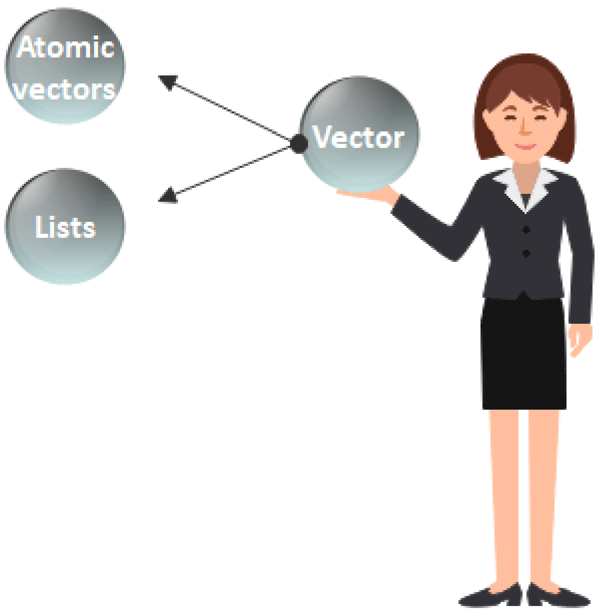
R Vector

A **vector** is a basic data structure which plays an important role in R programming.

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

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

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



There is only one difference between atomic vectors and lists. In an atomic vector, all the elements are of the same type, but in the list, the elements are of different data types. In this section, we will discuss only the atomic vectors. We will discuss lists briefly in the next topic.

How to create a vector in R?

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

1) Using the colon(:) operator

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

1. z<-x:y

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

Example:

1. a<-4:-10
2. a

Output

```
[1]  4  3  2  1  0  -1  -2  -3  -4  -5  -6  -7  -8  -9  -10
```

2) Using the seq() function

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

Example:

1. seq_vec<-seq(1,4,by=0.5)
2. seq_vec
3. class(seq_vec)

Output

```
[1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0
```

Example:

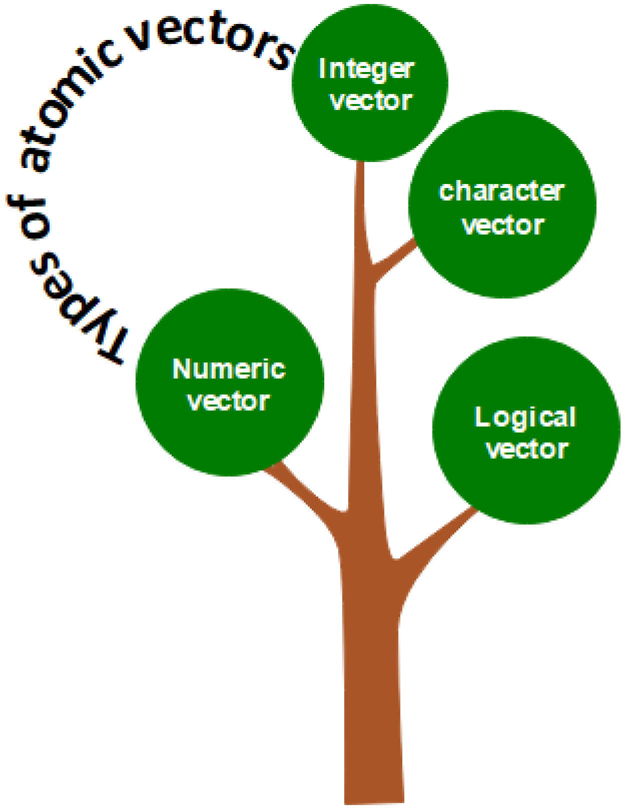
1. seq_vec<-seq(1,4,length.out=6)
2. seq_vec
3. class(seq_vec)

Output

```
[1]  1.0  1.6  2.2  2.8  3.4  4.0
[1]  "numeric"
```

Atomic vectors in R

In R, there are four types of atomic vectors. Atomic vectors play an important role in Data Science. Atomic vectors are created with the help of **c()** function. These atomic vectors are as follows:



Numeric vector

The decimal values are known as numeric data types in R. If we assign a decimal value to any variable d, then this d variable will become a numeric type. A vector which contains numeric elements is known as a numeric vector.

Example:

1. d<-45.5
2. num_vec<-c(10.1, 10.2, 33.2)
3. d
4. num_vec
5. class(d)
6. class(num_vec)

Output

```
[1]   45.5
[1]   10.1   10.2   33.2
```

```
[1]   "numeric"
[1]   "numeric"
```

Integer vector

A non-fraction numeric value is known as integer data. This integer data is represented by "Int." The Int size is 2 bytes and long Int size of 4 bytes. There is two way to assign an integer value to a variable, i.e., by using as.integer() function and appending of L to the value.

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

Example:

1. d<-as.integer(5)
2. e<-5L
3. int_vec<-c(1,2,3,4,5)
4. int_vec<-as.integer(int_vec)
5. int_vec1<-c(1L,2L,3L,4L,5L)
6. class(d)
7. class(e)
8. class(int_vec)
9. class(int_vec1)

Output

```
[1]   "integer"
[1]   "integer"
[1]   "integer"
[1]   "integer"
```

Character vector

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

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

Example:

```
1. d<-'shubham'
2. e<-"Arpita"
3. f<-65
4. f<-as.character(f)
5. d
6. e
7. f
8. char_vec<-c(1,2,3,4,5)
9. char_vec<-as.character(char_vec)
10. char_vec1<-c("shubham","arpita","nishka","vaishali")
11. char_vec
12. class(d)
13. class(e)
14. class(f)
15. class(char_vec)
16. class(char_vec1)
```

Output

```
[1]   "shubham"
[1]   "Arpita"
[1]   "65"
[1]   "1"   "2"   "3"   "4"   "5"
[1]   "shubham"   "arpita"   "nishka"   "vaishali"
[1]   "character"
[1]   "character"
[1]   "character"
[1]   "character"
[1]   "character"
```

Logical vector

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

Example:

```
1. d<-as.integer(5)
2. e<-as.integer(6)
```

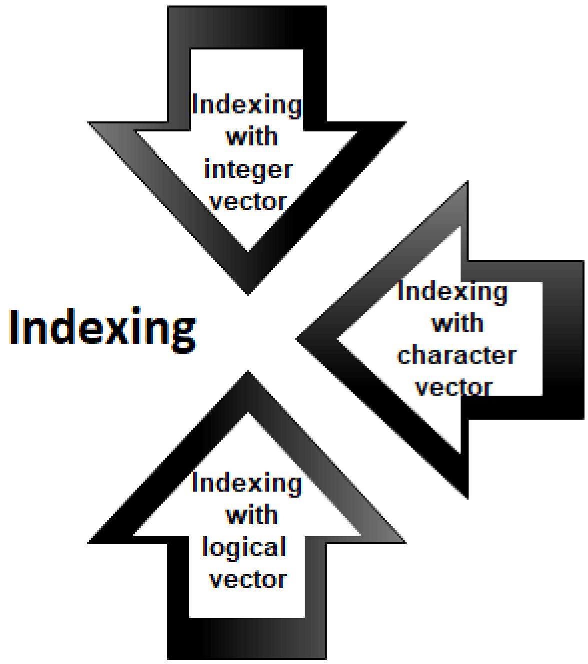
3. f<-as.integer(7)
4. g<-d>e
5. h<-e<f
6. g
7. h
8. log_vec<-c(d<e, d<f, e<d,e<f,f<d,f<e)
9. log_vec
10. class(g)
11. class(h)
12. class(log_vec)

Output

```
[1]   FALSE
[1]   TRUE
[1]   TRUE   TRUE   FALSE   TRUE   FALSE   FALSE
[1]   "logical"
[1]   "logical"
[1]   "logical"
```

Accessing elements of vectors

We can access the elements of a vector with the help of vector indexing. Indexing denotes the position where the value in a vector is stored. Indexing will be performed with the help of integer, character, or logic.



1) Indexing with integer vector

On integer vector, indexing is performed in the same way as we have applied in C, C++, and java. There is only one difference, i.e., in C, C++, and java the indexing starts from 0, but in R, the indexing starts from 1. Like other programming languages, we perform indexing by specifying an integer value in square braces [] next to our vector.

Example:

1. seq_vec<-seq(1,4,length.out=6)
2. seq_vec
3. seq_vec[2]

Output

```
[1] 1.0 1.6 2.2 2.8 3.4 4.0
[1] 1.6
```

2) Indexing with a character vector

In character vector indexing, we assign a unique key to each element of the vector. These keys are uniquely defined as each element and can be accessed very easily. Let's see an example to understand how it is performed.

Example:

1. char_vec<-c("shubham"=22,"arpita"=23,"vaishali"=25)
2. char_vec
3. char_vec["arpita"]

Output

```
shubham   arpita vaishali
   22       23      25
arpita
      23
```

3) Indexing with a logical vector

In logical indexing, it returns the values of those positions whose corresponding position has a logical vector TRUE. Let see an example to understand how it is performed on vectors.

Example:

1. a<-c(1,2,3,4,5,6)
2. a[c(TRUE,FALSE,TRUE,TRUE,FALSE,TRUE)]

Output

```
[1] 1 3 4 6
```

Vector Operation

In R, there are various operation which is performed on the vector. We can add, subtract, multiply or divide two or more vectors from each other. In data science, R plays an important role, and operations are required for data manipulation. There are the following types of operation which are performed on the vector.

1) Combining vectors

The c() function is not only used to create a vector, but also it is also used to combine two vectors. By combining one or more vectors, it forms a new vector which contains all the elements of each vector. Let see an example to see how c() function combines the vectors.

Example:

1. p<-c(1,2,4,5,7,8)
2. q<-c("shubham","arpita","nishka","gunjan","vaishali","sumit")
3. r<-c(p,q)

Output

```
[1] "1"        "2"       "4"       "5"       "7"        "8"
[7] "shubham"  "arpita"  "nishka"  "gunjan"  "vaishali" "sumit"
```

2) Arithmetic operations

We can perform all the arithmetic operation on vectors. The arithmetic operations are performed member-by-member on vectors. We can add, subtract, multiply, or divide two vectors. Let see an example to understand how arithmetic operations are performed on vectors.

Example:

1. a<-c(1,3,5,7)
2. b<-c(2,4,6,8)
3. a+b
4. a-b
5. a/b
6. a%%b

Output

```
[1]  3  7 11 15
[1] -1 -1 -1 -1
[1]  2 12 30 56
[1] 0.5000000 0.7500000 0.8333333 0.8750000
[1] 1 3 5 7
```

3) Logical Index vector

With the help of the logical index vector in R, we can form a new vector from a given vector. This vector has the same length as the original vector. The vector members are TRUE only when the corresponding members of the original vector are included in the slice; otherwise, it will be false. Let see an example to understand how a new vector is formed with the help of logical index vector.

Example:

1. a<-c("Shubham","Arpita","Nishka","Vaishali","Sumit","Gunjan")
2. b<-c(TRUE,FALSE,TRUE,TRUE,FALSE,FALSE)
3. a[b]

Output

```
[1] "Shubham"  "Nishka"   "Vaishali"
```

4) Numeric Index

In R, we specify the index between square braces [ ] for indexing a numerical value. If our index is negative, it will return us all the values except for the index which we have specified. For example, specifying [-3] will prompt R to convert -3 into its absolute value and then search for the value which occupies that index.

Example:

1. q<-c("shubham","arpita","nishka","gunjan","vaishali","sumit")
2. q[2]
3. q[-4]
4. q[15]

Output

```
[1] "arpita"
[1] "shubham"  "arpita"   "nishka"   "vaishali" "sumit"
[1] NA
```

5) Duplicate Index

An index vector allows duplicate values which means we can access one element twice in one operation. Let see an example to understand how duplicate index works.

Example:

1. q<-c("shubham","arpita","nishka","gunjan","vaishali","sumit")
2. q[c(2,4,4,3)]

Output

```
[1] "arpita" "gunjan" "gunjan" "nishka"
```

6) Range Indexes

Range index is used to slice our vector to form a new vector. For slicing, we used colon(:) operator. Range indexes are very helpful for the situation involving a large operator. Let see an example to understand how slicing is done with the help of the colon operator to form a new vector.

Example:

```
1. q<-c("shubham","arpita","nishka","gunjan","vaishali","sumit")
2. b<-q[2:5]
3. b
```

Output

```
[1] "arpita"   "nishka"   "gunjan"   "vaishali"
```

7) Out-of-order Indexes

In R, the index vector can be out-of-order. Below is an example in which a vector slice with the order of first and second values reversed.

Example:

```
1. q<-c("shubham","arpita","nishka","gunjan","vaishali","sumit")b<-q[2:5]
2. q[c(2,1,3,4,5,6)]
```

Output

```
[1] "arpita"   "shubham"  "nishka"   "gunjan"   "vaishali" "sumit"
```

8) Named vectors members

We first create our vector of characters as:

```
1. z=c("TensorFlow","PyTorch")
2. z
```

Output

```
[1] "TensorFlow" "PyTorch"
```

Once our vector of characters is created, we name the first vector member as "Start" and the second member as "End" as:

```
1. names(z)=c("Start","End")
2. z
```

Output

```
Start              End
"TensorFlow"     "PyTorch"
```

We retrieve the first member by its name as follows:

1. z["Start"]

Output

```
Start
"TensorFlow"
```

We can reverse the order with the help of the character string index vector.

1. z[c("Second","First")]

Output

```
     Second           First
"PyTorch"     "TensorFlow"
```

Applications of vectors

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

R Lists

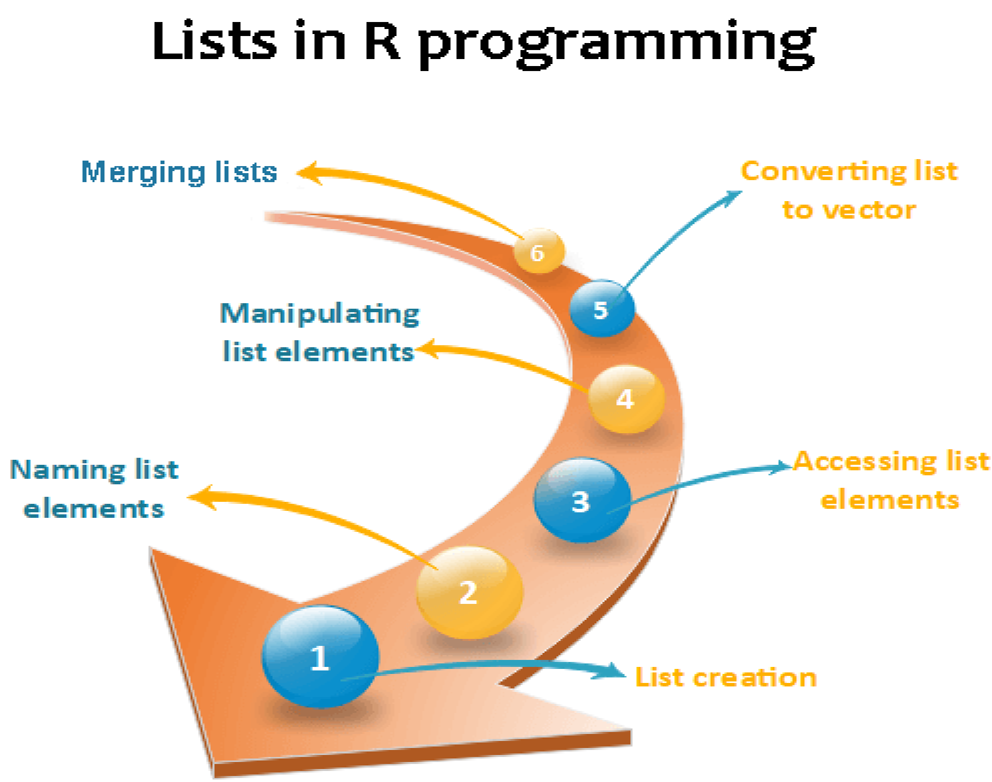
In R, lists are the second type of vector. Lists are the objects of R which contain elements of different types such as number, vectors, string and another list inside it. It can also contain a function or a matrix as its elements. A list is a data structure which has components of mixed data types. We can say, a list is a generic vector which contains other objects.

Example

1. vec <- c(3,4,5,6)
2. char_vec<-c("shubham","nishka","gunjan","sumit")
3. logic_vec<-c(TRUE,FALSE,FALSE,TRUE)
4. out_list<-list(vec,char_vec,logic_vec)
5. out_list

Output:

```
[[1]]
[1] 3 4 5 6
[[2]]
[1] "shubham" "nishka"  "gunjan"  "sumit"
[[3]]
[1]  TRUE FALSE FALSE  TRUE
```



Lists creation

The process of creating a list is the same as a vector. In R, the vector is created with the help of c() function. Like c() function, there is another function, i.e., list() which is used to create a list in R. A list avoid the drawback of the vector which is data type. We can add the elements in the list of different data types.

Syntax

1. list()

Example 1: Creating list with same data type

1. list_1<-list(1,2,3)
2. list_2<-list("Shubham","Arpita","Vaishali")
3. list_3<-list(c(1,2,3))
4. list_4<-list(TRUE,FALSE,TRUE)
5. list_1
6. list_2
7. list_3
8. list_4

Output:

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

[[1]]
[1] "Shubham"
[[2]]
[1] "Arpita"
[[3]]
[1] "Vaishali"

[[1]]
[1] 1 2 3

[[1]]
[1] TRUE
[[2]]
[1] FALSE
```

```
[[3]]
[1] TRUE
```

Example 2: Creating the list with different data type

1. list_data<-list("Shubham","Arpita",c(1,2,3,4,5),TRUE,FALSE,22.5,12L)
2. print(list_data)

In the above example, the list function will create a list with character, logical, numeric, and vector element. It will give the following output

Output:

```
[[1]]
[1] "Shubham"
[[2]]
[1] "Arpita"
[[3]]
[1] 1 2 3 4 5
[[4]]
[1] TRUE
[[5]]
[1] FALSE
[[6]]
[1] 22.5
[[7]]
[1] 12
```

Giving a name to list elements

R provides a very easy way for accessing elements, i.e., by giving the name to each element of a list. By assigning names to the elements, we can access the element easily. There are only three steps to print the list data corresponding to the name:

1. Creating a list.
2. Assign a name to the list elements with the help of names() function.
3. Print the list data.

Let see an example to understand how we can give the names to the list elements.

Example

1. # Creating a list containing a vector, a matrix and a list.
2. list_data <- list(c("Shubham","Nishka","Gunjan"), matrix(c(40,80,60,70,90,80), nrow = 2),

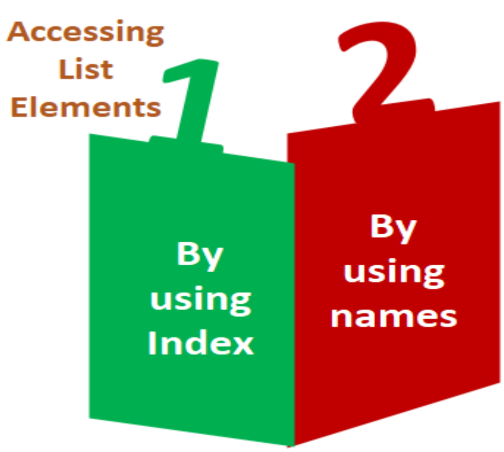
3.   list("BCA","MCA","B.tech"))
4. # Giving names to the elements in the list.
5. names(list_data) <- c("Students", "Marks", "Course")
6. # Show the list.
7. print(list_data)

Output:

```
$Students
[1] "Shubham" "Nishka"  "Gunjan"

$Marks
     [,1] [,2] [,3]
[1,]   40   60   90
[2,]   80   70   80

$Course
$Course[[1]]
[1] "BCA"

$Course[[2]]
[1] "MCA"

$Course[[3]]
[1] "B. tech."
```

Accessing List Elements

R provides two ways through which we can access the elements of a list. First one is the indexing method performed in the same way as a vector. In the second one, we can access the elements of a list with the help of names. It will be possible only with the named list.; we cannot access the elements of a list using names if the list is normal.



Let see an example of both methods to understand how they are used in the list to access elements.

Example 1: Accessing elements using index

1. # Creating a list containing a vector, a matrix and a list.
2. list_data <- list(c("Shubham","Arpita","Nishka"), matrix(c(40,80,60,70,90,80), nrow = 2),
3.   list("BCA","MCA","B.tech"))
4. # Accessing the first element of the list.
5. print(list_data[1])
6. # Accessing the third element. The third element is also a list, so all its elements will be p rinted.
7. print(list_data[3])

Output:

```
[[1]]
[1] "Shubham" "Arpita"  "Nishka"

