square", "length" = 5)

[1] 5

# by index

switch(1, "one", "two")

[1] "one"

# by index with complex expressions

switch(2, {"one"}, {"two"})

[1] "two"

# by index with complex named expression

switch(1, foo={"one"}, bar={"two"})

[1] "one"

# by name with complex named expression

switch("bar", foo={"one"}, bar={"two"})

[1] "two"

switch("bar", foo=c("one","one +"), bar= c("two","two +"))

[1] "two" "two +"

# with default option

switch("lounge", foo={"one"}, bar={"two"}, "three")

[1] "three"

# R - Loops

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and the following is the general form of a loop statement in most of the programming languages –



R programming language provides the following kinds of loop to handle looping requirements.

|  |  |
| --- | --- |
| **Sr.No.** | **Loop Type & Description** |
| 1 | [**repeat loop**](https://www.tutorialspoint.com/r/r_repeat_loop.htm)  Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 2 | [**while loop**](https://www.tutorialspoint.com/r/r_while_loop.htm)  Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| 3 | [**for loop**](https://www.tutorialspoint.com/r/r_for_loop.htm)  Like a while statement, except that it tests the condition at the end of the loop body. |

## Loop Control Statements

Loop control statements 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.

|  |  |
| --- | --- |
| **Sr.No.** | **Control Statement & Description** |
| 1 | [**break statement**](https://www.tutorialspoint.com/r/r_break_statement.htm)  Terminates the **loop** statement and transfers execution to the statement immediately following the loop. |
| 2 | [**Next statement**](https://www.tutorialspoint.com/r/r_next_statement.htm)  The **next** statement simulates the behavior of R switch. |

# R - Repeat Loop

The **Repeat loop** executes the same code again and again until a stop condition is met. Here conditional code is execute first and then break condition is checked. Even if the break condition is true for the first iteration, conditional code is execute at least once.

## Syntax

The basic syntax for creating a repeat loop in R is −

repeat {

commands

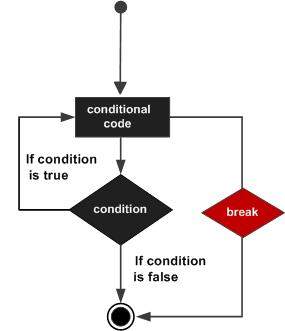
if(condition) {

break

}

}

## Flow Diagram



## Example

v <- c("Hello","World")

cnt <- 1

repeat {

print(v)

cnt <- cnt+1

if(cnt < 5) {

break

}

}

When the above code is compiled and executed, it produces the following result −

[1] "Hello" "World"

[1] "Hello" "World"

[1] "Hello" "World"

[1] "Hello" "World"

Take the sum of numbers between given number and 10.

total <- 0

number <- as.integer(readline(prompt="Please Enter any integer value below 10: "))

repeat {

total = total + number

number = number + 1

if (number > 10) {

break

}

}

print(paste("The total Sum of Numbers From the Repeat Loop is: ", total))

When the above code is compiled and executed, it produces the following result −

OUTPUT: We are going to enter number =5. It means, total = 5 + 6 + 7 + 8 + 9 + 10 = 45

[1] "The total Sum of Numbers From the Repeat Loop is: 45"

## **Infinite Repeat in R Programming**

If you forgot to use the Conditions or Break statement to terminate the Repeat loop then statements inside the loop will be executed infinite times (also called as infinite loop).

total <- 0

number <- as.integer(readline(prompt="Please Enter any integer Value: "))

repeat {

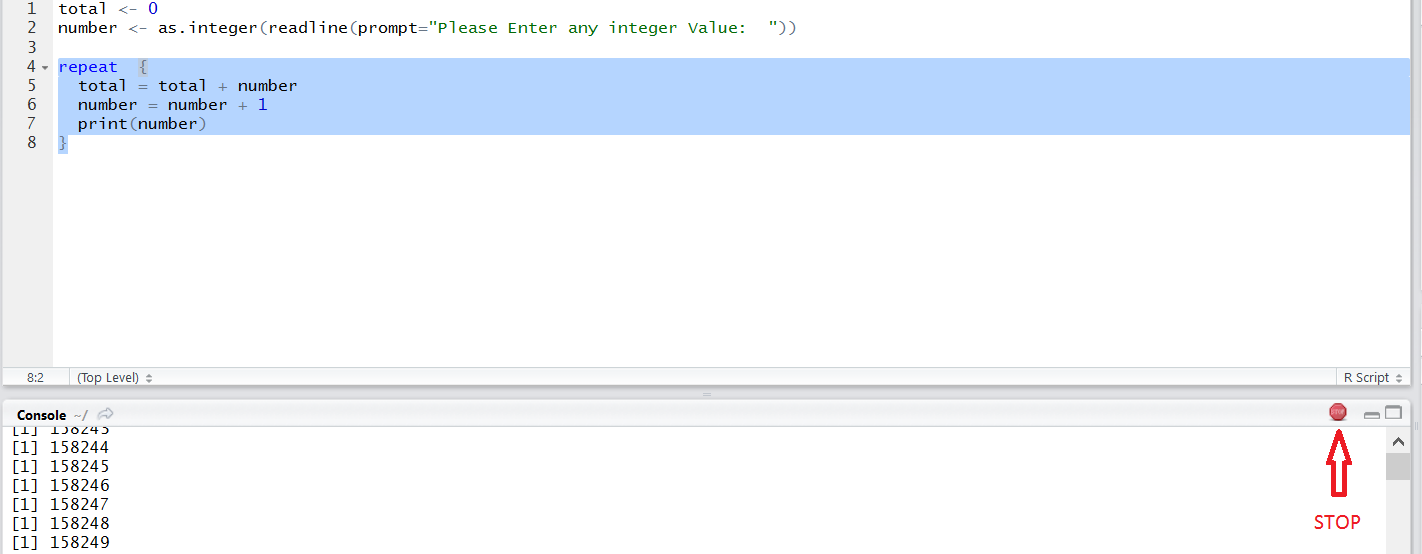
total = total + number

number = number + 1

print(number)

}

To stop it, click on stop button.



# R - While Loop

The While loop executes the same code again and again until a stop condition is met. Here break condition is checked first and then conditional code is executed.

## Syntax

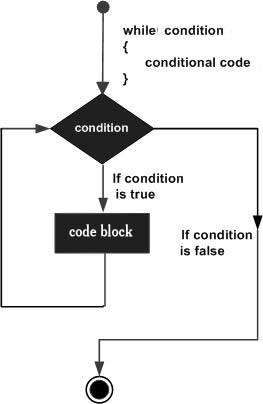
The basic syntax for creating a while loop in R is −

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 iteration")

cnt <- 1

while (cnt <= 5)

{

print(v)

cnt = cnt + 1

}

When the above code is compiled and executed, it produces the following result −

[1] "Hello" "while loop iteration"

[1] "Hello" "while loop iteration"

[1] "Hello" "while loop iteration "

[1] "Hello" "while loop iteration "

[1] "Hello" "while loop iteration "

While loop used to enter ATM pin.

readinteger <- function()

{

n <- readline(prompt="Please, enter your PIN: ")

}

response<-as.integer(readinteger())

cnt <- 1

while (response!=9999)

{

if(cnt == 4)

{

print("Sorry, exceed limit");

break()} else {

print("Sorry, enter correct PIN");

response<-as.integer(readinteger());

cnt <- cnt+1 }

}

When the above code is compiled and executed, it produces the following result −

[1] "Sorry, enter correct PIN"

Please, enter your PIN: 2222

[1] "Sorry, enter correct PIN"

Please, enter your PIN: 3333

[1] "Sorry, enter correct PIN"

Please, enter your PIN: 4444

[1] "Sorry, exceed limit"

# R - 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.

## Syntax

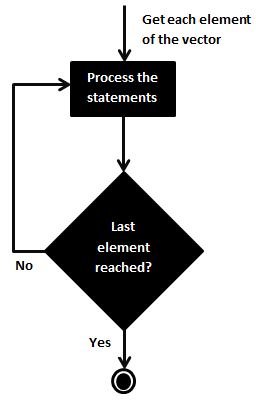
The basic syntax for creating a **for** loop statement in R is −

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)

}

When the above code is compiled and executed, it produces the following result −

[1] "A"

[1] "B"

[1] "C"

[1] "D"

for(i in letters[1:5])

print(i)

[1] "a"

[1] "b"

[1] "c"

[1] "d"

[1] "e"

for (year in 2000:2017){

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

}

[1] "The year is 2000"