[[1]]
[[1]][[1]]
[1] "BCA"

[[1]][[2]]
[1] "MCA"

[[1]][[3]]
[1] "B.tech"
```

Example 2: Accessing elements using names

1. # Creating a list containing a vector, a matrix and a list.
2. list_data <- list(c("Shubham","Arpita","Nishka"), matrix(c(40,80,60,70,90,80), nrow = 2),list ("BCA","MCA","B.tech"))
3. # Giving names to the elements in the list.
4. names(list_data) <- c("Student", "Marks", "Course")
5. # Accessing the first element of the list.
6. print(list_data["Student"])
7. print(list_data$Marks)
8. print(list_data)

Output:

```
$Student
[1] "Shubham" "Arpita"  "Nishka"

        [,1] [,2] [,3]
[1,]    40   60   90
[2,]    80   70   80

$Student
[1] "Shubham" "Arpita"  "Nishka"

$Marks
     [,1] [,2] [,3]
[1,]    40   60   90
[2,]    80   70   80

$Course
$Course[[1]]
[1] "BCA"
$Course[[2]]
[1] "MCA"
$Course[[3]]
[1] "B. tech."
```

Manipulation of list elements

R allows us to add, delete, or update elements in the list. We can update an element of a list from anywhere, but elements can add or delete only at the end of the list. To remove an element from a specified index, we will assign it a null value. We can update the element of a list by overriding it from the new value. Let see an example to understand how we can add, delete, or update the elements in the list.

Example

1. # Creating a list containing a vector, a matrix and a list.
2. list_data <- list(c("Shubham","Arpita","Nishka"), matrix(c(40,80,60,70,90,80), nrow = 2),
3.   list("BCA","MCA","B.tech"))
4. # Giving names to the elements in the list.
5. names(list_data) <- c("Student", "Marks", "Course")
6. # Adding element at the end of the list.
7. list_data[4] <- "Moradabad"
8. print(list_data[4])
9. # Removing the last element.
10. list_data[4] <- NULL

```
# Printing the 4th Element.
print(list_data[4])
# Updating the 3rd Element.
list_data[3] <- "Masters of computer applications"
print(list_data[3])
```

Output:

```
[[1]]
[1] "Moradabad"

$<NA>
NULL

$Course
[1] "Masters of computer applications"
```

Converting list to vector

There is a drawback with the list, i.e., we cannot perform all the arithmetic operations on list elements. To remove this, drawback R provides unlist() function. This function converts the list into vectors. In some cases, it is required to convert a list into a vector so that we can use the elements of the vector for further manipulation.

The unlist() function takes the list as a parameter and change into a vector. Let see an example to understand how to unlist() function is used in R.

Example

```
# Creating lists.
list1 <- list(10:20)
print(list1)
list2 <-list(5:14)
print(list2)
# Converting the lists to vectors.
v1 <- unlist(list1)
v2 <- unlist(list2)
print(v1)
print(v2)
adding the vectors
```

12. result <- v1+v2
13. 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
```

Merging Lists

R allows us to merge one or more lists into one list. Merging is done with the help of the list() function also. To merge the lists, we have to pass all the lists into list function as a parameter, and it returns a list which contains all the elements which are present in the lists. Let see an example to understand how the merging process is done.

Example

1. # Creating two lists.
2. Even_list <- list(2,4,6,8,10)
3. Odd_list <- list(1,3,5,7,9)
4. # Merging the two lists.
5. merged.list <- list(Even_list,Odd_list)
6. # Printing the merged list.
7. print(merged.list)

Output:

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

[[1]][[2]]
[1] 4

[[1]][[3]]
[1] 6
```

```
[[1]][[4]]
[1] 8

[[1]][[5]]
[1] 10


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

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

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

[[2]][[4]]
[1] 7

[[2]][[5]]
[1] 9
```

R Arrays

In R, arrays are the data objects which allow us to store data in more than two dimensions. In R, an array is created with the help of the **array()** function. This array() function takes a vector as an input and to create an array it uses vectors values in the **dim** parameter.

For example- if we will create an array of dimension (2, 3, 4) then it will create 4 rectangular matrices of 2 row and 3 columns.

R Array Syntax

There is the following syntax of R arrays:

1. array_name <- array(data, dim= (row_size, column_size, matrices, dim_names))

data

The data is the first argument in the array() function. It is an input vector which is given to the array.

matrices

In R, the array consists of multi-dimensional matrices.

row_size

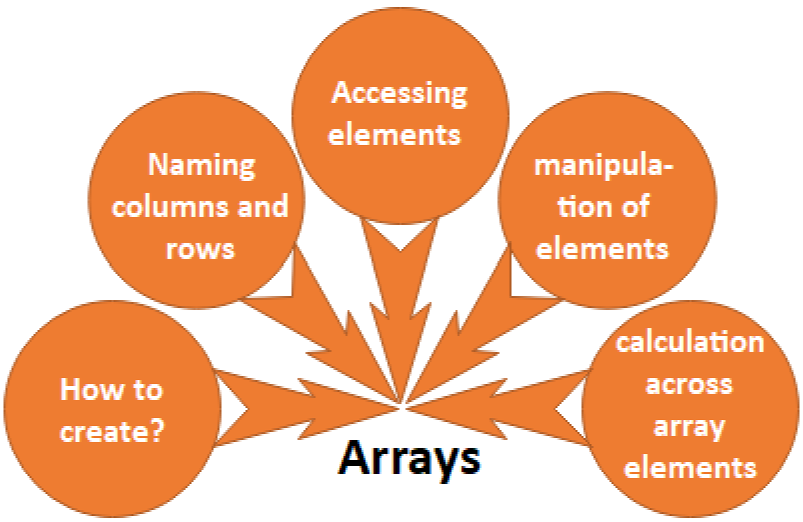
This parameter defines the number of row elements which an array can store.

column_size

This parameter defines the number of columns elements which an array can store.

dim_names

This parameter is used to change the default names of rows and columns.



How to create?

In R, array creation is quite simple. We can easily create an array using vector and array() function. In array, data is stored in the form of the matrix. There are only two steps to create a matrix which are as follows

1. In the first step, we will create two vectors of different lengths.
2. Once our vectors are created, we take these vectors as inputs to the array.

Let see an example to understand how we can implement an array with the help of the vectors and array() function.

Example

1. #Creating two vectors of different lengths
2. vec1 <-c(1,3,5)
3. vec2 <-c(10,11,12,13,14,15)
4. #Taking these vectors as input to the array
5. res <- array(c(vec1,vec2),dim=c(3,3,2))
6. print(res)

Output

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

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

Naming rows and columns

In R, we can give the names to the rows, columns, and matrices of the array. This is done with the help of the dim name parameter of the array() function.

It is not necessary to give the name to the rows and columns. It is only used to differentiate the row and column for better understanding.

Below is an example, in which we create two arrays and giving names to the rows, columns, and matrices.

Example

1. #Creating two vectors of different lengths
2. vec1 <-c(1,3,5)
3. vec2 <-c(10,11,12,13,14,15)
4. #Initializing names for rows, columns and matrices
5. col_names <- c("Col1","Col2","Col3")
6. row_names <- c("Row1","Row2","Row3")
7. matrix_names <- c("Matrix1","Matrix2")

```
#Taking the vectors as input to the array
res <- array(c(vec1,vec2),dim=c(3,3,2),dimnames=list(row_names,col_names,matrix_names))
print(res)
```

Output

```
, , Matrix1

     Col1 Col2 Col3
Row1    1   10   13
Row2    3   11   14
Row3    5   12   15

, , Matrix2

     Col1 Col2 Col3
Row1    1   10   13
Row2    3   11   14
Row3    5   12   15
```

Accessing array elements

Like C or C++, we can access the elements of the array. The elements are accessed with the help of the index. Simply, we can access the elements of the array with the help of the indexing method. Let see an example to understand how we can access the elements of the array using the indexing method.

Example

```
, , Matrix1
     Col1 Col2 Col3
Row1    1   10   13
Row2    3   11   14
Row3    5   12   15

, , Matrix2
     Col1 Col2 Col3
Row1    1   10   13
Row2    3   11   14
Row3    5   12   15
```

```
 Col1 Col2 Col3
   5   12   15
 [1] 13
      Col1 Col2 Col3
Row1    1   10   13
Row2    3   11   14
Row3    5   12   15
```

Manipulation of elements

The array is made up matrices in multiple dimensions so that the operations on elements of an array are carried out by accessing elements of the matrices.

Example

```
#Creating two vectors of different lengths
vec1 <-c(1,3,5)
vec2 <-c(10,11,12,13,14,15)
#Taking the vectors as input to the array1
res1 <- array(c(vec1,vec2),dim=c(3,3,2))
print(res1)
#Creating two vectors of different lengths
vec1 <-c(8,4,7)
vec2 <-c(16,73,48,46,36,73)
 #Taking the vectors as input to the array2
res2 <- array(c(vec1,vec2),dim=c(3,3,2))
print(res2)
 #Creating matrices from these arrays
mat1 <- res1[,,2]
mat2 <- res2[,,2]
res3 <- mat1+mat2
print(res3)
```

Output

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

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

, , 1
     [,1] [,2] [,3]
[1,]    8   16   46
[2,]    4   73   36
[3,]    7   48   73

, , 2
     [,1] [,2] [,3]
[1,]    8   16   46
[2,]    4   73   36
[3,]    7   48   73


     [,1] [,2] [,3]
[1,]    9   26   59
[2,]    7   84   50
[3,]   12   60   88
```

Calculations across array elements

For calculation purpose, r provides **apply()** function. This apply function contains three parameters i.e., x, margin, and function.

This function takes the array on which we have to perform the calculations. The basic syntax of the apply() function is as follows:

1. apply(x, margin, fun)

Here, x is an array, and a margin is the name of the dataset which is used and fun is the function which is to be applied to the elements of the array.

Example

1. #Creating two vectors of different lengths

2. vec1 <-c(1,3,5)
3. vec2 <-c(10,11,12,13,14,15)
4. #Taking the vectors as input to the array1
5. res1 <- array(c(vec1,vec2),dim=c(3,3,2))
6. print(res1)
7. #using apply function
8. result <- apply(res1,c(1),sum)
9. print(result)

Output

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

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

[1] 48 56 64
```

R Matrix

In R, a two-dimensional rectangular data set is known as a matrix. A matrix is created with the help of the vector input to the matrix function. On R matrices, we can perform addition, subtraction, multiplication, and division operation.

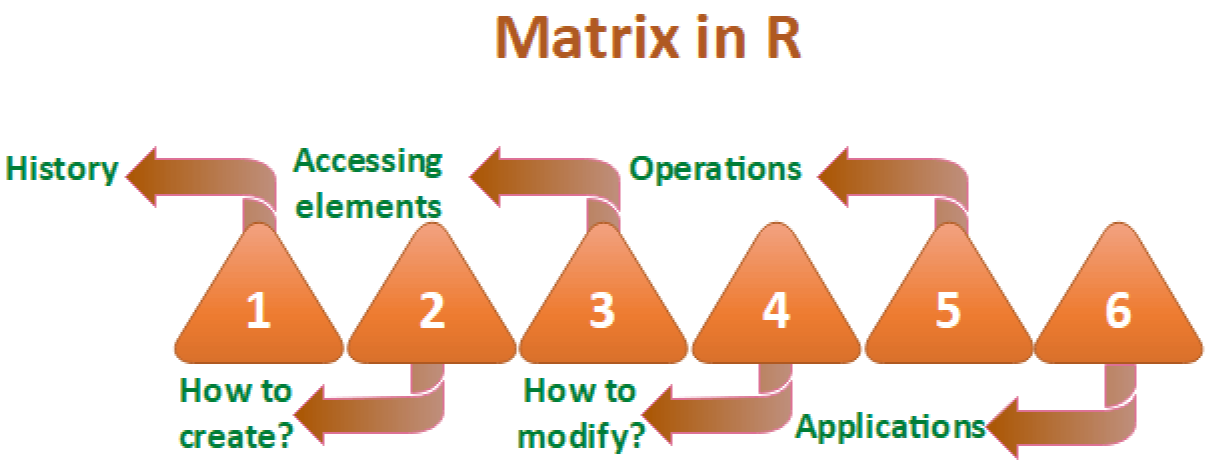
In the R matrix, elements are arranged in a fixed number of rows and columns. The matrix elements are the real numbers. In R, we use matrix function, which can easily reproduce the memory representation of the matrix. In the R matrix, all the elements must share a common basic type.

Example

1. matrix1<-matrix(c(11, 13, 15, 12, 14, 16),nrow =2, ncol =3, byrow = TRUE)
2. matrix1

Output

```
[,1]   [,2]   [,3]
[1,]    11     13     15
[2,]    12     14     16
```



History of matrices in R

The word "Matrix" is the Latin word for womb which means a place where something is formed or produced. Two authors of historical importance have used the word "Matrix" for unusual ways. They proposed this axiom as a means to reduce any function to one of the lower types so that at the "bottom" (0order) the function is identical to its extension.

Any possible function other than a matrix from the matrix holds true with the help of the process of generalization. It will be true only when the proposition (which asserts function in question) is true. It will hold true for all or one of the value of argument only when the other argument is undetermined.

How to create a matrix in R?

Like vector and list, R provides a function which creates a matrix. R provides the matrix() function to create a matrix. This function plays an important role in data analysis. There is the following syntax of the matrix in R:

1. matrix(data, nrow, ncol, byrow, dim_name)

data

The first argument in matrix function is data. It is the input vector which is the data elements of the matrix.

nrow

The second argument is the number of rows which we want to create in the matrix.

ncol

The third argument is the number of columns which we want to create in the matrix.

byrow

The byrow parameter is a logical clue. If its value is true, then the input vector elements are arranged by row.

dim_name

The dim_name parameter is the name assigned to the rows and columns.

Let's see an example to understand how matrix function is used to create a matrix and arrange the elements sequentially by row or column.

Example

```
1.  #Arranging elements sequentially by row.
2.  P <- matrix(c(5:16), nrow = 4, byrow = TRUE)
3.  print(P)
4.  # Arranging elements sequentially by column.
5.  Q <- matrix(c(3:14), nrow = 4, byrow = FALSE)
6.  print(Q)
7.  # Defining the column and row names.
8.  row_names = c("row1", "row2", "row3", "row4")
9.  ccol_names = c("col1", "col2", "col3")
10. R <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(row_names, col_names))
11. print(R)
```

Output

```
       [,1] [,2] [,3]
[1,]     5    6    7
[2,]     8    9   10
[3,]    11   12   13
[4,]    14   15   16

       [,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 matrix elements in R

Like C and C++, we can easily access the elements of our matrix by using the index of the element. There are three ways to access the elements from the matrix.

1. We can access the element which presents on nth row and mth column.
2. We can access all the elements of the matrix which are present on the nth row.
3. We can also access all the elements of the matrix which are present on the mth column.

Let see an example to understand how elements are accessed from the matrix present on nth row mth column, nth row, or mth column.

Example

1. # Defining the column and row names.
2. row_names = c("row1", "row2", "row3", "row4")
3. ccol_names = c("col1", "col2", "col3")
4. #Creating matrix
5. R <- matrix(c(5:16), nrow = 4, byrow = TRUE, dimnames = list(row_names, col_names))
6. print(R)
7. #Accessing element present on 3rd row and 2nd column
8. print(R[3,2])

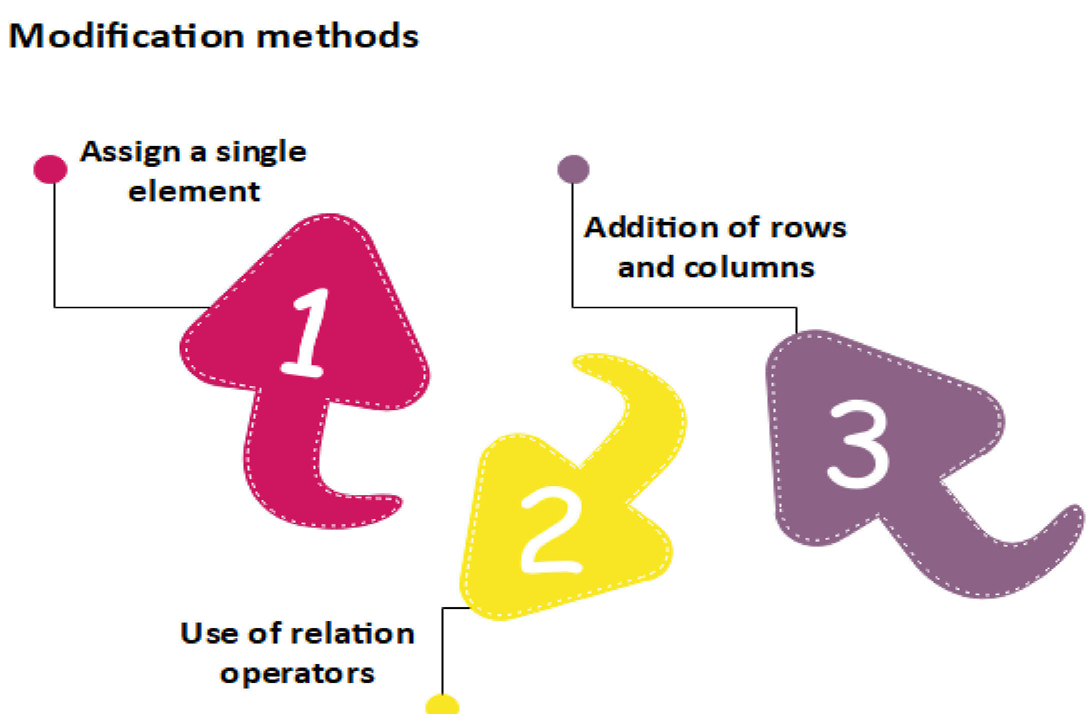
```
#Accessing element present in 3rd row
print(R[3,])
#Accessing element present in 2nd column
print(R[,2])
```

Output

```
      col1 col2 col3
row1     5    6    7
row2     8    9   10
row3    11   12   13
row4    14   15   16

[1] 12

col1 col2 col3
  11   12   13

row1 row2 row3 row4
   6    9   12   15
```

Modification of the matrix

R allows us to do modification in the matrix. There are several methods to do modification in the matrix, which are as follows:



Assign a single element

In matrix modification, the first method is to assign a single element to the matrix at a particular position. By assigning a new value to that position, the old value will get replaced with the new one. This modification technique is quite simple to perform matrix modification. The basic syntax for it is as follows:

1. matrix[n, m]<-y

Here, n and m are the rows and columns of the element, respectively. And, y is the value which we assign to modify our matrix.

Let see an example to understand how modification will be done:

Example

```
1. # Defining the column and row names.
2. row_names = c("row1", "row2", "row3", "row4")
3. ccol_names = c("col1", "col2", "col3")
4. R <- matrix(c(5:16), nrow = 4, byrow = TRUE, dimnames = list(row_names, col_names))
5. print(R)
6. #Assigning value 20 to the element at 3d roe and 2nd column
7. R[3,2]<-20
8. print(R)
```

Output

```
      col1 col2 col3
row1     5    6    7
row2     8    9   10
row3    11   12   13
row4    14   15   16

      col1 col2 col3
row1     5    6    7
row2     8    9   10
row3    11   20   13
row4    14   15   16
```

Use of Relational Operator

R provides another way to perform matrix medication. In this method, we used some relational operators like >, <, ==. Like the first method, the second method is quite simple to use. Let see an example to understand how this method modifies the matrix.

Example 1

1. # Defining the column and row names.
2. row_names = c("row1", "row2", "row3", "row4")
3. ccol_names = c("col1", "col2", "col3")
4. R <- matrix(c(5:16), nrow = 4, byrow = TRUE, dimnames = list(row_names, col_names))
5. print(R)
6. #Replacing element that equal to the 12
7. R[R==12]<-0
8. print(R)

Output

```
     col1 col2 col3
row1    5    6    7
row2    8    9   10
row3   11   12   13
row4   14   15   16

     col1 col2 col3
row1    5    6    7
row2    8    9   10
row3   11    0   13
row4   14   15   16
```

Example 2

1. # Defining the column and row names.
2. row_names = c("row1", "row2", "row3", "row4")
3. ccol_names = c("col1", "col2", "col3")
4. R <- matrix(c(5:16), nrow = 4, byrow = TRUE, dimnames = list(row_names, col_names))
5. print(R)
6. #Replacing elements whose values are greater than 12
7. R[R>12]<-0
8. print(R)

Output

```
      col1 col2 col3
row1     5    6    7
row2     8    9   10
row3    11   12   13
row4    14   15   16

      col1 col2 col3
row1     5    6    7
row2     8    9   10
row3    11   12    0
row4     0    0    0
```

Addition of Rows and Columns

The third method of matrix modification is through the addition of rows and columns using the cbind() and rbind() function. The cbind() and rbind() function are used to add a column and a row respectively. Let see an example to understand the working of cbind() and rbind() functions.