[1] "The year is 2001"

[1] "The year is 2002"

[1] "The year is 2003"

[1] "The year is 2004"

[1] "The year is 2005"

[1] "The year is 2006"

[1] "The year is 2007"

[1] "The year is 2008"

[1] "The year is 2009"

[1] "The year is 2010"

[1] "The year is 2011"

[1] "The year is 2012"

[1] "The year is 2013"

[1] "The year is 2014"

[1] "The year is 2015"

[1] "The year is 2016"

[1] "The year is 2017"

for(i in 1:5) print(1:i)

[1] 1

[1] 1 2

[1] 1 2 3

[1] 1 2 3 4

[1] 1 2 3 4 5

for(n in c(2,5,10,20,50)) {

x <- stats::rnorm(n)

cat(n, ": ", sum(x^2), "\n", sep = "")

}

# Create a vector filled with random normal values

u1 <- rnorm(15)

print("This loop calculates the square of the first 10 elements of vector u1")

u1

# Initialize usq

usq <- 0

for(i in 1:10) {

# i-th element of 'u1' squared into 'i'-th position of 'usq'

usq[i] <- u1[i]\*u1[i]

print(usq[i])

}

print(i)

**Output as below:**

[1] "This loop calculates the square of the first 10 elements of vector u1"

[1] 0.149195934 -0.139303172 -0.198272005 -0.789264348 2.677009231 0.193597546

[7] -0.759557359 0.072116770 0.111880115 -1.136383031 0.008112255 -0.429596574

[13] -0.790761812 0.608455073 -0.608093421

[1] 0.02225943

[1] 0.01940537

[1] 0.03931179

[1] 0.6229382

[1] 7.166378

[1] 0.03748001

[1] 0.5769274

[1] 0.005200828

[1] 0.01251716

[1] 1.291366

[1] 10

# Create a 10 x 10 matrix (of 10 rows and 10 columns)

mymat <- matrix(nrow=10, ncol=10)

# For each row and for each column, assign values based on position: product of two indexes

for(i in 1:dim(mymat)[1]) {

for(j in 1:dim(mymat)[2]) {

mymat[i,j] = i\*j

}

}

mymat

Output

[,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]

[1,] 1 2 3 4 5 6 7 8 9 10

[2,] 2 4 6 8 10 12 14 16 18 20

[3,] 3 6 9 12 15 18 21 24 27 30

[4,] 4 8 12 16 20 24 28 32 36 40

[5,] 5 10 15 20 25 30 35 40 45 50

[6,] 6 12 18 24 30 36 42 48 54 60

[7,] 7 14 21 28 35 42 49 56 63 70

[8,] 8 16 24 32 40 48 56 64 72 80

[9,] 9 18 27 36 45 54 63 72 81 90

[10,] 10 20 30 40 50 60 70 80 90 100

# Make a lower triangular matrix (zeroes in upper right corner)

m=10

n=10

# A counter to count the assignment

ctr=0

# Create a 10 x 10 matrix with zeroes

mymat = matrix(0,m,n)

for(i in 1:m) {

for(j in 1:n) {

if(i==j) {

break;

} else {

# you assign the values only when i<>j

mymat[i,j] = i\*j

ctr=ctr+1

}

}

}

# Print how many matrix cells were assigned

print(ctr)

mymat

Output as follows:

[1] 45

[,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]

[1,] 0 0 0 0 0 0 0 0 0 0

[2,] 2 0 0 0 0 0 0 0 0 0

[3,] 3 6 0 0 0 0 0 0 0 0

[4,] 4 8 12 0 0 0 0 0 0 0

[5,] 5 10 15 20 0 0 0 0 0 0

[6,] 6 12 18 24 30 0 0 0 0 0

[7,] 7 14 21 28 35 42 0 0 0 0

[8,] 8 16 24 32 40 48 56 0 0 0

[9,] 9 18 27 36 45 54 63 72 0 0

[10,] 10 20 30 40 50 60 70 80 90 0

**next** discontinues a particular iteration and jumps to the next cycle. It is equivalent to “continue“ in Oracle PL/SQL.

# Display odd numbers.

m=20

for (k in 1:m){

if (!k %% 2)

next

print(k)

}

# R - Functions

A function is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.

In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish their actions.