Example 1

```
1. # Defining the column and row names.
2. row_names = c("row1", "row2", "row3", "row4")
3. ccol_names = c("col1", "col2", "col3")
4. R <- matrix(c(5:16), nrow = 4, byrow = TRUE, dimnames = list(row_names, col_names))
5. print(R)
6. #Adding row
7. rbind(R,c(17,18,19))
8. #Adding column
9. cbind(R,c(17,18,19,20))
10. #transpose of the matrix using the t() function:
11. t(R)
12. #Modifying the dimension of the matrix using the dim() function
13. dim(R)<-c(1,12)
14. print(R)
```

Output

```
      col1 col2 col3
row1     5    6    7
row2     8    9   10
row3    11   12   13
row4    14   15   16

      col1 col2 col3
row1     5    6    7
row2     8    9   10
row3    11   12   13
row4    14   15   16
          17   18   19

      col1 col2 col3
row1     5    6    7 17
row2     8    9   10 18
row3    11   12   13 19
row4    14   15   16 20

      row1 row2 row3 row4
col1     5    8   11   14
col2     6    9   12   15
col3     7   10   13   16

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
[1,]    5    8   11   14    6    9   12   15    7    10    13    16
```

Matrix operations

In R, we can perform the mathematical operations on a matrix such as addition, subtraction, multiplication, etc. For performing the mathematical operation on the matrix, it is required that both the matrix should have the same dimensions.

Let see an example to understand how mathematical operations are performed on the matrix.

Example 1

1. R <- matrix(c(5:16), nrow = 4,ncol=3)
2. S <- matrix(c(1:12), nrow = 4,ncol=3)
3. #Addition
4. sum<-R+S
5. print(sum)
6. #Subtraction
7. sub<-R-S
8. print(sub)
9. #Multiplication
10. mul<-R*S
11. print(mul)
12. #Multiplication by constant
13. mul1<-R*12
14. print(mul1)
15. #Division
16. div<-R/S
17. print(div)

Output

```
     [,1] [,2] [,3]
[1,]    6   14   22
[2,]    8   16   24
[3,]   10   18   26
[4,]   12   20   28

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

     [,1] [,2] [,3]
[1,]    5   45  117
[2,]   12   60  140
[3,]   21   77  165
[4,]   32   96  192
```

```
      [,1] [,2] [,3]
[1,]   60  108  156
[2,]   72  120  168
[3,]   84  132  180
[4,]   96  144  192

      [,1]     [,2]      [,3]
[1,] 5.000000 1.800000 1.444444
[2,] 3.000000 1.666667 1.400000
[3,] 2.333333 1.571429 1.363636
[4,] 2.000000 1.500000 1.333333
```

Applications of matrix

1. In geology, Matrices takes surveys and plot graphs, statistics, and used to study in different fields.
2. Matrix is the representation method which helps in plotting common survey things.
3. In robotics and automation, Matrices have the topmost elements for the robot movements.
4. Matrices are mainly used in calculating the gross domestic products in Economics, and it also helps in calculating the capability of goods and products.
5. In computer-based application, matrices play a crucial role in the creation of realistic seeming motion.

R Data Frame

A data frame is a two-dimensional array-like structure or a table in which a column contains values of one variable, and rows contains one set of values from each column. A data frame is a special case of the list in which each component has equal length.

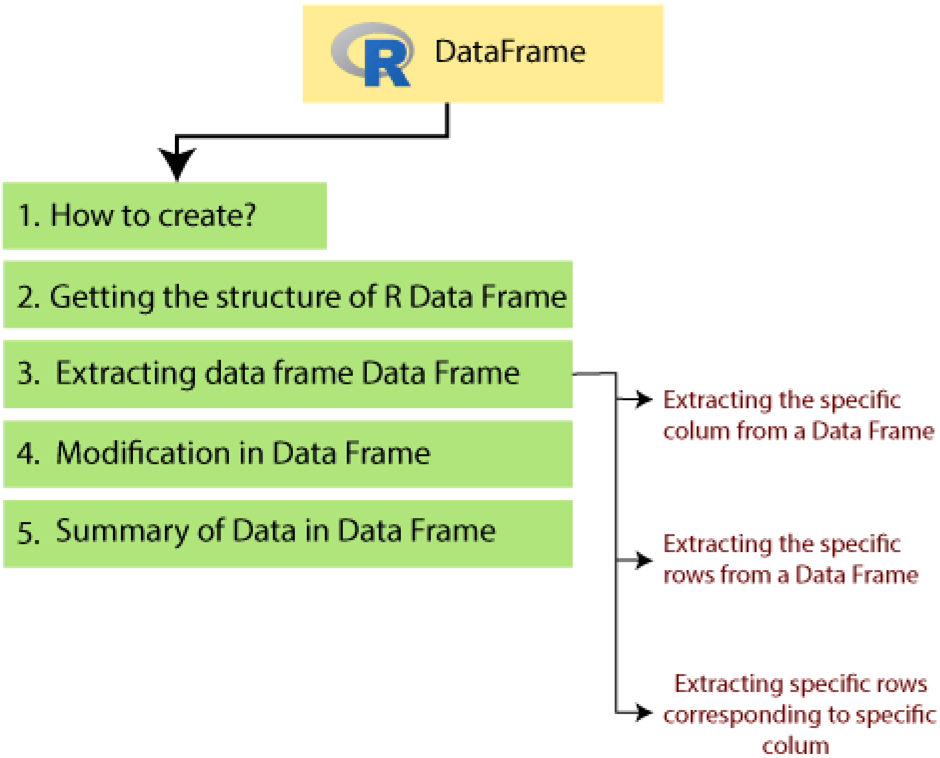
A data frame is used to store data table and the vectors which are present in the form of a list in a data frame, are of equal length.

In a simple way, it is a list of equal length vectors. A matrix can contain one type of data, but a data frame can contain different data types such as numeric, character, factor, etc.

There are following characteristics of a data frame.

- The columns name should be non-empty.

- The rows name should be unique.
- The data which is stored in a data frame can be a factor, numeric, or character type.
- Each column contains the same number of data items.



How to create Data Frame

In R, the data frames are created with the help of frame() function of data. This function contains the vectors of any type such as numeric, character, or integer. In below example, we create a data frame that contains employee id (integer vector), employee name(character vector), salary(numeric vector), and starting date(Date vector).

Example

1. # Creating the data frame.
2. emp.data<- data.frame(
3. employee_id = c (1:5),
4. employee_name = c("Shubham","Arpita","Nishka","Gunjan","Sumit"),
5. sal = c(623.3,915.2,611.0,729.0,843.25),

```
starting_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
stringsAsFactors = FALSE
)
# Printing the data frame.
print(emp.data)
```

Output

```
employee_idemployee_namesalstarting_date
1            1       Shubham623.30    2012-01-01
2            2        Arpita915.20    2013-09-23
3            3        Nishka611.00    2014-11-15
4            4        Gunjan729.00    2014-05-11
5           5         Sumit843.25    2015-03-27
```

Getting the structure of R Data Frame

In R, we can find the structure of our data frame. R provides an in-build function called str() which returns the data with its complete structure. In below example, we have created a frame using a vector of different data type and extracted the structure of it.

Example

```
# Creating the data frame.
emp.data<- data.frame(
employee_id = c (1:5),
employee_name = c("Shubham","Arpita","Nishka","Gunjan","Sumit"),
sal = c(623.3,515.2,611.0,729.0,843.25),
starting_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
stringsAsFactors = FALSE
)
# Printing the structure of data frame.
str(emp.data)
```

Output

```
'data.frame':	5 obs. of  4 variables:
 $ employee_id  : int  1 2 3 4 5
 $ employee_name: chr  "Shubham" "Arpita" "Nishka" "Gunjan" ...
 $ sal          : num  623 515 611 729 843
 $ starting_date: Date, format: "2012-01-01" "2013-09-23" ...
```

Extracting data from Data Frame

The data of the data frame is very crucial for us. To manipulate the data of the data frame, it is essential to extract it from the data frame. We can extract the data in three ways which are as follows:

1. We can extract the specific columns from a data frame using the column name.
2. We can extract the specific rows also from a data frame.
3. We can extract the specific rows corresponding to specific columns.

Let's see an example of each one to understand how data is extracted from the data frame with the help these ways.

Extracting the specific columns from a data frame

Example

```
# Creating the data frame.
emp.data<- data.frame(
employee_id = c (1:5),
employee_name= c("Shubham","Arpita","Nishka","Gunjan","Sumit"),
sal = c(623.3,515.2,611.0,729.0,843.25),
starting_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
stringsAsFactors = FALSE
)
# Extracting specific columns from a data frame
final <- data.frame(emp.data$employee_id,emp.data$sal)
print(final)
```

Output

```
emp.data.employee_idemp.data.sal
1                              1        623.30
2                              2        515.20
3                              3        611.00
4                              4        729.00
5                              5        843.25
```

Extracting the specific rows from a data frame

Example

```
# Creating the data frame.
emp.data<- data.frame(
employee_id = c (1:5),
employee_name = c("Shubham","Arpita","Nishka","Gunjan","Sumit"),
sal = c(623.3,515.2,611.0,729.0,843.25),
starting_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
   "2015-03-27")),
stringsAsFactors = FALSE
)
# Extracting first row from a data frame
final <- emp.data[1,]
print(final)
# Extracting last two row from a data frame
final <- emp.data[4:5,]
print(final)
```

Output

```
        employee_id  employee_name     sal        starting_date
1           1            Shubham       623.3         2012-01-01

       employee_id  employee_name     sal       starting_date
4          4           Gunjan        729.00       2014-05-11
5          5           Sumit         843.25       2015-03-27
```

Extracting specific rows corresponding to specific columns

Example

```
# Creating the data frame.
emp.data<- data.frame(
employee_id = c (1:5),
employee_name = c("Shubham","Arpita","Nishka","Gunjan","Sumit"),
sal = c(623.3,515.2,611.0,729.0,843.25),
starting_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
stringsAsFactors = FALSE
)
# Extracting 2nd and 3rd row corresponding to the 1st and 4th column
final <- emp.data[c(2,3),c(1,4)]
print(final)
```

Output

```
        employee_id   starting_date
2            2            2013-09-23
3            3            2014-11-15
```

Modification in Data Frame

R allows us to do modification in our data frame. Like matrices modification, we can modify our data frame through re-assignment. We cannot only add rows and columns, but also we can delete them. The data frame is expanded by adding rows and columns.

We can

1. Add a column by adding a column vector with the help of a new column name using cbind() function.
2. Add rows by adding new rows in the same structure as the existing data frame and using rbind() function
3. Delete the columns by assigning a NULL value to them.
4. Delete the rows by re-assignment to them.

Let's see an example to understand how rbind() function works and how the modification is done in our data frame.

Example: Adding rows and columns

```
# Creating the data frame.
emp.data<- data.frame(
employee_id = c (1:5),
employee_name = c("Shubham","Arpita","Nishka","Gunjan","Sumit"),
sal = c(623.3,515.2,611.0,729.0,843.25),
starting_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
stringsAsFactors = FALSE
)
print(emp.data)
#Adding row in the data frame
x <- list(6,"Vaishali",547,"2015-09-01")
rbind(emp.data,x)
#Adding column in the data frame
y <- c("Moradabad","Lucknow","Etah","Sambhal","Khurja")
cbind(emp.data,Address=y)
```

Output

```
     employee_id  employee_name    sal          starting_date
1        1             Shubham    623.30          2012-01-01
2        2             Arpita     515.20          2013-09-23
3        3             Nishka     611.00          2014-11-15
4        4             Gunjan     729.00          2014-05-11
5        5             Sumit      843.25          2015-03-27
     employee_id  employee_name     sal         starting_date
1        1             Shubham     623.30          2012-01-01
2        2             Arpita      515.20          2013-09-23
3        3             Nishka      611.00          2014-11-15
4        4             Gunjan      729.00          2014-05-11
5        5             Sumit       843.25          2015-03-27
6        6             Vaishali    547.00          2015-09-01
     employee_id     employee_name     sal        starting_date         Address
1        1             Shubham       623.30          2012-01-01         Moradabad
2        2             Arpita        515.20          2013-09-23         Lucknow
3        3             Nishka        611.00          2014-11-15         Etah
4        4             Gunjan        729.00          2014-05-11         Sambhal
5        5             Sumit         843.25          2015-03-27         Khurja
```

Example: Delete rows and columns

```
# Creating the data frame.
emp.data<- data.frame(
employee_id = c (1:5),
employee_name = c("Shubham","Arpita","Nishka","Gunjan","Sumit"),
sal = c(623.3,515.2,611.0,729.0,843.25),
starting_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
stringsAsFactors = FALSE
)
print(emp.data)
#Delete rows from data frame
emp.data<-emp.data[-1,]
print(emp.data)
 #Delete column from the data frame
emp.data$starting_date<-NULL
print(emp.data)
```

Output

```
employee_idemployee_namesalstarting_date
1           1        Shubham623.30    2012-01-01
2           2         Arpita515.20    2013-09-23
3           3         Nishka611.00    2014-11-15
4           4         Gunjan729.00    2014-05-11
5           5          Sumit843.25    2015-03-27
employee_idemployee_namesalstarting_date
2           2         Arpita515.20    2013-09-23
3           3         Nishka611.00    2014-11-15
4           4         Gunjan729.00    2014-05-11
5           5          Sumit843.25    2015-03-27
employee_idemployee_namesal
1           1        Shubham623.30
2           2         Arpita515.20
3         3        Nishka611.00
4           4         Gunjan729.00
5           5          Sumit843.25
```

Summary of data in Data Frames

In some cases, it is required to find the statistical summary and nature of the data in the data frame. R provides the summary() function to extract the statistical summary and nature of the data. This function takes the data frame as a parameter and returns the statistical information of the data. Let?s see an example to understand how this function is used in R:

Example

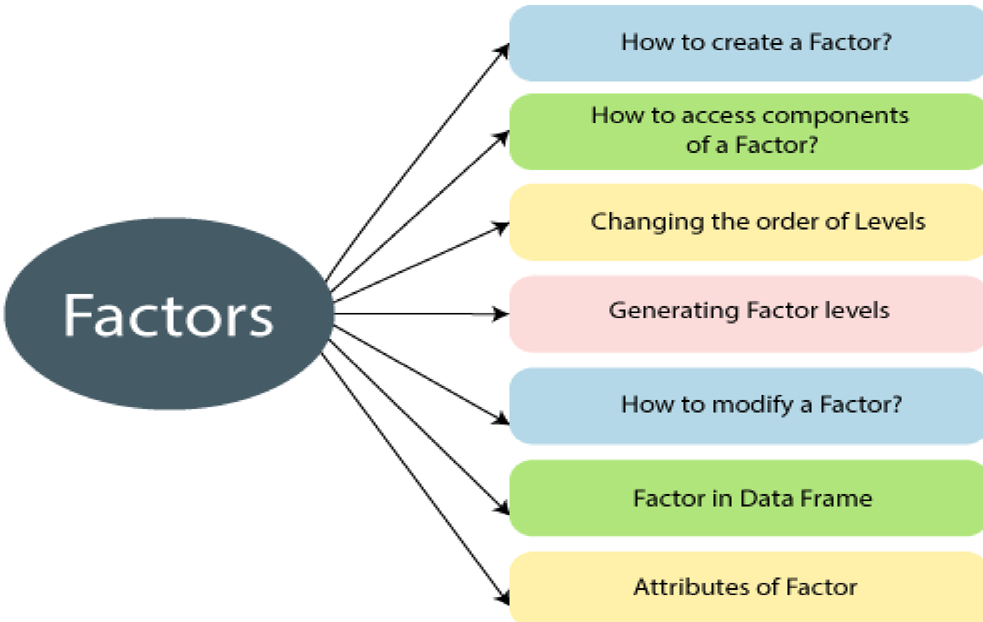
```
# Creating the data frame.
emp.data<- data.frame(
employee_id = c (1:5),
employee_name = c("Shubham","Arpita","Nishka","Gunjan","Sumit"),
sal = c(623.3,515.2,611.0,729.0,843.25),
starting_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
    "2015-03-27")),
stringsAsFactors = FALSE
)
print(emp.data)
 #Printing the summary
print(summary(emp.data))
```

Output

```
employee_idemployee_namesalstarting_date
1            1       Shubham623.30    2012-01-01
2            2        Arpita515.20    2013-09-23
3            3  Nishka611.00    2014-11-15
4            4        Gunjan729.00    2014-05-11
5            5         Sumit843.25    2015-03-27

employee_idemployee_namesalstarting_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
```

R factors

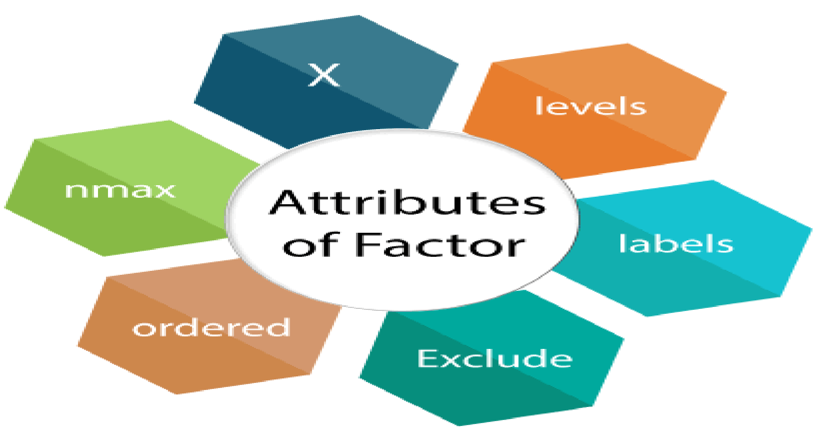
The factor is a data structure which is used for fields which take only predefined finite number of values. These are the variable which takes a limited number of different values. These are the data objects which are used to categorize the data and to store it on multiple levels. It can store both integers and strings values, and are useful in the column that has a limited number of unique values.



Factors have labels which are associated with the unique integers stored in it. It contains predefined set value known as levels and by default R always sorts levels in alphabetical order.

Attributes of a factor

There are the following attributes of a factor in R



a. **X**

It is the input vector which is to be transformed into a factor.

b. **levels**

   It is an input vector that represents a set of unique values which are taken by x.

c. **labels**

   It is a character vector which corresponds to the number of labels.

d. **Exclude**

   It is used to specify the value which we want to be excluded,

e. **ordered**

   It is a logical attribute which determines if the levels are ordered.

f. **nmax**

   It is used to specify the upper bound for the maximum number of level.

How to create a factor?

In R, it is quite simple to create a factor. A factor is created in two steps

1. In the first step, we create a vector.
2. Next step is to convert the vector into a factor,

R provides factor() function to convert the vector into factor. There is the following syntax of factor() function

1. factor_data<- factor(vector)

Let's see an example to understand how factor function is used.

Example

1. # Creating a vector as input.
2. data <- c("Shubham","Nishka","Arpita","Nishka","Shubham","Sumit","Nishka","Shubham","Sumit","Arpita","Sumit")
3. print(data)
4. print(is.factor(data))
5.

6. # Applying the factor function.
7. factor_data<- factor(data)
8. print(factor_data)
9. print(is.factor(factor_data))

Output

```
[1] "Shubham" "Nishka"  "Arpita"  "Nishka"  "Shubham" "Sumit"   "Nishka"
 [8] "Shubham" "Sumit"   "Arpita"  "Sumit"
[1] FALSE
 [1] Shubham Nishka Arpita Nishka Shubham Sumit Nishka Shubham Sumit
[10] Arpita Sumit
Levels: Arpita Nishka Shubham Sumit
[1] TRUE
```

Accessing components of factor

Like vectors, we can access the components of factors. The process of accessing components of factor is much more similar to the vectors. We can access the element with the help of the indexing method or using logical vectors. Let's see an example in which we understand the different-different ways of accessing the components.

Example

1. # Creating a vector as input.
2. data <- c("Shubham","Nishka","Arpita","Nishka","Shubham","Sumit","Nishka","Shubham","Sumit","Arpita","Sumit")
3. # Applying the factor function.
4. factor_data<- factor(data)
5. #Printing all elements of factor
6. print(factor_data)
7. #Accessing 4th element of factor
8. print(factor_data[4])
9. #Accessing 5th and 7th element
10. print(factor_data[c(5,7)])
11. #Accessing all elemcent except 4th one
12. print(factor_data[-4])
13. #Accessing elements using logical vector

14. print(factor_data[c(TRUE,FALSE,FALSE,FALSE,TRUE,TRUE,TRUE,FALSE,FALSE,FALSE,TRUE)])

Output

```
[1] Shubham Nishka Arpita Nishka Shubham Sumit Nishka Shubham Sumit
[10] Arpita Sumit
Levels: Arpita Nishka Shubham Sumit

[1] Nishka
Levels: Arpita Nishka Shubham Sumit

[1] Shubham Nishka
Levels: Arpita Nishka Shubham Sumit

 [1] Shubham Nishka Arpita Shubham Sumit Nishka Shubham Sumit Arpita
[10] Sumit
Levels: Arpita Nishka Shubham Sumit

[1] Shubham Shubham Sumit Nishka Sumit
Levels: Arpita Nishka Shubham Sumit
```

Modification of factor

Like data frames, R allows us to modify the factor. We can modify the value of a factor by simply re-assigning it. In R, we cannot choose values outside of its predefined levels means we cannot insert value if it's level is not present on it. For this purpose, we have to create a level of that value, and then we can add it to our factor.

Let's see an example to understand how the modification is done in factors.

Example

1. # Creating a vector as input.
2. data <- c("Shubham","Nishka","Arpita","Nishka","Shubham")
3. # Applying the factor function.
4. factor_data<- factor(data)
5. #Printing all elements of factor
6. print(factor_data)
7. #Change 4th element of factor with sumit
8. factor_data[4] <-"Arpita"
9. print(factor_data)
10.

11. #change 4th element of factor with "Gunjan"
12. factor_data[4] <- "Gunjan"   # cannot assign values outside levels
13. print(factor_data)
14. #Adding the value to the level
15. levels(factor_data) <- c(levels(factor_data),"Gunjan")#Adding new level
16. factor_data[4] <- "Gunjan"
17. print(factor_data)

Output

```
[1] Shubham Nishka Arpita Nishka Shubham
Levels: Arpita Nishka Shubham
[1] Shubham Nishka Arpita Arpita Shubham
Levels: Arpita Nishka Shubham
Warning message:
In `[<-.factor`(`*tmp*`, 4, value = "Gunjan") :
  invalid factor level, NA generated
[1] Shubham Nishka Arpita  Shubham
Levels: Arpita Nishka Shubham
[1] Shubham Nishka Arpita Gunjan Shubham
Levels: Arpita Nishka Shubham Gunjan
```

Factor in Data Frame

When we create a frame with a column of text data, R treats this text column as categorical data and creates factor on it.

Example

1. # Creating the vectors for data frame.
2. height <- c(132,162,152,166,139,147,122)
3. weight <- c(40,49,48,40,67,52,53)
4. gender <- c("male","male","female","female","male","female","male")
5. # Creating the data frame.
6. input_data<- data.frame(height,weight,gender)
7. print(input_data)
8. # Testing if the gender column is a factor.
9. print(is.factor(input_data$gender))
10. # Printing the gender column to see the levels.
11. print(input_data$gender)

Output

```
height weight gender
1    132     40   male
2    162     49   male
3    152     48 female
4    166     40 female
5    139     67   male
6    147     52 female
7    122     53   male
[1] TRUE
[1] male   male   female female male   female male
Levels: female male
```

Changing order of the levels

In R, we can change the order of the levels in the factor with the help of the factor function.

Example

1. data <- c("Nishka","Gunjan","Shubham","Arpita","Arpita","Sumit","Gunjan","Shubham")
2. # Creating the factors
3. factor_data<- factor(data)
4. print(factor_data)
5. # Apply the factor function with the required order of the level.
6. new_order_factor<- factor(factor_data,levels = c("Gunjan","Nishka","Arpita","Shubham","Sumit"))
7. print(new_order_factor)

Output

```
[1] Nishka Gunjan Shubham Arpita Arpita Sumit Gunjan Shubham
Levels: Arpita Gunjan Nishka Shubham Sumit
[1] Nishka Gunjan Shubham Arpita Arpita Sumit Gunjan Shubham
Levels: Gunjan Nishka Arpita Shubham Sumit
```

Generating Factor Levels

R provides gl() function to generate factor levels. This function takes three arguments i.e., n, k, and labels. Here, n and k are the integers which indicate how many levels we want and how many times each level is required.

There is the following syntax of gl() function which is as follows

1. gl(n, k, labels)

    1. n indicates the number of levels.
    2. k indicates the number of replications.
    3. labels is a vector of labels for the resulting factor levels.

Example

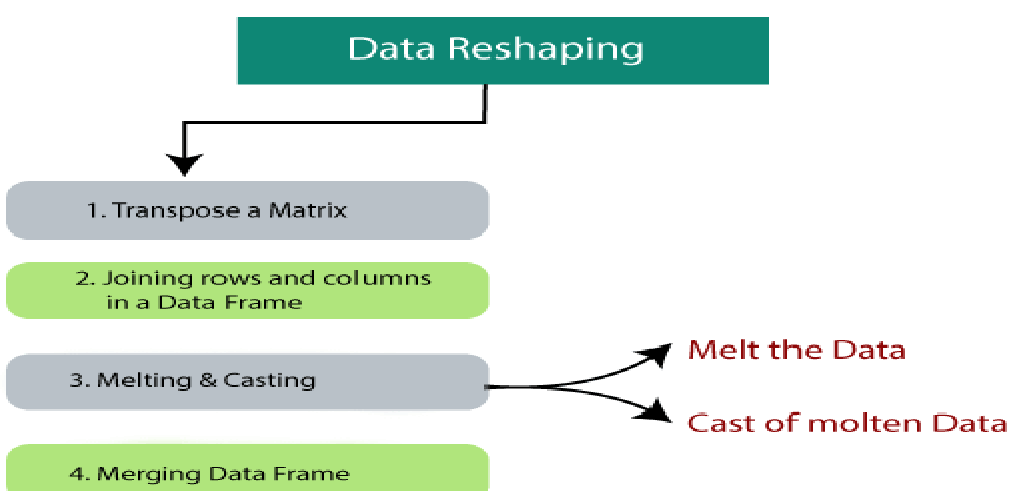
```
1. gen_factor<- gl(3,5,labels=c("BCA","MCA","B.Tech"))
2. gen_factor
```

Output

```
[1] BCA BCA BCA BCA BCA MCA MCA MCA MCA MCA
[11] B.Tech B.Tech B.Tech B.Tech B.Tech
Levels: BCA MCA B.Tech
```

Data Reshaping in R

In R, Data Reshaping is about changing how the data is organized into rows and columns. In R, data processing is done by taking the input as a data frame. It is much easier to extract data from the rows and columns of a data frame, but there is a problem when we need a data frame in a format which is different from the format in which we received it. R provides many functions to merge, split, and change the rows to columns and vice-versa in a data frame.



Transpose a Matrix

R allows us to calculate the transpose of a matrix or a data frame by providing t() function. This t() function takes the matrix or data frame as an input and return the transpose of the input matrix or data frame. The syntax of t() function is as follows:

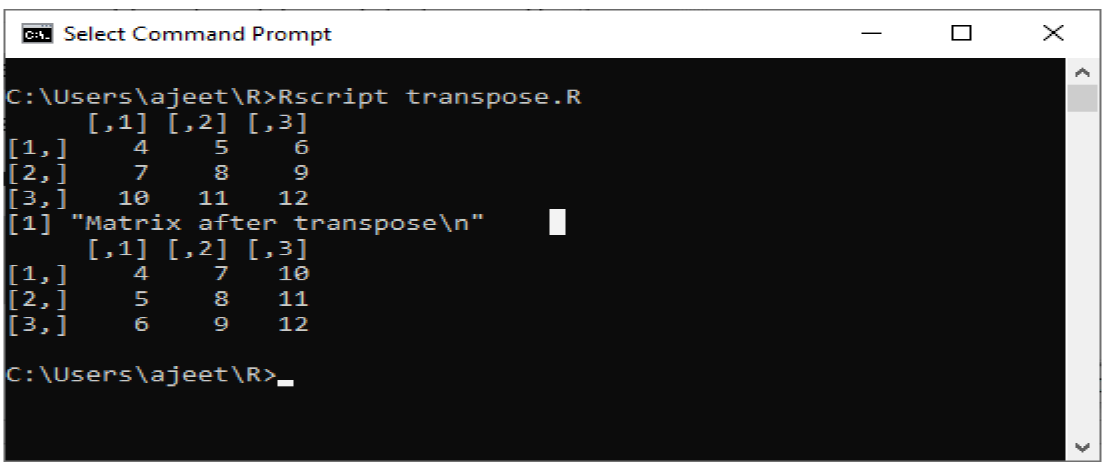
1. t(Matrix/data frame)

Let's see an example to understand how this function is used

Example

1. a <- matrix(c(4:12),nrow=3,byrow=TRUE)
2. a
3. print("Matrix after transpose\n")
4. b <- t(a)
5. b

Output:



Joining rows and columns in Data Frame

R allows us to join multiple vectors to create a data frame. For this purpose R provides cbind() function. R also provides rbind() function, which allows us to merge two data frame. In some situation, we need to merge data frames to access the information which depends on both the data frame. There is the following syntax of cbind() function and rbind() function.

```
cbind(vector1, vector2,.......vectorN)
rbind(dataframe1, dataframe2,........dataframeN)
```

Let's see an example to understand how cbind() and rbind() function is used.

Example

```
#Creating vector objects
Name <- c("Shubham Rastogi","Nishka Jain","Gunjan Garg","Sumit Chaudhary")
Address <- c("Moradabad","Etah","Sambhal","Khurja")
Marks <- c(255,355,455,655)
#Combining vectors into one data frame
info <- cbind(Name,Address,Marks)
#Printing data frame
print(info)
# Creating another data frame with similar columns
new.stuinfo <- data.frame(
   Name = c("Deepmala","Arun"),
   Address = c("Khurja","Moradabad"),
   Marks = c("755","855"),
   stringsAsFactors=FALSE
)
#Printing a header.
cat("# # # The Second data frame\n")
#Printing the data frame.
print(new.stuinfo)
# Combining rows form both the data frames.
all.info <- rbind(info,new.stuinfo)
# Printing a header.
cat("# # # The combined data frame\n")
# Printing the result.
print(all.info)
```

Output:

```
C:\Users\ajeet\R>Rscript transpose.R
     Name              Address     Marks
[1,] "Shubham Rastogi" "Moradabad" "255"
[2,] "Nishka Jain"     "Etah"      "355"
[3,] "Gunjan Garg"     "Sambhal"   "455"
[4,] "Sumit Chaudhary" "Khurja"    "655"
# # # The Second data frame
      Name   Address Marks
1 Deepmala    Khurja   755
2     Arun Moradabad   855
# # # The combined data frame
             Name   Address Marks
1 Shubham Rastogi Moradabad   255
2     Nishka Jain      Etah   355
3     Gunjan Garg   Sambhal   455
4 Sumit Chaudhary    Khurja   655
5        Deepmala    Khurja   755
6            Arun Moradabad   855

C:\Users\ajeet\R>
```

Merging Data Frame

R provides the merge() function to merge two data frames. In the merging process, there is a constraint i.e.; data frames must have the same column names.

Let's take an example in which we take the dataset about Diabetes in Pima Indian Women which is present in the "MASS" library. We will merge two datasets on the basis of the value of the blood pressure and body mass index. When selecting these two columns for merging, the records where values of these two variables match in both data sets are combined together to form a single data frame.

Example

```
library(MASS)
merging_pima<- merge(x = Pima.te, y = Pima.tr,
  by.x = c("bp", "bmi"),
  by.y = c("bp", "bmi")
)
```

6. print(merging_pima)
7. nrow(merging_pima)

Output:

```
Command Prompt

C:\Users\ajeet\R>Rscript melting.R
   bp  bmi npreg.x glu.x skin.x ped.x age.x type.x npreg.y glu.y skin.y ped.y
1  60 33.8       1   117     23 0.466    27     No       2   125     20 0.088
2  64 29.7       2    75     24 0.370    33     No       2   100     23 0.368
3  64 31.2       5   189     33 0.583    29    Yes       3   158     13 0.295
4  64 33.2       4   117     27 0.230    24     No       1    96     27 0.289
5  66 38.1       3   115     39 0.150    28     No       1   114     36 0.289
6  68 38.5       2   100     25 0.324    26     No       7   129     49 0.439
7  70 27.4       1   116     28 0.204    21     No       0   124     20 0.254
8  70 33.1       4    91     32 0.446    22     No       9   123     44 0.374
9  70 35.4       9   124     33 0.282    34     No       6   134     23 0.542
10 72 25.6       1   157     21 0.123    24     No       4    99     17 0.294
11 72 37.7       5    95     33 0.370    27     No       6   103     32 0.324
12 74 25.9       9   134     33 0.460    81     No       8   126     38 0.162
13 74 25.9       1    95     21 0.673    36     No       8   126     38 0.162
14 78 27.6       5    88     30 0.258    37     No       6   125     31 0.565
15 78 27.6      10   122     31 0.512    45     No       6   125     31 0.565
16 78 39.4       2   112     50 0.175    24     No       4   112     40 0.236
17 88 34.5       1   117     24 0.403    40    Yes       4   127     11 0.598
   age.y type.y
1     31     No
2     21     No
3     24     No
4     21     No
5     21     No
6     43    Yes
7     36    Yes
8     40     No
9     29    Yes
10    28     No
11    55     No
12    39     No
13    39     No
14    49    Yes
15    49    Yes
16    38     No
17    28     No
[1] 17

C:\Users\ajeet\R>
```

Melting and Casting

In R, the most important and interesting topic is about changing the shape of the data in multiple steps to get the desired shape. For this purpose, R provides melt() and cast() function. To understand its process, consider a dataset called ships which is present in the MASS library.

Example

1. library(MASS)
2. print(ships)

Output:

```
Command Prompt

C:\Users\ajeet\R>Rscript melting.R
   type year period service incidents
1     A   60     60     127         0
2     A   60     75      63         0
3     A   65     60    1095         3
4     A   65     75    1095         4
5     A   70     60    1512         6
6     A   70     75    3353        18
7     A   75     60       0         0
8     A   75     75    2244        11
9     B   60     60   44882        39
10    B   60     75   17176        29
11    B   65     60   28609        58
12    B   65     75   20370        53
13    B   70     60    7064        12
14    B   70     75   13099        44
15    B   75     60       0         0
16    B   75     75    7117        18
17    C   60     60    1179         1
18    C   60     75     552         1
19    C   65     60     781         0
20    C   65     75     676         1
21    C   70     60     783         6
22    C   70     75    1948         2
23    C   75     60       0         0
24    C   75     75     274         1
25    D   60     60     251         0
26    D   60     75     105         0
27    D   65     60     288         0
28    D   65     75     192         0
29    D   70     60     349         2
30    D   70     75    1208        11
31    D   75     60       0         0
```

Melt the Data

Now we will use the above data to organize it by melting it. Melting means the conversion of columns into multiple rows. We will convert all the columns except type and year of the above dataset into multiple rows.

Example

```
1. library(MASS)
2. library(reshape2)
3. molten_ships <- melt(ships, id = c("type","year"))
4. print(molten_ships)
```

Output:

Command Prompt

```
C:\Users\ajeet\R>Rscript melting.R
   type year  variable value
1     A   60    period    60
2     A   60    period    75
3     A   65    period    60
4     A   65    period    75
5     A   70    period    60
6     A   70    period    75
7     A   75    period    60
8     A   75    period    75
9     B   60    period    60
10    B   60    period    75
11    B   65    period    60
12    B   65    period    75
13    B   70    period    60
14    B   70    period    75
15    B   75    period    60
16    B   75    period    75
17    C   60    period    60
18    C   60    period    75
19    C   65    period    60
20    C   65    period    75
21    C   70    period    60
22    C   70    period    75
23    C   75    period    60
24    C   75    period    75
25    D   60    period    60
26    D   60    period    75
27    D   65    period    60
28    D   65    period    75
29    D   70    period    60
30    D   70    period    75
31    D   75    period    60
32    D   75    period    75
33    E   60    period    60
34    E   60    period    75
35    E   65    period    60
36    E   65    period    75
37    E   70    period    60
38    E   70    period    75
39    E   75    period    60
40    E   75    period    75
41    A   60   service   127
```

Casting of Molten Data

After melting the data, we can cast it into a new form where the aggregate of each type of ship for each year is created. For this purpose, R provides cast() function.

Let's starts doing the casting of our molten data.

Example

```
1. library(MASS)
2. library(reshape2)
3. #Melting the data
4. molten.ships <- melt(ships, id = c("type","year"))
5. print("Molted Data")
6. print(molten.ships)
7. #Casting of data
8. recasted.ship <- dcast(molten.ships, type+year~variable,sum)
9. print("Cast Data")
10. print(recasted.ship)
```

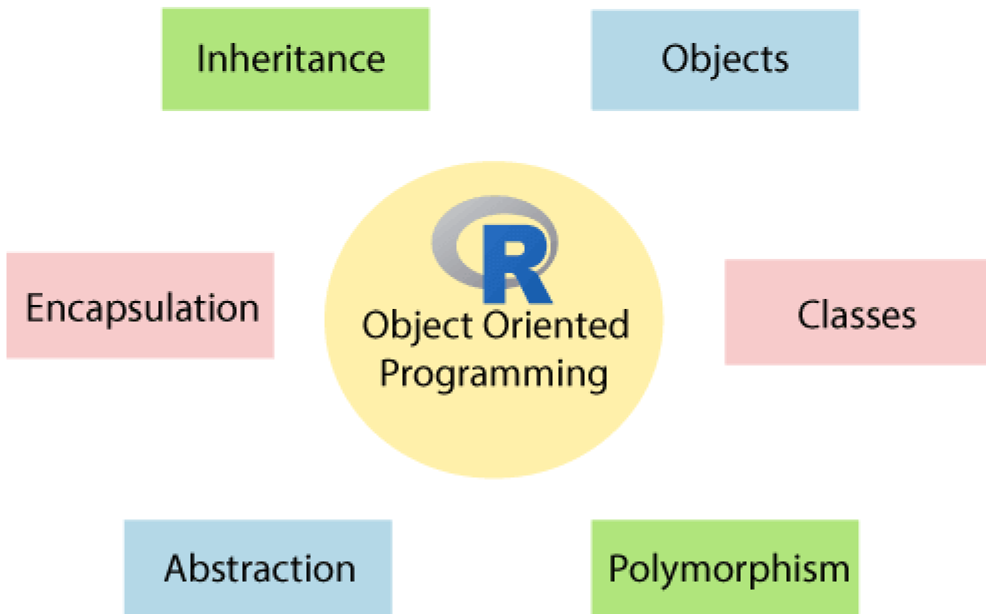
Output:

Command Prompt

```
101    C   70 incidents     6
102    C   70 incidents     2
103    C   75 incidents     0
104    C   75 incidents     1
105    D   60 incidents     0
106    D   60 incidents     0
107    D   65 incidents     0
108    D   65 incidents     0
109    D   70 incidents     2
110    D   70 incidents    11
111    D   75 incidents     0
112    D   75 incidents     4
113    E   60 incidents     0
114    E   60 incidents     0
115    E   65 incidents     7
116    E   65 incidents     7
117    E   70 incidents     5
118    E   70 incidents    12
119    E   75 incidents     0
120    E   75 incidents     1
[1] "Cast Data"
   type year period service incidents
1     A   60    135     190         0
2     A   65    135    2190         7
3     A   70    135    4865        24
4     A   75    135    2244        11
5     B   60    135   62058        68
6     B   65    135   48979       111
7     B   70    135   20163        56
8     B   75    135    7117        18
9     C   60    135    1731         2
10    C   65    135    1457         1
11    C   70    135    2731         8
12    C   75    135     274         1
13    D   60    135     356         0
14    D   65    135     480         0
15    D   70    135    1557        13
16    D   75    135    2051         4
17    E   60    135      45         0
18    E   65    135    1226        14
19    E   70    135    3318        17
20    E   75    135     542         1
```

What is Object-Oriented Programming in R?

Object-Oriented Programming (OOP) is the most popular programming language. With the help of oops concepts, we can construct the modular pieces of code which are used to build blocks for large systems. R is a functional language, and we can do programming in oops style. In R, oops is a great tool to manage the complexity of larger programs.



In Object-Oriented Programming, S3 and S4 are the two important systems.

S3

In oops, the S3 is used to overload any function. So that we can call the functions with different names and it depends on the type of input parameter or the number of parameters.

S4

S4 is the most important characteristic of oops. However, this is a limitation, as it is quite difficult to debug. There is an optional reference class for S4.

Objects and Classes in R

In R, everything is an object. Therefore, programmers perform OOPS concept when they write code in R. An object is a data structure which has some methods that can act upon its attributes.