The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

## Function Definition

An R function is created by using the keyword **function**. The basic syntax of an R function definition is as follows −

function\_name <- function(arg\_1, arg\_2, ...) {

Function body

}

## Function Components

The different parts of a function are −

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

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

## Built-in Function

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

# Create a sequence of numbers from 1 to 30.

print(seq(1,30))

# Create a sequence of numbers from 1 to 30 with diffrence of 05.

print(seq(0,30,5))

# Find mean of numbers from 25 to 50.

print(mean(25:50))

# Find sum of numbers frm 1 to 10.

print(sum(1:10))

When we execute the above code, it produces the following result −

[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

[1] 0 5 10 15 20 25 30

[1] 37.5

[1] 55

## User-defined Function

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function is created and used.

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

rm(square.function)

square.function <- function(a) {

for(i in 1:a) {

b <- i^2

print(b)

}

}

## Calling a Function

# Call the function square.function supplying 5 as an argument.

square.function(5)

When we execute the above code, it produces the following result −

[1] 1

[1] 4

[1] 9

[1] 16

[1] 25

### Calling a Function without an Argument

# Create a function without an argument.

square.function <- function() {

for(i in 1:5) {

print(i^2)

}

}

# Call the function without supplying an argument.

square.function

square.function()

When we execute the above code, it produces the following result −

square.function

function() {

for(i in 1:5) {

print(i^2)

}

}

[1] 1

[1] 4

[1] 9

[1] 16

[1] 25

### Calling a Function with Argument Values (by position and by name)

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

# Create a function with arguments.

emp.function <- function(empcode,ename,location) {

print(paste(empcode,' is the emp code for Mr.', ename, ' work from ', location, ' location' ))

}

emp.function(114649,'Hari Yadav','Noida')

emp.function('Noida','Hari Yadav',114649)

emp.function(location = 'Noida',ename = 'Hari Yadav',empcode = 114649)

When we execute the above code, it produces the following result −

[1] "114649 is the emp code for Mr. Hari Yadav work from Noida location"

[1] "Noida is the emp code for Mr. Hari Yadav work from 114649 location"

[1] "114649 is the emp code for Mr. Hari Yadav work from Noida location"

### 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.

emp.function <- function(empcode,ename,location, title = 'Executive') {

print(paste(empcode,' is the emp code for Mr.', ename, ' work from ', location, ' location', ' his designation is ', title))

}

emp.function(114649,'Hari Yadav','Noida','TL')

emp.function(114649,'Hari Yadav','Noida')

When we execute the above code, it produces the following result −

[1] "114649 is the emp code for Mr. Hari Yadav work from Noida location his designation is TL"

[1] "114649 is the emp code for Mr. Hari Yadav work from Noida location his designation is Executive"

### 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)

When we execute the above code, it produces the following result −

[1] 36

[1] 6

Error in print(b) : argument "b" is missing, with no default

# R - Strings

Any value written within a pair of single quote or double quotes in R 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 both double quotes or both single quote. They can not 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 can not be inserted into a string starting and ending with double quotes.
* Single quote can not be inserted into a string starting and ending with single quote.

### Examples of Valid Strings

Following examples clarify the rules about creating a string in R.

a <- 'Start and end with single quote'

print(a)

b <- "Start and end with double quotes"

print(b)

c <- "single quote ' in between double quotes"

print(c)

d <- 'Double quotes " in between single quote'

print(d)

When the above code is run we get the following 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

e <- 'Mixed quotes"

print(e)

f <- 'Single quote ' inside single quote'

print(f)

g <- "Double quotes " inside double quotes"

print(g)

When we run the script it fails giving below results.

Error: unexpected symbol in:

"print(e)

f <- 'Single"

Execution halted

## String Manipulation

### 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

The basic syntax for paste function is −

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

paste0(..., collapse = NULL)

Following is the description of the parameters used −

* **...** 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 <- 'Good'

c <- 'Morning'

d <- "India"

print(paste(a,b,c,d))

print(paste(a,b,c,d, sep = "-"))

print(paste(a,b,c,d, sep = "", collapse = NULL))

print(paste(a,b,c,d, sep = "", collapse = "")) # same

Try to separate by tab.