In R, classes are the outline or design for the object. Classes encapsulate the data members, along with the functions. In R, there are two most important classes, i.e., S3 and S4, which play an important role in performing OOPs concepts.

Let's discuss both the classes one by one with their examples for better understanding.

1) S3 Class

With the help of the S3 class, we can take advantage of the ability to implement the generic function OO. Furthermore, using only the first argument, S3 is capable of dispatching. S3 differs from traditional programming languages such as Java, C ++, and C #, which implement OO passing messages. This makes S3 easy to implement. In the S3 class, the generic function calls the method. S3 is very casual and has no formal definition of classes.

S3 requires very little knowledge from the programmer.

Creating an S3 class

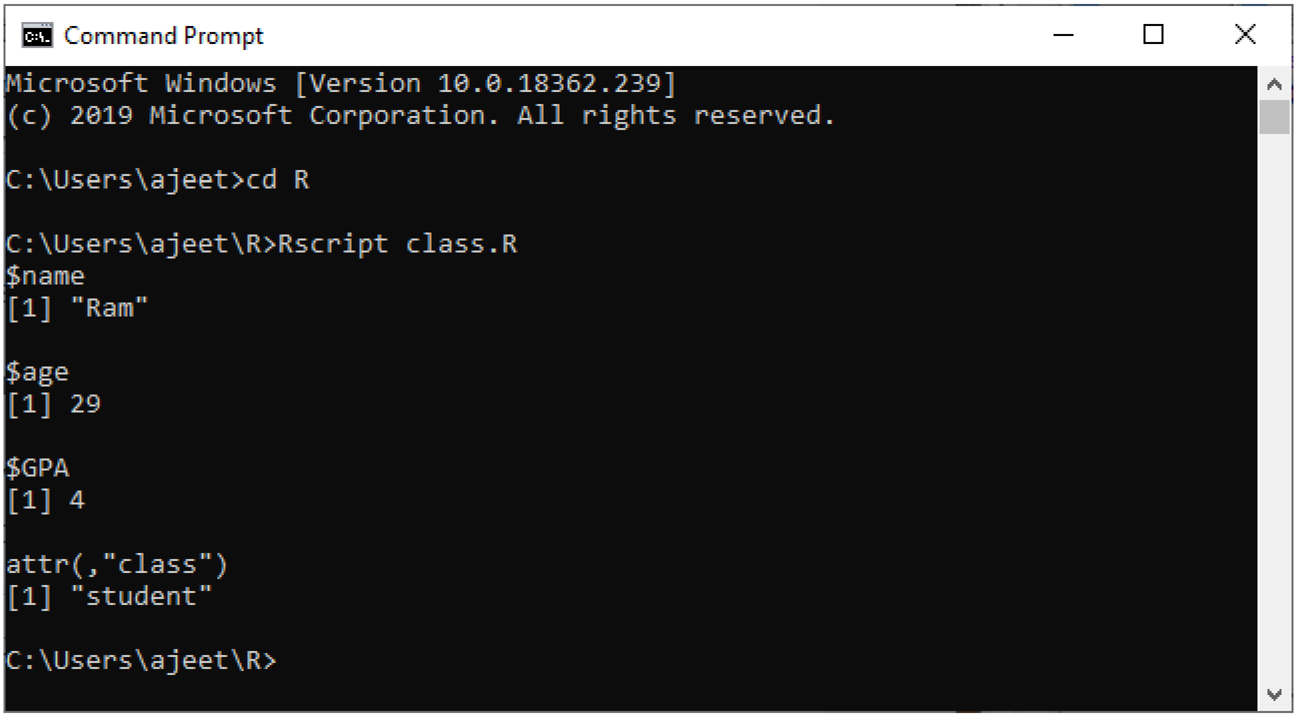
In R, we define a function which will create a class and return the object of the created class. A list is made with relevant members, class of the list is determined, and a copy of the list is returned. There is the following syntax to create a class

1. variable_name <- list(member1, member2, member3.........memberN)

Example

1. s <- list(name = "Ram", age = 29, GPA = 4.0)
2. class(s) <- "Faculty"
3. s

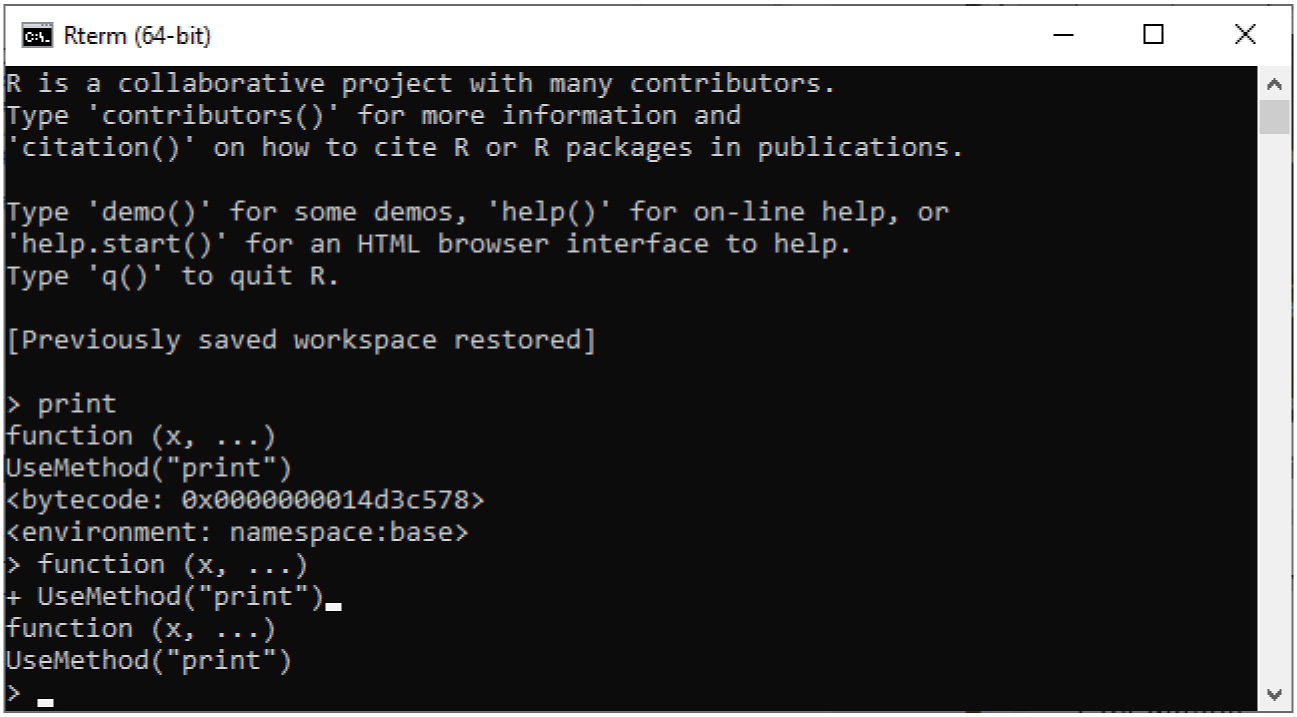
Output



There is the following way in which we define our generic function print.

1. print
2. function(x, ...)
3. UseMethod("Print")

When we execute or run the above code, it will give us the following output:



Like print function, we will make a generic function GPA to assign a new value to our GPA member. In the following way we will make the generic function GPA

```
GPA <- function(obj1){
UseMethod("GPA")
}
```

Once our generic function GPA is created, we will implement a default function for it

```
GPA.default <- function(obj){
cat("We are entering in generic function\n")
}
```

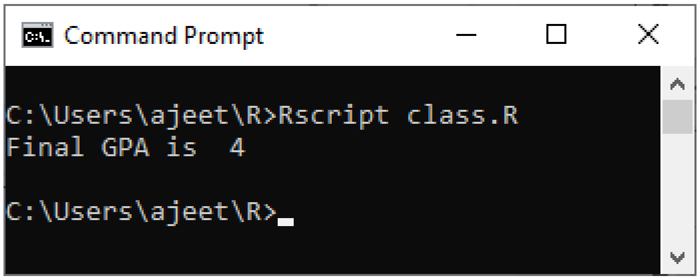
After that we will make a new method for our GPA function in the following way

```
GPA.faculty <- function(obj1){
cat("Final GPA is ",obj1$GPA,"\n")
}
```

And at last we will run the method GPA as

```
GPA(s)
```

Output



Inheritance in S3

Inheritance means extracting the features of one class into another class. In the S3 class of R, inheritance is achieved by applying the class attribute in a vector.

For inheritance, we first create a function which creates new object of class faculty in the following way

```
faculty<- function(n,a,g) {
value <- list(nname=n, aage=a, GPA=g)
attr(value, "class") <- "faculty"
value
}
```

After that we will define a method for generic function print() as

```
 print.student <- function(obj1) {
cat(1obj$name, "\n")
cat(1obj$age, "years old\n")
cat("GPA:", obj1$GPA, "\n")
}
```

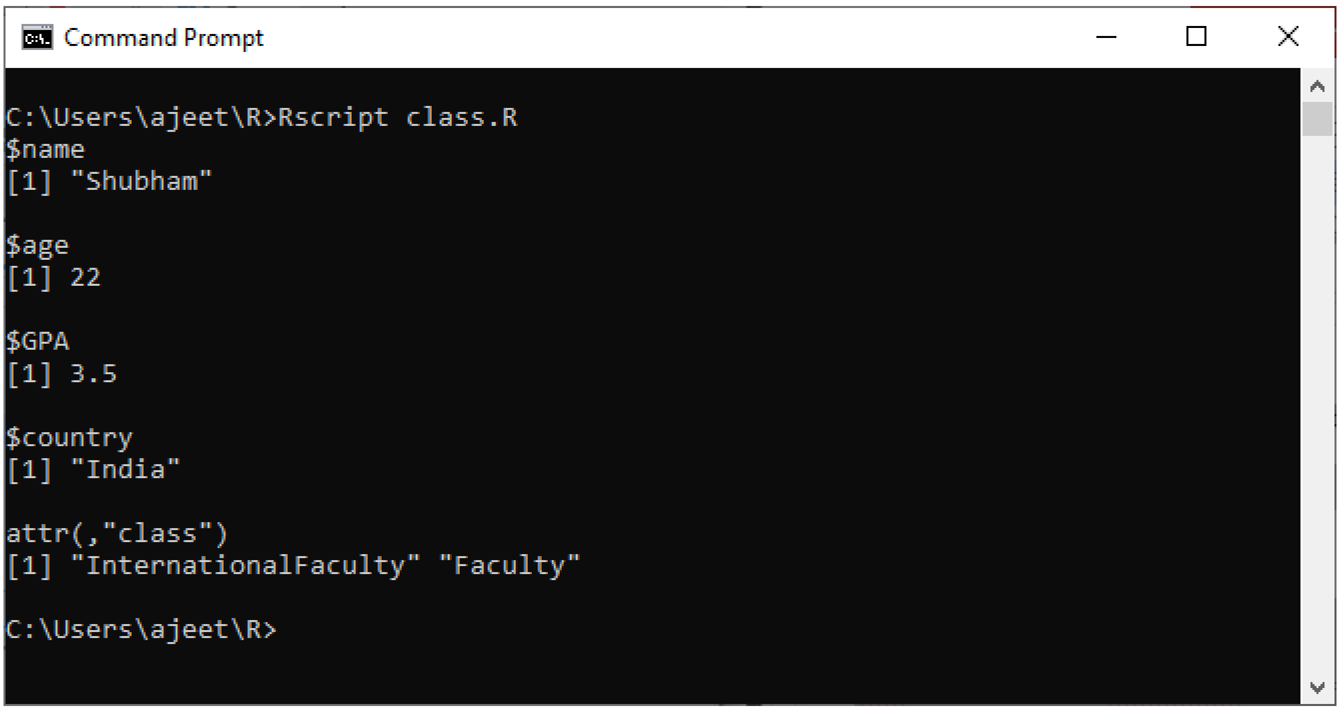
Now, we will create an object of class InternationalFaculty which will inherit from faculty class. This process will be done by assigning a character vector of class name as:

```
class(Objet) <- c(child, parent)
```

so,

1. # create a list
2. fac <- list(name="Shubham", age=22, GPA=3.5, country="India")
3. # make it of the class InternationalFaculty which is derived from the class Faculty
4. class(fac) <- c("InternationalFaculty","Faculty")
5. # print it out
6. fac

When we run the above code which we have discussed, it will generate the following output:



We can see above that, we have not defined any method of form print. InternationalFaculty (), the method called print.Faculty(). This method of class Faculty was inherited.

So our next step is to defined print.InternationalFaculty() in the following way:

1. print.InternationalFaculty<- function(obj1) {
2. cat(obj1$name, "is from", obj1$country, "\n")
3. }

The above function will overwrite the method defined for class faculty as

1. Fac

```
C:\Users\ajeet\R>Rscript class.R
$name
[1] "Shubham"

$age
[1] 22

$GPA
[1] 3.5

$country
[1] "India"

attr(,"class")
[1] "InternationalFaculty" "Faculty"
Shubham is from India

C:\Users\ajeet\R>
```

getS3method and getAnywhere function

There are the two most common and popular S3 method functions which are used in R. The first method is **getS3method()** and the second one is **getAnywhere()**.

S3 finds the appropriate method associated with a class, and it is useful to see how a method is implemented. Sometimes, the methods are non-visible, because they are hidden in a namespace. We use getS3method or getAnywhere to solve this problem.

getS3method

Rterm (64-bit)

```
> exists("predict.ppr")
[1] FALSE
> getS3method("predict","ppr")
function (object, newdata, ...)
{
    if (missing(newdata))
        return(fitted(object))
    if (!is.null(object$terms)) {
        newdata <- as.data.frame(newdata)
        rn <- row.names(newdata)
        Terms <- delete.response(object$terms)
        m <- model.frame(Terms, newdata, na.action = na.omit,
            xlev = object$xlevels)
        if (!is.null(cl <- attr(Terms, "dataClasses")))
            .checkMFClasses(cl, m)
        keep <- match(row.names(m), rn)
        x <- model.matrix(Terms, m, contrasts.arg = object$contrasts)
    }
    else {
        x <- as.matrix(newdata)
        keep <- seq_len(nrow(x))
        rn <- dimnames(x)[[1L]]
    }
    if (ncol(x) != object$p)
        stop("wrong number of columns in 'x'")
    res <- matrix(NA, length(keep), object$q, dimnames = list(rn,
        object$ynames))
    res[keep, ] <- matrix(.Fortran(C_pppred, as.integer(nrow(x)),
        as.double(x), as.double(object$smod), y = double(nrow(x) *
            object$q), double(2 * object$smod[4L]))$y, ncol = object$q)
    drop(res)
}
<bytecode: 0x0000000004ae3608>
<environment: namespace:stats>
>
```

getAnywhere function

1. getAnywhere("simpleloess")

2) S4 Class

The S4 class is similar to the S3 but is more formal than the latter one. It differs from S3 in two different ways. First, in S4, there are formal class definitions which provide a description and representation of classes. In addition, it has special auxiliary functions for defining methods and generics. The S4 also offers multiple dispatches. This means that common functions are capable of taking methods based on multiple arguments which are based on class.

Creating an S4 class

In R, we use setClass() command for creating S4 class. In S4 class, we will specify a function for verifying the data consistency and also specify the default value. In R, member variables are called slots.

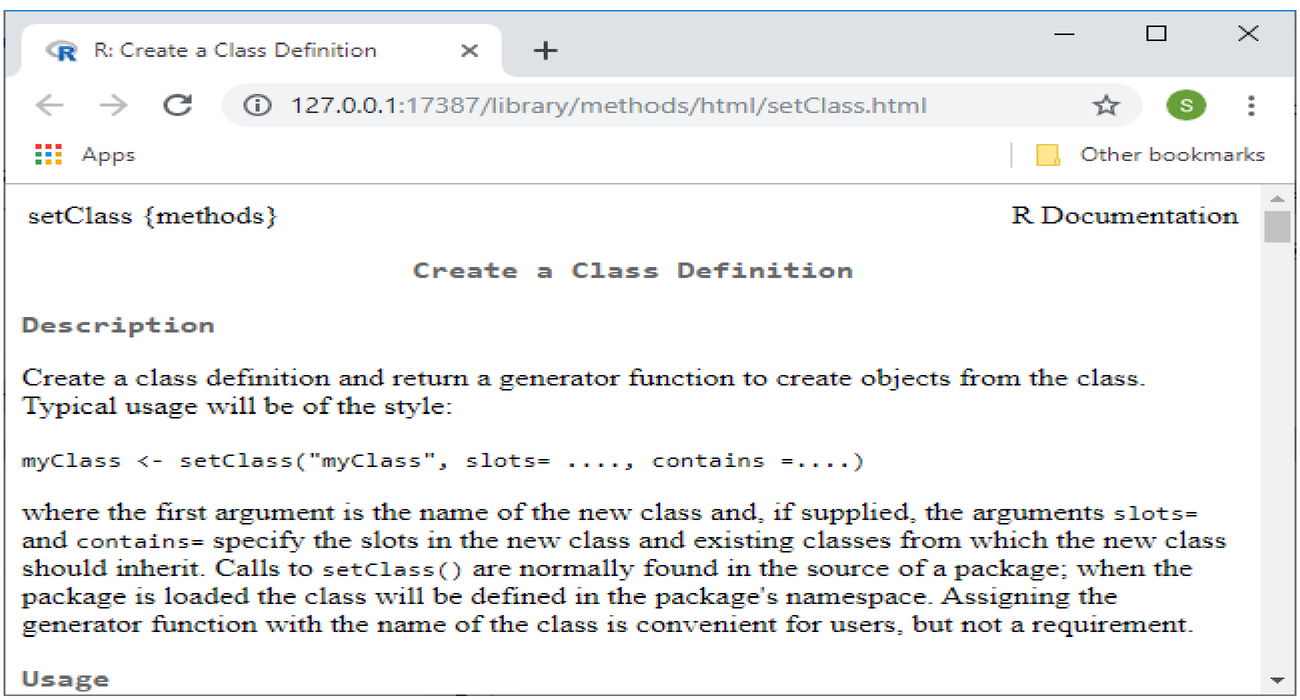
To create an S3 class, we have to define the class and its slots. There are the following steps to create an S4 class

Step 1:

In the first step, we will create a new class called faculty with three slots name, age, and GPA.

```
1. setClass("faculty", slots=list(name="character", age="numeric", GPA="numeric"))
```

There are many other optional arguments of setClass() function which we can explore by using **?setClass** command.

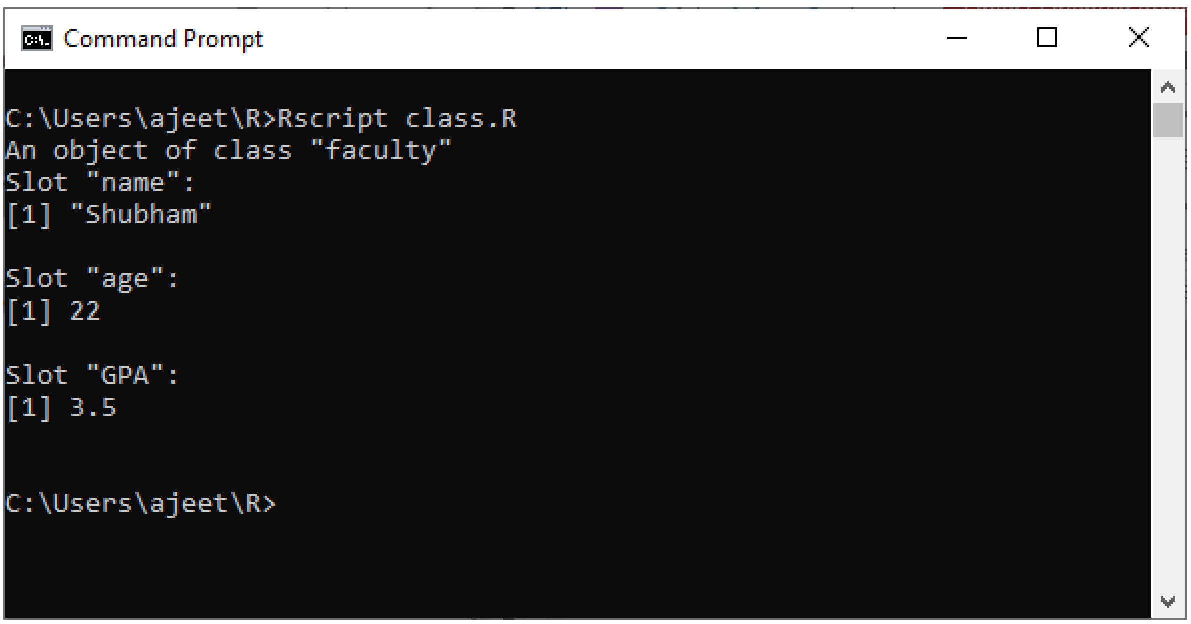


Step 2:

In the next step, we will create the object of S4 class. R provides new() function to create an object of S4 class. In this new function we pass the class name and the values for the slots in the following way:

1. setClass("faculty", slots=list(name="character", age="numeric", GPA="numeric"))
2. # creating an object using new()
3. # providing the class name and value for slots
4. s <- new("faculty",name="Shubham", age=22, GPA=3.5)
5. s

It will generate the following output

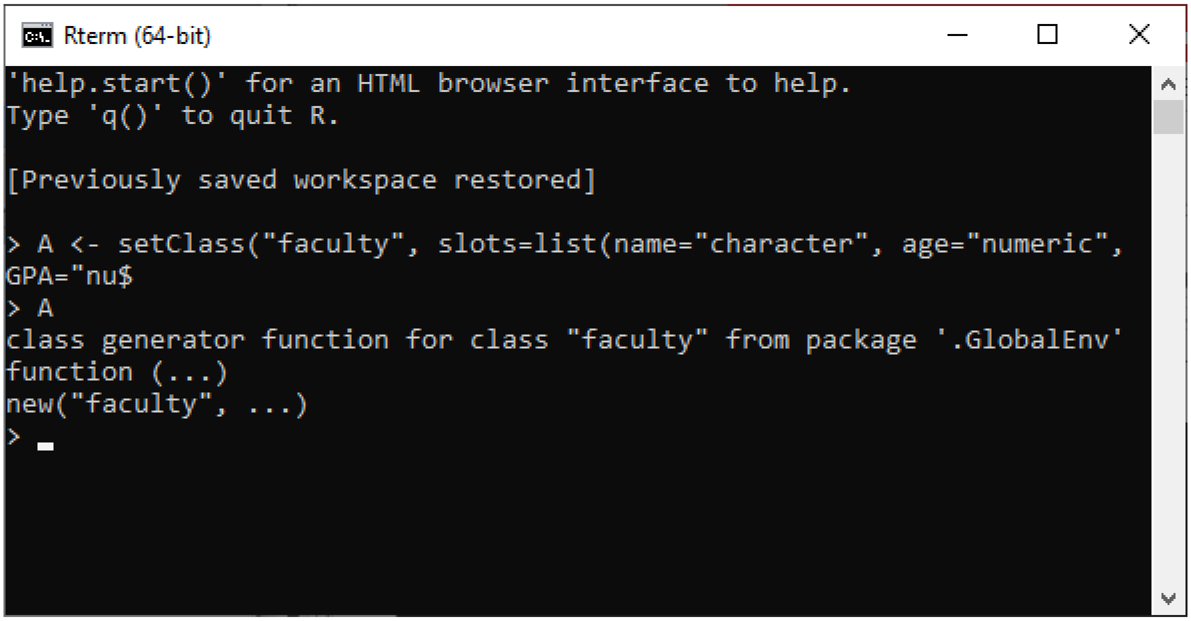


Creating S4 objects using a generator function

The setClass() function returns a generator function. This generator function helps in creating new objects. And it acts as a constructor.

1. A <- setClass("faculty", slots=list(name="character", age="numeric", GPA="numeric"))
2. A

It will generate the following output:

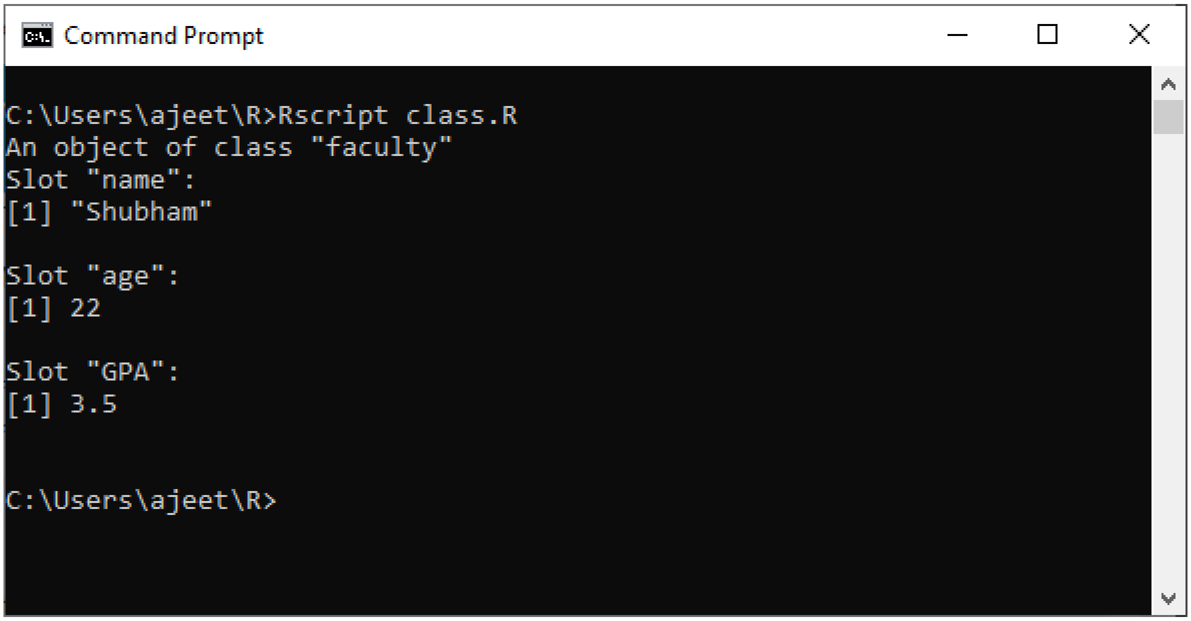


Now we can use the above constructor function to create new objects. The constructor in turn uses the new() function to create objects. It is just a wrap around. Let's see an example to understand how S4 object is created with the help of generator function.

Example

1. faculty<-setClass("faculty", slots=list(name="character", age="numeric", GPA="numeric"))
2. # creating an object using generator() function
3. # providing the class name and value for slots
4. faculty(name="Shubham", age=22, GPA=3.5)

Output



Inheritance in S4 class

Like S3 class, we can perform inheritance in S4 class also. The derived class will inherit both attributes and methods of the parent class. Let's start understanding that how we can perform inheritance in S4 class. There are the following ways to perform inheritance in S4 class:

Step 1:

In the first step, we will create or define class with appropriate slots in the following way:

```
setClass("faculty",
slots=list(name="character", age="numeric", GPA="numeric")
)
```

Step 2:

After defining class, our next step is to define class method for the display() generic function. This will be done in the following manner:

```
setMethod("show",
"faculty",
function(obj) {
cat(obj@name, "\n")
```

5. cat(obj@age, "years old\n")
6. cat("GPA:", obj@GPA, "\n")
7. }
8.)

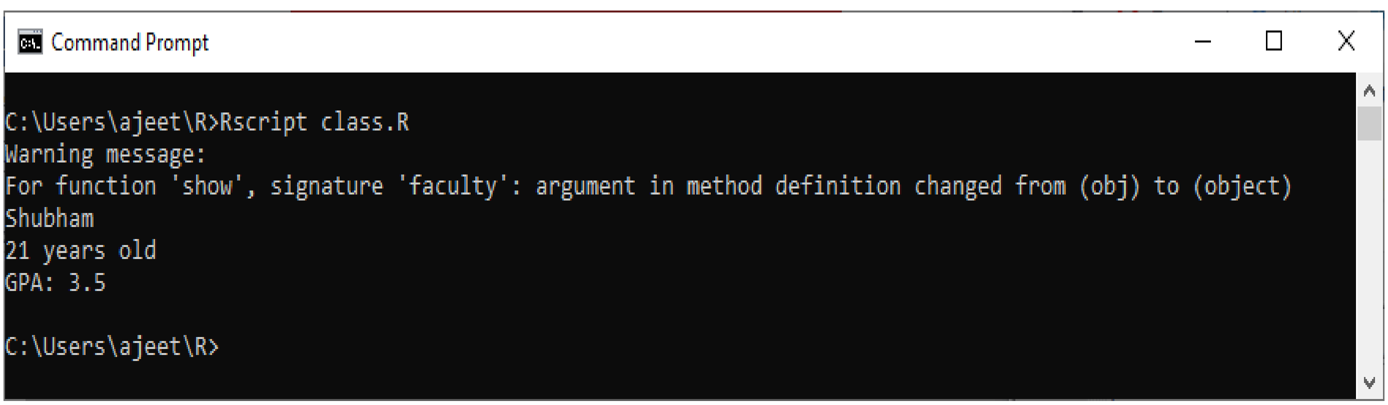
Step 3:

In the next step, we will define the derived class with the argument contains. The derived class is defined in the following way

1. setClass("Internationalfaculty",
2. slots=list(country="character"),
3. contains="faculty"
4.)

In our derived class we have defined only one attribute i.e. country. Other attributes will be inherited from its parent class.

1. s <- new("Internationalfaculty",name="John", age=21, GPA=3.5, country="India")
2. show(s)

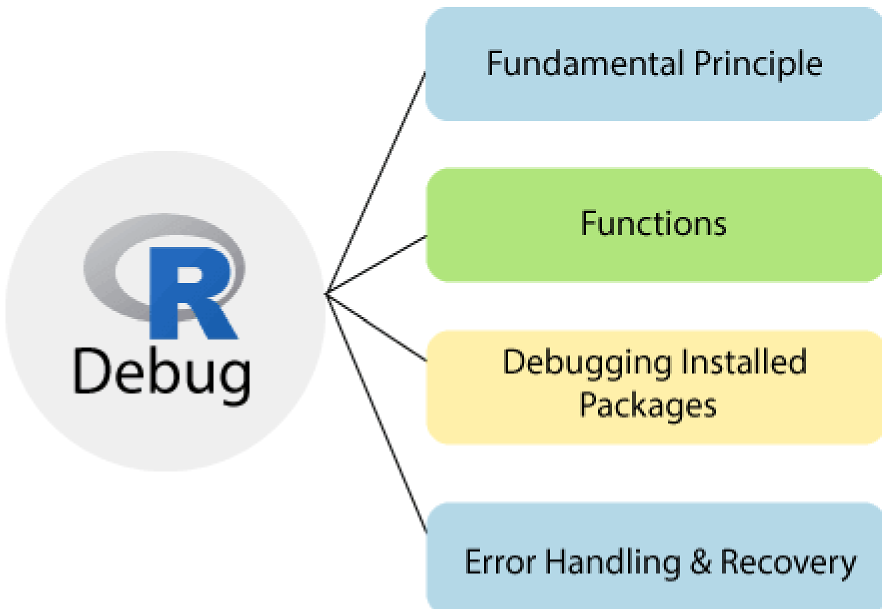


When we did show(s), the method defines for class faculty gets called. We can also define methods for the derived class of the base class as in the case of the S3 system.

What is R Debug?

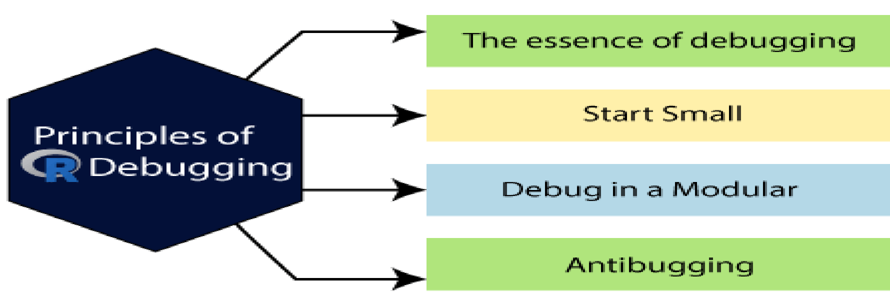
In computer programming, debugging is a multi-step process which involves identifying a problem, isolating the source of the problem, and then fixing the problem or determining a way to work around it. The final step of debugging is to test an improvement or workaround and ensure that it works.

The grammatically correct program may give us incorrect results due to some logical errors which are known as "**bug**." In case, if such errors occur, then we need to find out why and where they have occurred so that we can fix them. The procedure to identify and fix bugs is called "**debugging**."



Fundamental principles of Debugging

R programmers find that they spend more time in debugging of a program than actually writing it or code it. This makes debugging skills less valuable. In R, there are various principles of debugging which help the programmers to spend their time in writing and coding rather than in debugging. These principles are as follows:



1. The essence of debugging

Fixing a bugging is a process of confirmation. It gradually confirms that many aspects we believe to be true about the code are true actually. When it is found that one such assumption is false, there we found a clue to the bug's location.

For example

```
a <- b^2 + 3*c(z, 2)
x<- 28
if (x+q> 0)
t<- 1
else
    u<- -10
```

2. Start Small

Stick to small, simple test cases, at least at the beginning of the R debug process. Working with big data objects can make it difficult to think about the problem. Of course, we should eventually test our code in large, complex cases, but start small.

3. Debug in a Modular

Most professional software developers agree that the code should be written in a modular manner. Our first-level code should not be too long for a function call. And those functions should not be too long and should call another function if necessary. This makes the code easier to write and helps others understand when the time comes to extend the code.

We should debug in a top-down manner. Suppose we have the debug state of our function f () and it has the below line.

For example

```
Y <- g (x, 8)
```

Currently, say no to debug (g). Execute the line and see if g () returns the value that we expect. If this happens, we simply have to avoid the single-step time-consuming process through g(). If g () returns an incorrect value, now is the time to call debug (g).

4. Antibugging

If there is a section of a code in which a variable z should be positive then we can insert the following line for better performance:

Stopifnot(z>0)

When there is a bug in the code like the value of z is equal to -3, then the **Stopifnot()** function is called and will bring things right there with an error message :

Error:x>0 is not TRUE

Functions

In R, for debugging purposes, there are lots of functions available. These functions play an important role in removing bugs from our code. R provides the following functions of debugging:

1) traceback()

If our code has already crashed and we want to know where the offensive line is, try traceback (). This will (sometimes) show the location somewhere in the code of the problem. When an R function fails, an error is printed on the screen. Immediately after the error, we can call traceback () to see on which function the error occurred. The traceback () function prints the list of functions which were called before the error had occurred. The functions are printed in reverse order.

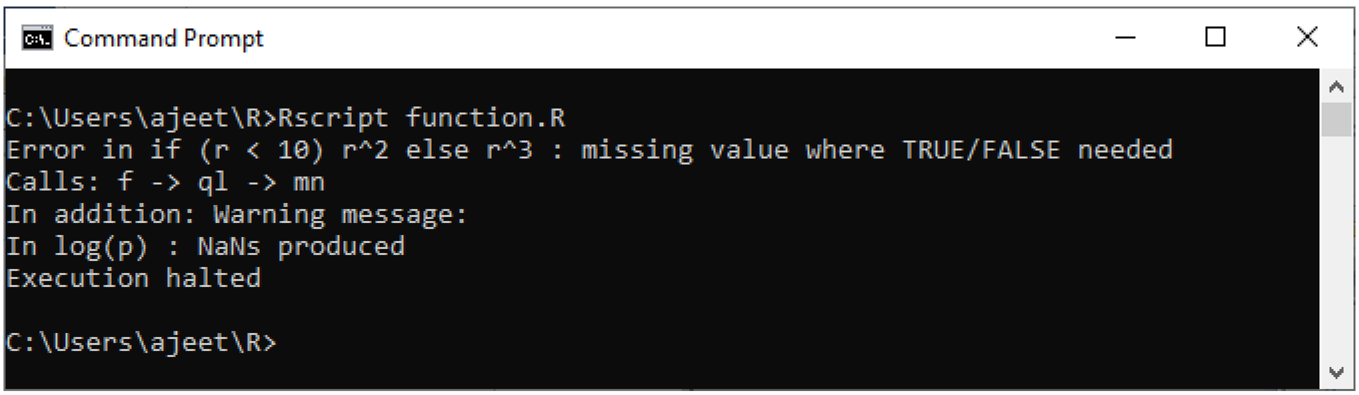
Let's see an example to understand how we can use the traceback() function

Example

```
f <- function(a){
   x <- a-ql(a)
   x
}
ql<- function(b){
   r <- b*mn(b)
   r
```

```
}
mn<- function(p){
    r <- log(p)
    if(r<10)
        r^2
    else
        r^3
}
f(-2)
```

When we run the above code, it will generate the following output:



After finding the following error we call our traceback() function and when we run, it will show the following output:

```
traceback()
```

2) debug()

In R, debug () function allows the user to step through the execution of a function. At any point, we can print the values of the variables or draw a graph of the results within the function. While debugging, we can just type "c" to continue to the end of the current block of code. Traceback () does not tell us where the function error occurred. To know which line is causing the error, we have to step through the function using debug ().

Let's see an example to understand how the debug function is used in R.

Example

```r
func<- function(a,value){
subt<- value-a
squar<- subt^2
collect <- sum(squar)
collect
}
set.seed(100)
value <- rnorm(100)
func(1,value)
debug(func)
func(1,value)
```

Output

```
Command Prompt

C:\Users\ajeet\R>Rscript function.R
[1] 202.5615
debugging in: func(1, value)
debug: {
    subt <- value - a
    squar <- subt^2
    collect <- sum(squar)
    collect
}
debug: subt <- value - a
debug: squar <- subt^2
debug: collect <- sum(squar)
debug: collect
exiting from: func(1, value)
[1] 202.5615

C:\Users\ajeet\R>
```

3) browser()

The browser() function halts the execution of a function until the user allows it to continue. This is useful if we don't want to step through the complete code, line-by-line, but we wish to stop it at a certain point so we can check what's going on.

Inserting a call into the browser() in a function will pause the function's execution at the point where the browser () is called. It is same as using debug (), except that we can control where the execution gets pause.

Let's see an example to understand how the browser() function is used in R.

Example

```
a<-function(b) {
  browser() ## a break point inserted here
  c<-log(b)
  if(c<10)
    c^2
  else
    c^3
}
a(-1)
```

Output

```
Command Prompt

C:\Users\ajeet\R>Rscript function.R
Called from: a(-1)
debug: c <- log(b)
debug: if (c < 10) c^2 else c^3
Error in if (c < 10) c^2 else c^3 : missing value where TRUE/FALSE needed
Calls: a
In addition: Warning message:
In log(b) : NaNs produced
Execution halted

C:\Users\ajeet\R>
```

4) trace()

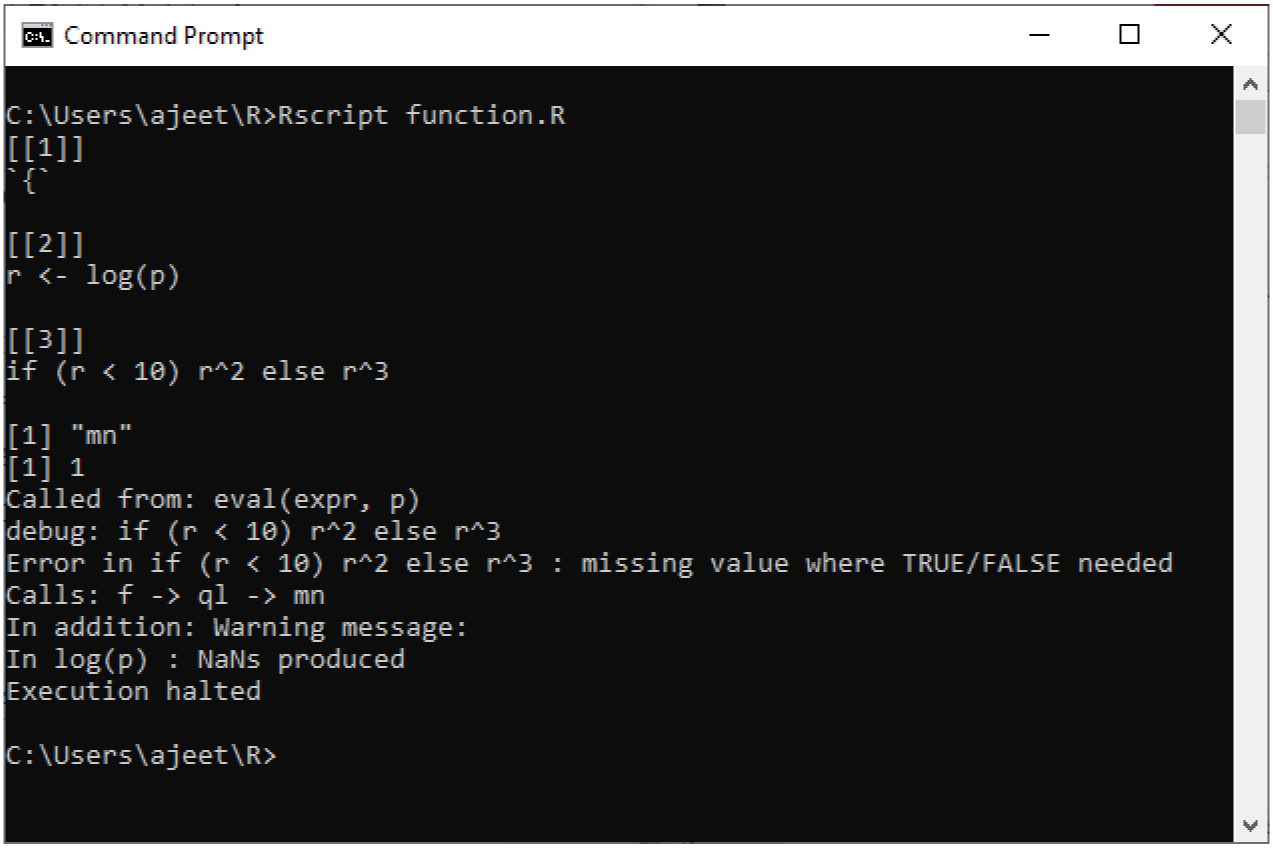
The trace() function call allows the user to insert bits of code into the function. The syntax for the R debug function trace () is a bit awkward for first-time users. It may be better to use debug ().