When we execute the above code, it produces the following result −

[1] "Hello Good Morning India"

[1] "Hello-Good-Morning-India"

[1] "HelloGoodMorningIndia"

[1] "HelloGoodMorningIndia"

Difference between paste() and paste0()

paste (1:10, 1:10)

paste0(1:10, 1:10)

[1] "1 1" "2 2" "3 3" "4 4" "5 5" "6 6" "7 7" "8 8" "9 9" "10 10"

[1] "11" "22" "33" "44" "55" "66" "77" "88" "99" "1010"

## If you pass several vectors to paste0 or paste, they are concatenated in a vectorized way.

paste0(1:12, c("st", "nd", "rd", rep("th", 9)))

[1] "1st" "2nd" "3rd" "4th" "5th" "6th" "7th" "8th" "9th" "10th" "11th" "12th"

## paste works the same, but separates each input with a space.

paste(1:12, c("st", "nd", "rd", rep("th", 9)))

[1] "1 st" "2 nd" "3 rd" "4 th" "5 th" "6 th" "7 th" "8 th" "9 th" "10 th" "11 th" "12 th"

## Notice that the recycling rules make every input as long as the longest input.

paste0(1:15, c("st", "nd", "rd", rep("th", 9)))

[1] "1st" "2nd" "3rd" "4th" "5th" "6th" "7th" "8th" "9th" "10th" "11th" "12th" "13st" "14nd"

[15] "15rd"

paste(1:15, c("st", "nd", "rd", rep("th", 9)))

[1] "1 st" "2 nd" "3 rd" "4 th" "5 th" "6 th" "7 th" "8 th" "9 th" "10 th" "11 th" "12 th"

[13] "13 st" "14 nd" "15 rd"

**Consolidated example.**

## When passing a single vector, paste0 and paste work like as.character.

paste0(1:12)

paste(1:12) # same

as.character(1:12) # same

## If you pass several vectors to paste0, they are concatenated in a vectorized way.

(nth <- paste0(1:12, c("st", "nd", "rd", rep("th", 9))))

## paste works the same, but separates each input with a space.

paste(1:12, c("st", "nd", "rd", rep("th", 9)))

## Notice that the recycling rules make every input as long as the longest input.

paste(month.abb, "is the", nth, "month of the year.")

paste(month.abb, letters)

## You can change the separator by passing a sep argument

## which can be multiple characters.

paste(month.abb, "is the", nth, "month of the year.", sep = "\_\*\_")

## To collapse the output into a single string, pass a collapse argument.

paste0(nth, collapse = ", ")

## For inputs of length 1, use the sep argument rather than collapse

paste("1st", "2nd", "3rd", collapse = NULL) # probably not what you wanted

paste("1st", "2nd", "3rd", sep = ", ")

## You can combine the sep and collapse arguments together.

paste(month.abb, nth, sep = ": ", collapse = "; ")

### Formatting numbers & strings - format() function

Numbers and strings can be formatted to a specific style using **format()**function. Format treats everything as a string.

### Syntax

The basic syntax for format function is −

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

Following is the description of the parameters used −

* **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.
* **justify** 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)

When we execute the above code, it produces the following result −

[1] "23.1234568"

[1] "6.000000e+00" "1.314521e+01"

[1] "23.47000"

[1] "6"

[1] " 13.7"

[1] "Hello "

[1] " Hello "

### Counting number of characters in a string - nchar() function

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

### Syntax

The basic syntax for nchar() function is −

nchar(x)

Following is the description of the parameters used −

* **x** is the vector input.

### Example

result <- nchar("Count the number of characters")

print(result)

When we execute the above code, it produces the following result −

[1] 30

### Changing the case - toupper() & tolower() functions

These functions change the case of characters of a string.

### Syntax

The basic syntax for toupper() & tolower() function is −

toupper(x)

tolower(x)

Following is the description of the parameters used −

* **x** is the vector input.

### Example

# Changing to Upper case.

result <- toupper("Changing To Upper")

print(result)

# Changing to lower case.

result <- tolower("Changing To Lower")

print(result)

When we execute the above code, it produces the following result −

[1] "CHANGING TO UPPER"

[1] "changing to lower"

### Extracting parts of a string - substring() function

This function extracts parts of a String.

### Syntax

The basic syntax for substring() function is −

substring(x,first,last)

Following is the description of the parameters used −

* **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)

result <- sub("my","our","India is my country.")

print(result)

When we execute the above code, it produces the following result −

[1] "act"

"India is our country."