Let's see an example to understand how the browser() function is used in R.

Example

```
f <- function(a){
    x <- a-ql(a)
    x
}
ql<- function(b){
    r <- b*mn(b)
    r
}
mn<- function(p){
    r <- log(p)
    if(r<10)
        r^2
    else
        r^3
}
as.list(body(mn))
trace("mn",quote(if(is.nan(r)){browser()}),at=3,print=FALSE)
f(1)
f(-1)
```

Output



5) recover()

When we will perform debugging of a function, recover () allows us to examine variables in an upper-level function.

By typing a number in the selection, we are navigated to the function on the call stack and deployed in a browser environment.

The recover () function is used as an error handler, set using options () (eg. Adopt (error = retrieval)).

When a function throws an error, execution is stopped at the point of failure. We can browse the function call and examine the environment to find the source of the problem.

Example

```
f <- function(a){
   x <- a-ql(a)
   x
```

```
}
ql<- function(b){
    r <- b*mn(b)
    r
}
mn<- function(p){
    r <- log(p)
    if(r<10)
        r^2
    else
        r^3
}
as.list(body(mn))
trace("mn",quote(if(is.nan(r)){recover()}),at=3,print=FALSE)
f(-1)
```

Output

```
Command Prompt

C:\Users\ajeet\R>Rscript function.R
[[1]]
`{`

[[2]]
r <- log(p)

[[3]]
if (r < 10) r^2 else r^3

[1] "mn"
recover called non-interactively; frames dumped, use debugger() to view
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
Calls: f -> ql -> mn
In addition: Warning message:
In log(p) : NaNs produced
Execution halted

C:\Users\ajeet\R>
```

Debugging Installed Packages

There is probability of an error stemming by an installed R package. The several ways by which we can solve our problem are as follows:

- Setting the options (error = recover) and then it is proceeded line by line by the code using n.
- In complex situations, we should have a copy of the function code. In R the function entering is used to print out the function code which can be copied into the text editor. We can edit this by loading it into the global workspace and then by performing debugging.
- If our problems are not solved, then we have to download the source code. We can also use the devtools package and the install(), load_all() functions to make our procedure quicker.

Error Handling and Recovery

Exception or error handling is a process of response to odd events of code that interrupts the flow of code. In general, the scope for the exception handler begins with a try and ends with a catch. R provides the try (), and trycatch () functions for the same.

The try () function is the wrapper function for trycatch () that prints the error and then continues. On the other hand, trycatch () gives us control of the error function and, optionally, also continues the process of the function.

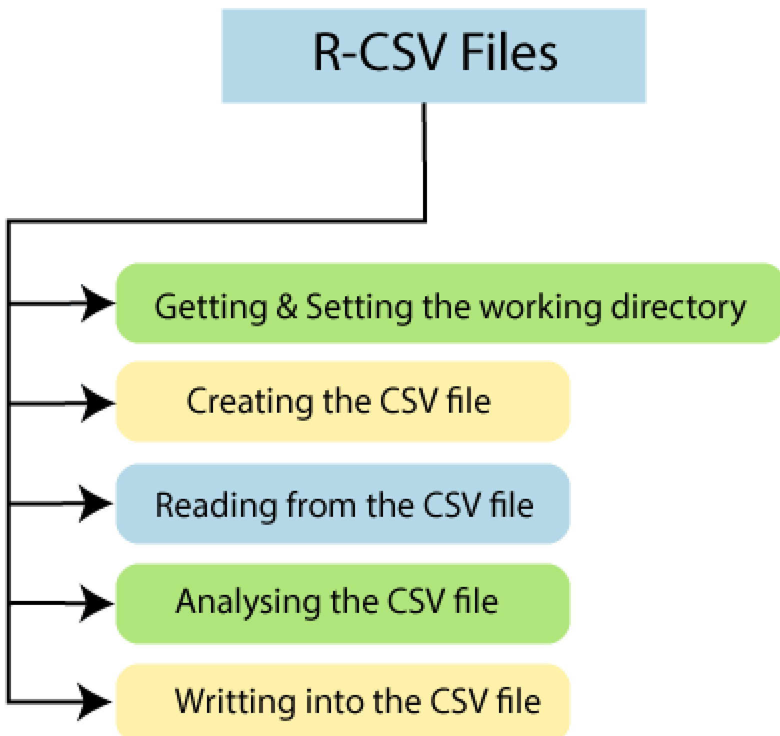
R CSV Files

A **Comma-Separated Values (CSV) file** is a plain text file which contains a list of data. These files are often used for the exchange of data between different applications. For example, databases and contact managers mostly support CSV files.

These files can sometimes be called **character-separated values** or **comma-delimited files**. They often use the comma character to separate data, but sometimes use other characters such as semicolons. The idea is that we can export the complex data from one application to a CSV file, and then importing the data in that CSV file to another application.

Storing data in excel spreadsheets is the most common way for data storing, which is used by the data scientists. There are lots of packages in R designed for accessing data from the excel spreadsheet. Users often find it easier to save their spreadsheets in comma-separated value files and then use R's built-in functionality to read and manipulate the data.

R allows us to read data from files which are stored outside the R environment. Let's start understanding how we can read and write data into CSV files. The file should be present in the current working directory so that R can read it. We can also set our directory and read file from there.



Getting and setting the working directory

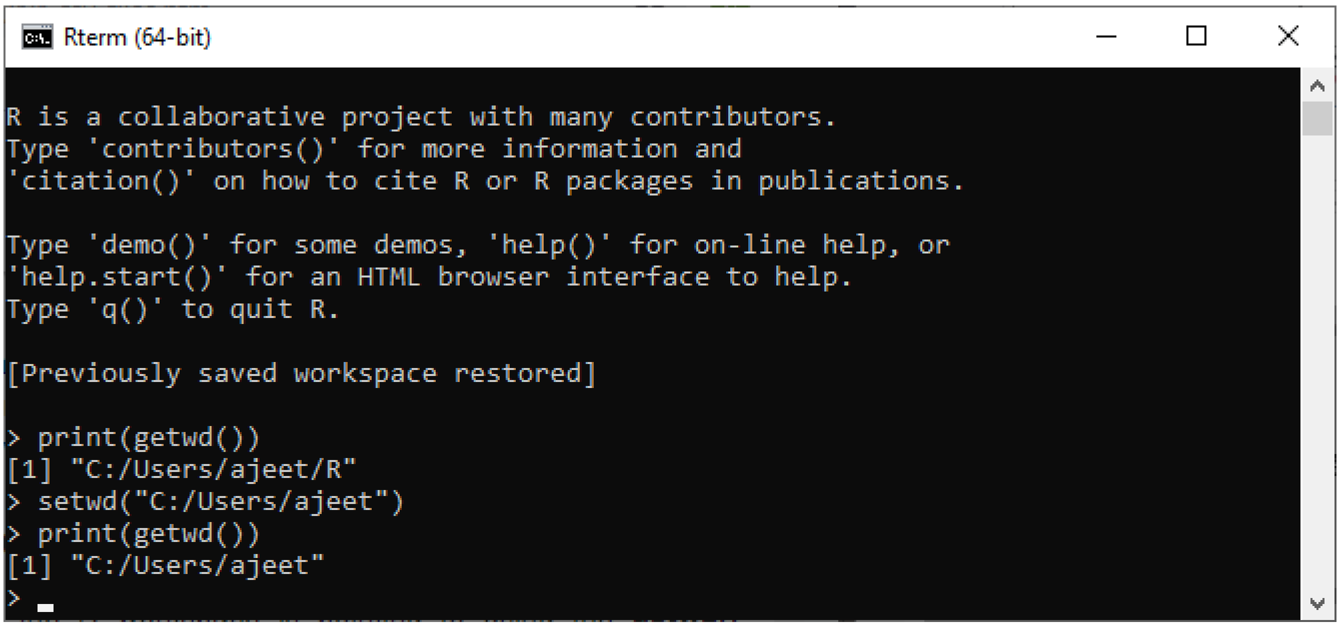
In R, getwd() and setwd() are the two useful functions. The getwd() function is used to check on which directory the R workspace is pointing. And the setwd() function is used to set a new working directory to read and write files from that directory.

Let's see an example to understand how getwd() and setwd() functions are used.

Example

```
# Getting and printing current working directory.
print(getwd())
# Setting the current working directory.
setwd("C:/Users/ajeet")
# Getting and printingthe current working directory.
print(getwd())
```

Output



Creating a CSV File

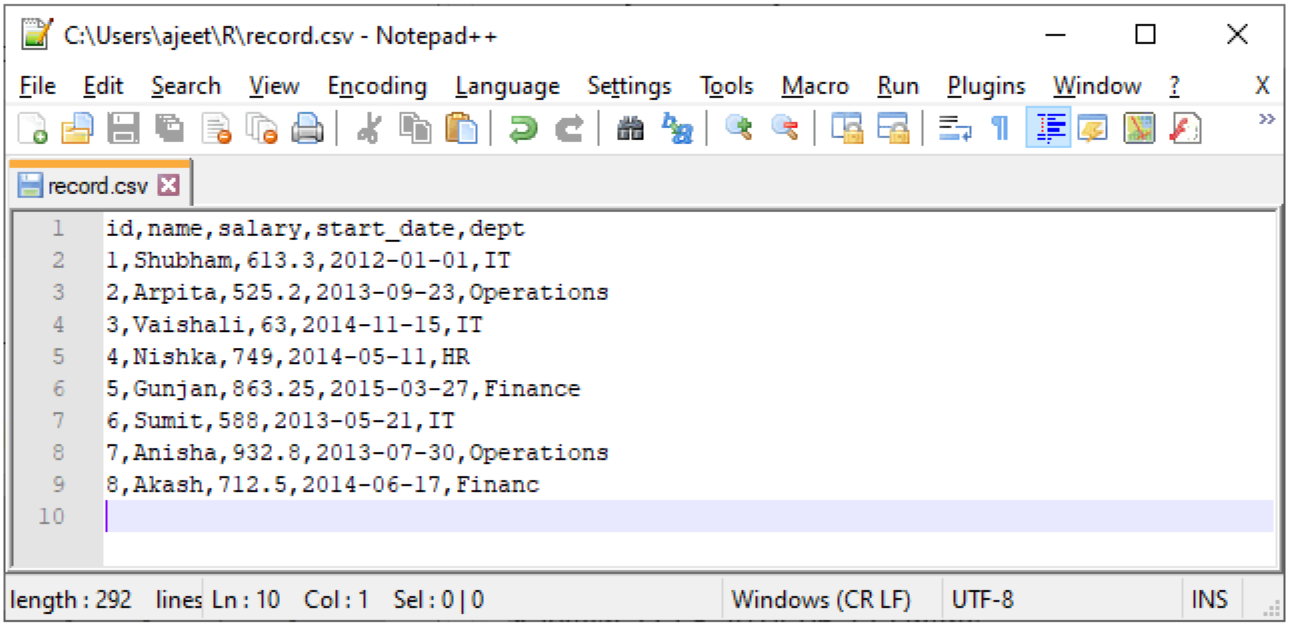
A text file in which a comma separates the value in a column is known as a CSV file. Let's start by creating a CSV file with the help of the data, which is mentioned below by saving with .csv extension using the save As All files(*.*) option in the notepad.

Example: record.csv

```
id,name,salary,start_date,dept
1,Shubham,613.3,2012-01-01,IT
2,Arpita,525.2,2013-09-23,Operations
3,Vaishali,63,2014-11-15,IT
4,Nishka,749,2014-05-11,HR
```

6. 5,Gunjan,863.25,2015-03-27,Finance
7. 6,Sumit,588,2013-05-21,IT
8. 7,Anisha,932.8,2013-07-30,Operations
9. 8,Akash,712.5,2014-06-17,Financ

Output




```
id,name,salary,start_date,dept
1,Shubham,613.3,2012-01-01,IT
2,Arpita,525.2,2013-09-23,Operations
3,Vaishali,63,2014-11-15,IT
4,Nishka,749,2014-05-11,HR
5,Gunjan,863.25,2015-03-27,Finance
6,Sumit,588,2013-05-21,IT
7,Anisha,932.8,2013-07-30,Operations
8,Akash,712.5,2014-06-17,Financ
```


Reading a CSV file

R has a rich set of functions. R provides read.csv() function, which allows us to read a CSV file available in our current working directory. This function takes the file name as an input and returns all the records present on it.

Let's use our record.csv file to read records from it using read.csv() function.

Example

1. data <- read.csv("record.csv")
2. print(data)

When we execute above code, it will give the following output

Output

Select Command Prompt

```
C:\Users\ajeet\R>Rscript datafile.R
  id     name salary start_date       dept
1  1  Shubham 613.30 2012-01-01         IT
2  2   Arpita 525.20 2013-09-23 Operations
3  3 Vaishali  63.00 2014-11-15         IT
4  4   Nishka 749.00 2014-05-11         HR
5  5   Gunjan 863.25 2015-03-27    Finance
6  6    Sumit 588.00 2013-05-21         IT
7  7   Anisha 932.80 2013-07-30 Operations
8  8    Akash 712.50 2014-06-17     Financ

C:\Users\ajeet\R>_
```

Analyzing the CSV File

When we read data from the .csv file using **read.csv()** function, by default, it gives the output as a data frame. Before analyzing data, let's start checking the form of our output with the help of **is.data.frame()** function. After that, we will check the number of rows and number of columns with the help of **nrow()** and **ncol()** function.

Example

1. csv_data<- read.csv("record.csv")
2. print(is.data.frame(csv_data))
3. print(ncol(csv_data))
4. print(nrow(csv_data))

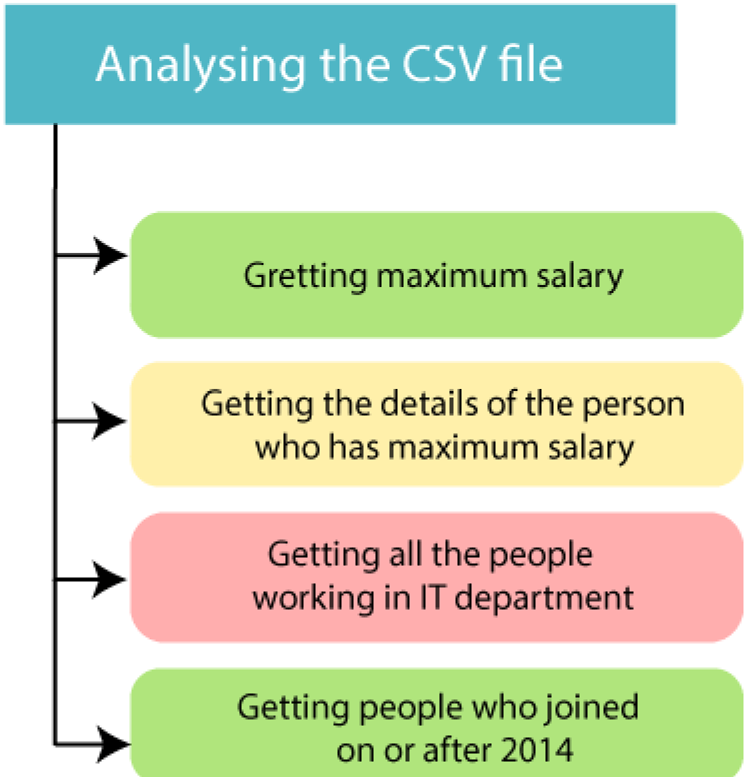
When we run above code, it will generate the following output:

Output

Command Prompt

```
C:\Users\ajeet\R>Rscript datafile.R
[1] TRUE
[1] 5
[1] 8

C:\Users\ajeet\R>_
```

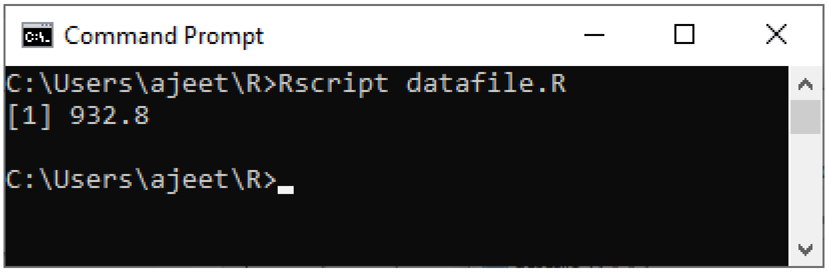
From the above output, it is clear that our data is read in the form of the data frame. So we can apply all the functions of the data frame, which we have discussed in the earlier sections.



Example: Getting the maximum salary

```
# Creating a data frame.
csv_data<- read.csv("record.csv")
# Getting the maximum salary from data frame.
max_sal<- max(csv_data$salary)
print(max_sal)
```

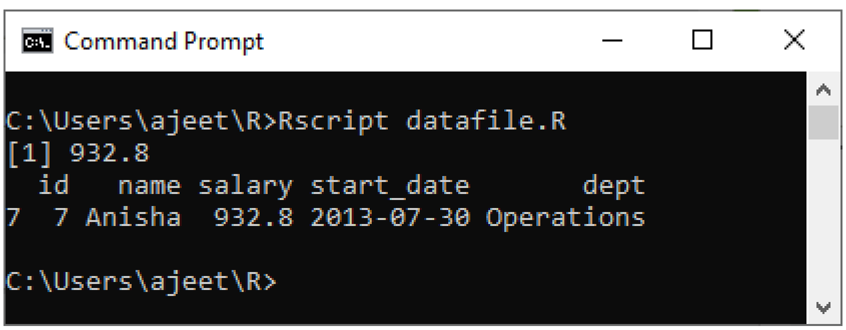
Output


```
C:\Users\ajeet\R>Rscript datafile.R
[1] 932.8

C:\Users\ajeet\R>_
```

Example: Getting the details of the person who have a maximum salary

```
# Creating a data frame.
csv_data<- read.csv("record.csv")
# Getting the maximum salary from data frame.
max_sal<- max(csv_data$salary)
print(max_sal)
#Getting the detais of the pweson who have maximum salary
details <- subset(csv_data,salary==max(salary))
print(details)
```

Output


```
C:\Users\ajeet\R>Rscript datafile.R
[1] 932.8
  id   name salary start_date       dept
7  7 Anisha  932.8 2013-07-30 Operations

C:\Users\ajeet\R>
```

Example: Getting the details of all the persons who are working in the IT department

```
# Creating a data frame.
csv_data<- read.csv("record.csv")
#Getting the detais of all the pweson who are working in IT department
details <- subset(csv_data,dept=="IT")
print(details)
```

Output

```
C:\Users\ajeet\R>Rscript datafile.R
  id      name salary start_date dept
1  1   Shubham  613.3 2012-01-01   IT
3  3 Vaishali    63.0 2014-11-15   IT
6  6     Sumit  588.0 2013-05-21   IT

C:\Users\ajeet\R>
```

Example: Getting the details of the persons whose salary is greater than 600 and working in the IT department.

```
# Creating a data frame.
csv_data<- read.csv("record.csv")
#Getting the detais of all the pweson who are working in IT department
details <- subset(csv_data,dept=="IT"&salary>600)
print(details)
```

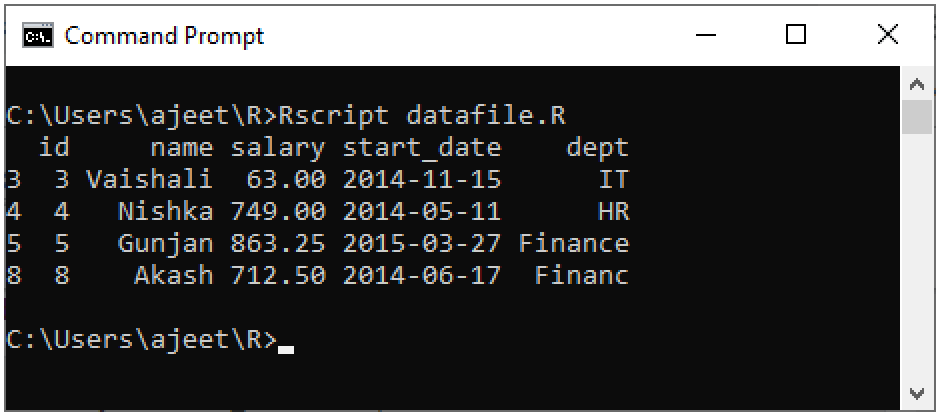
Output

```
C:\Users\ajeet\R>Rscript datafile.R
  id     name salary start_date dept
1  1 Shubham  613.3 2012-01-01   IT

C:\Users\ajeet\R>
```

Example: Getting details of those peoples who joined on or after 2014.

```
# Creating a data frame.
csv_data<- read.csv("record.csv")
#Getting details of those peoples who joined on or after 2014
details <- subset(csv_data,as.Date(start_date)>as.Date("2014-01-01"))
print(details)
```

Output



Writing into a CSV file

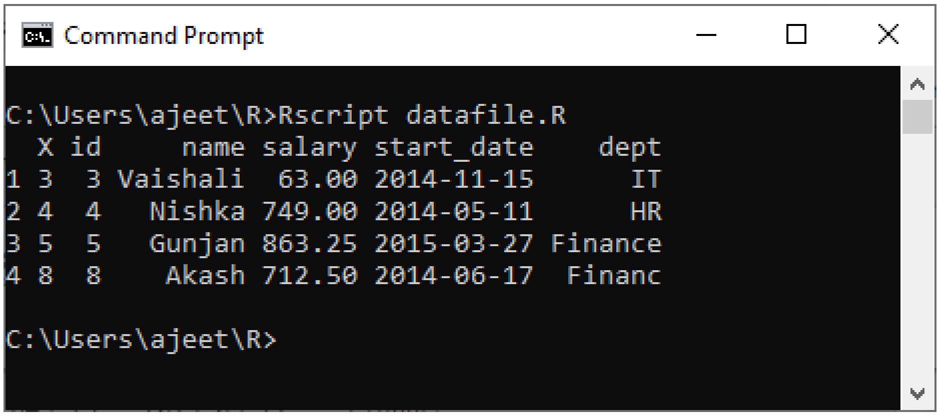
Like reading and analyzing, R also allows us to write into the .csv file. For this purpose, R provides a write.csv() function. This function creates a CSV file from an existing data frame. This function creates the file in the current working directory.

Let's see an example to understand how **write.csv()** function is used to create an output CSV file.

Example

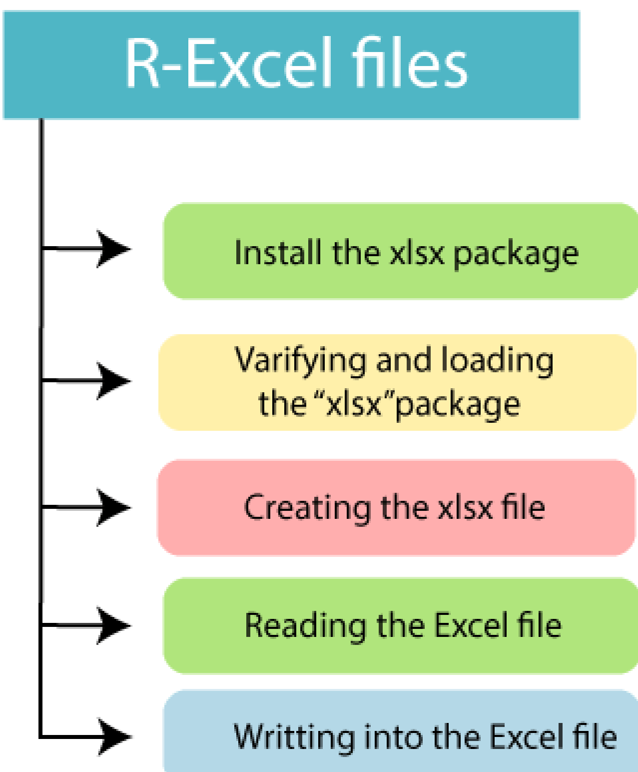
1. csv_data<- read.csv("record.csv")
2. #Getting details of those peoples who joined on or after 2014
3. details <- subset(csv_data,as.Date(start_date)>as.Date("2014-01-01"))
4. # Writing filtered data into a new file.
5. write.csv(details,"output.csv")
6. new_details<- read.csv("output.csv")
7. print(new_details)

Output



R Excel file

The xlsx is a file extension of a spreadsheet file format which was created by Microsoft to work with Microsoft Excel. In the present era, Microsoft Excel is a widely used spreadsheet program that sores data in the .xls or .xlsx format. R allows us to read data directly from these files by providing some excel specific packages. There are lots of packages such as XLConnect, xlsx, gdata, etc. We will use xlsx package, which not only allows us to read data from an excel file but also allow us to write data in it.



Install xlsx Package

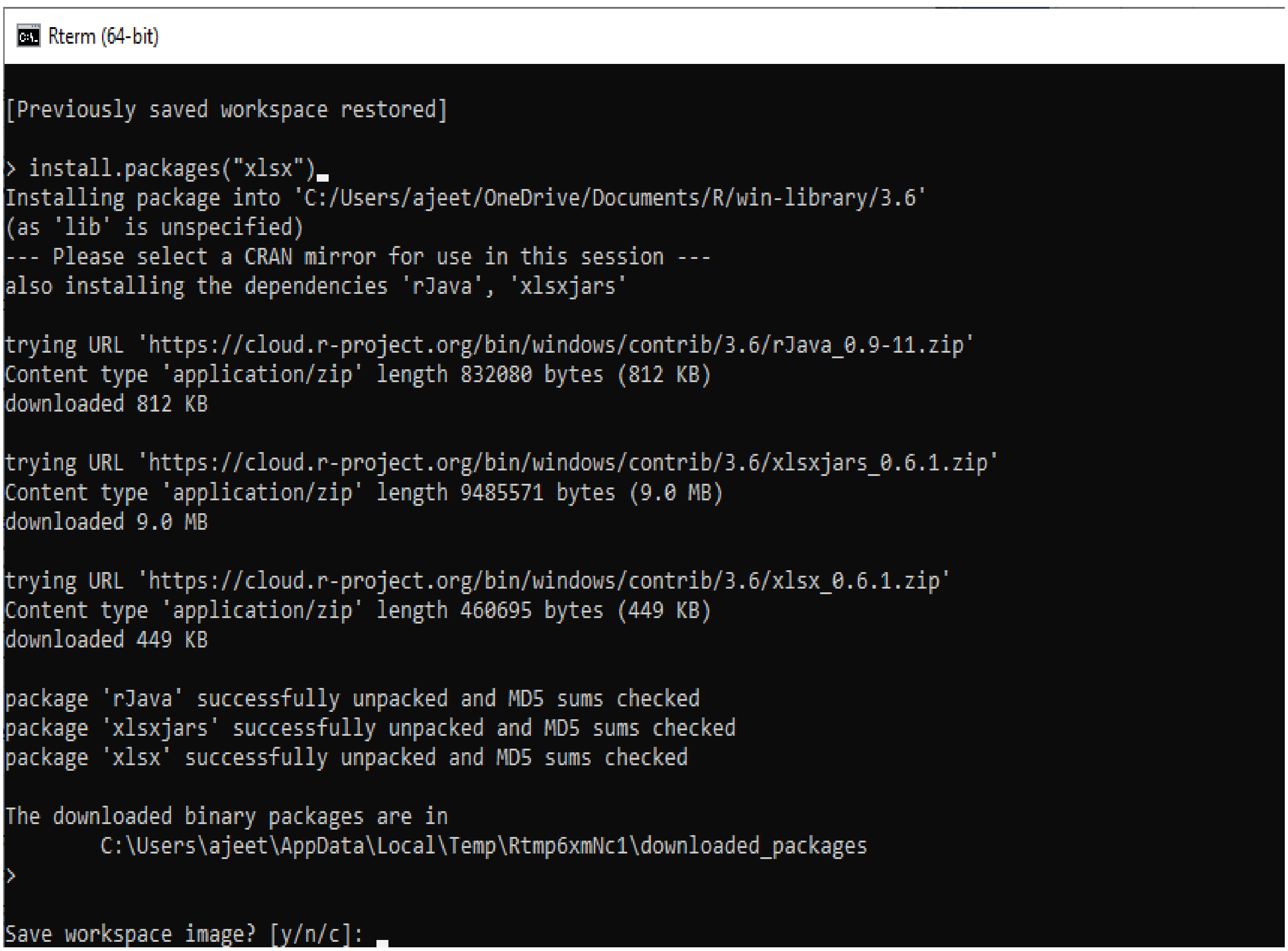
Our primary task is to install "xlsx" package with the help of install.package command. When we install the xlsx package, it will ask us to install some additional packages on which this package is dependent. For installing the additional packages, the same command is used with the required package name. There is the following syntax of install command:

1. install.packages("package name")

Example

1. install.packages("xlsx")

Output



```
Rterm (64-bit)

[Previously saved workspace restored]

> install.packages("xlsx")
Installing package into 'C:/Users/ajeet/OneDrive/Documents/R/win-library/3.6'
(as 'lib' is unspecified)
--- Please select a CRAN mirror for use in this session ---
also installing the dependencies 'rJava', 'xlsxjars'

trying URL 'https://cloud.r-project.org/bin/windows/contrib/3.6/rJava_0.9-11.zip'
Content type 'application/zip' length 832080 bytes (812 KB)
downloaded 812 KB

trying URL 'https://cloud.r-project.org/bin/windows/contrib/3.6/xlsxjars_0.6.1.zip'
Content type 'application/zip' length 9485571 bytes (9.0 MB)
downloaded 9.0 MB

trying URL 'https://cloud.r-project.org/bin/windows/contrib/3.6/xlsx_0.6.1.zip'
Content type 'application/zip' length 460695 bytes (449 KB)
downloaded 449 KB

package 'rJava' successfully unpacked and MD5 sums checked
package 'xlsxjars' successfully unpacked and MD5 sums checked
package 'xlsx' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\ajeet\AppData\Local\Temp\Rtmp6xmNc1\downloaded_packages
>

Save workspace image? [y/n/c]:
```

Verifying and Loading of "xlsx" Package

In R, grepl() and any() functions are used to verify the package. If the packages are installed, these functions will return True else return False. For verifying the package, both the functions are used together.

For loading purposes, we use the library() function with the appropriate package name. This function loads all the additional packages also.

Example

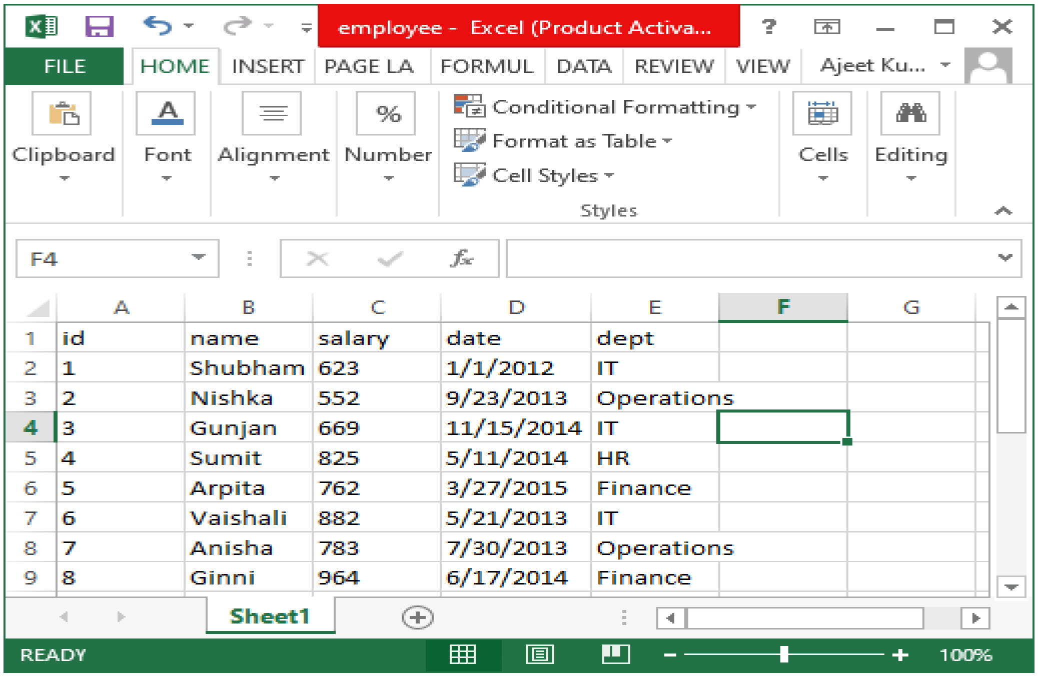
1. #Installing xlsx package
2. install.packages("xlsx")
3. # Verifying the package is installed.
4. any(grepl("xlsx",installed.packages()))
5. # Loading the library into R workspace.
6. library("xlsx")

Output

```
Rterm (64-bit)
Type 'q()' to quit R.

[Previously saved workspace restored]

> #Installing xlsx package
> install.packages("xlsx")
Installing package into 'C:/Users/ajeet/OneDrive/Documents/R/win-library/3.6'
(as 'lib' is unspecified)
--- Please select a CRAN mirror for use in this session ---
trying URL 'https://cloud.r-project.org/bin/windows/contrib/3.6/xlsx_0.6.1.zip'
Content type 'application/zip' length 460695 bytes (449 KB)
downloaded 449 KB

package 'xlsx' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\ajeet\AppData\Local\Temp\RtmpkpsFGi\downloaded_packages
>
> # Verifying the package is installed.
> any(grepl("xlsx",installed.packages()))
[1] TRUE
>
> # Loading the library into R workspace.
> library("xlsx")
> print(labrary("xlsx"))
Error in labrary("xlsx") : could not find function "labrary"
> print(library("xlsx"))
[1] "xlsx"      "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"
> 
```

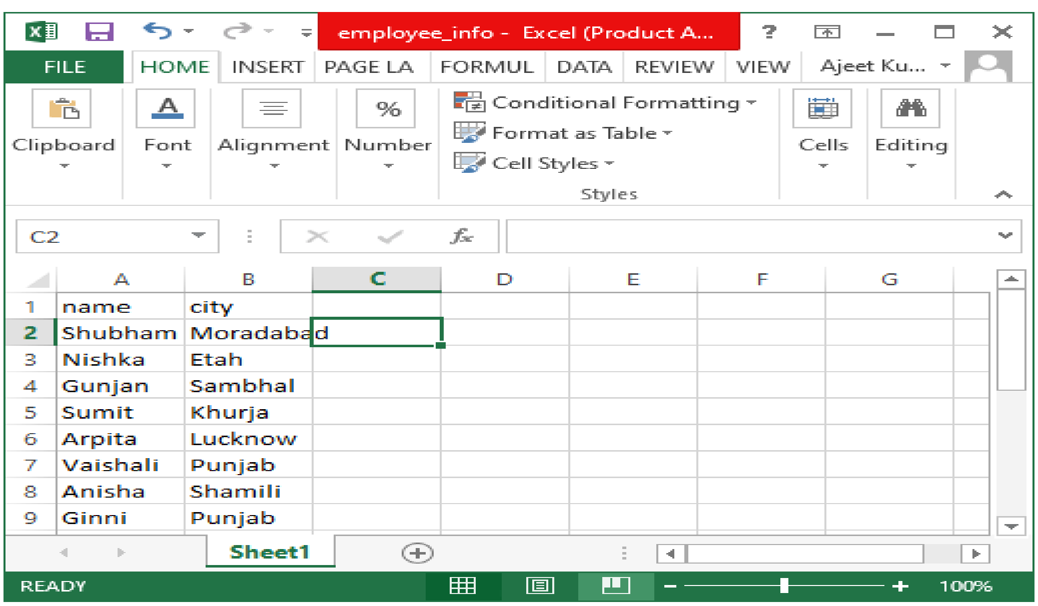
Creating an xlsx File

Once the xlsx package is loaded into our system, we will create an excel file with the following data and named it employee.



id	name	salary	date	dept
1	Shubham	623	1/1/2012	IT
2	Nishka	552	9/23/2013	Operations
3	Gunjan	669	11/15/2014	IT
4	Sumit	825	5/11/2014	HR
5	Arpita	762	3/27/2015	Finance
6	Vaishali	882	5/21/2013	IT
7	Anisha	783	7/30/2013	Operations
8	Ginni	964	6/17/2014	Finance

Apart from this, we will create another table with the following data and give it a name as employee_info.



name	city
Shubham	Moradabad
Nishka	Etah
Gunjan	Sambhal
Sumit	Khurja
Arpita	Lucknow
Vaishali	Punjab
Anisha	Shamili
Ginni	Punjab

Note: ***Both the files will be saved in the current working directory of the R workspace.***

Reading the Excel File

Like the CSV file, we can read data from an excel file. R provides read.xlsx() function, which takes two arguments as input, i.e., file name and index of the sheet. This function returns the excel data in the form of a data frame in the R environment. There is the following syntax of read.xlsx() function:

```
1. read.xlsx(file_name,sheet_index)
```

Let's see an example in which we read data from our employee.xlsx file.

Example

```
1. #Loading xlsx package
2. library("xlsx")
3.  # Reading the first worksheet in the file employee.xlsx.
4. excel_data<- read.xlsx("employee.xlsx", sheetIndex = 1)
5. print(excel_data)
```

Output

```
Command Prompt

C:\Users\ajeet\R>Rscript excel.R
  id     name salary       date       dept
1  1  Shubham    623 2012-01-01         IT
2  2   Nishka    552 2013-09-23 Operations
3  3   Gunjan    669 2014-11-15         IT
4  4    Sumit    825 2014-05-11         HR
5  5   Arpita    762 2015-03-27    Finance
6  6 Vaishali    882 2013-05-21         IT
7  7   Anisha    783 2013-07-30 Operations
8  8    Ginni    964 2014-06-17    Finance

C:\Users\ajeet\R>
```

Writing data into Excel File

In R, we can also write the data into our .xlsx file. R provides a write.xlsx() function to write data into the excel file. There is the following syntax of write.xlsx() function:

```
write.xlsx(data_frame,file_name,col.names,row.names,sheetnames,append)
```

Here,

- The data_frame is our data, which we want to insert into our excel file.
- The file_names is the name of that file in which we want to insert our data.
- The col.names and row.names are the logical values that are specifying whether the column names/row names of the data frame are to be written to the file.
- The append is a logical value, which indicates our data should be appended or not into an existing file.

Let's see an example to understand how write.xlsx() function works with its parameters.

Example

```
#Loading xlsx package
library("xlsx")
#Creating data frame
emp.data<- data.frame(
name = c("Raman","Rafia","Himanshu","jasmine","Yash"),
salary = c(623.3,915.2,611.0,729.0,843.25),
start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11","2015-03-27")),
dept = c("Operations","IT","HR","IT","Finance"),
stringsAsFactors = FALSE
)
# Writing the first data set in employee.xlsxRscript
write.xlsx(emp.data, file = "employee.xlsx", col.names=TRUE, row.names=TRUE,sheetName="Sheet2",append = TRUE)
 # Reading the first worksheet in the file employee.xlsx.
excel_data<- read.xlsx("employee.xlsx", sheetIndex = 1)
print(excel_data)
  # Reading the first worksheet in the file employee.xlsx.
```

```
excel_data <- read.xlsx("employee.xlsx", sheetIndex = 2)
print(excel_data)
```

Output

```
Command Prompt

C:\Users\ajeet\R>Rscript excel.R
  id      name salary start_date       dept
1  1   Shubham    623 2012-01-01         IT
2  2    Nishka    552 2013-09-23 Operations
3  3    Gunjan    669 2014-11-15         IT
4  4     Sumit    825 2014-05-11         HR
5  5    Arpita    762 2015-03-27    Finance
6  6 Vaishali    882 2013-05-21         IT
7  7    Anisha    783 2013-07-30 Operations
8  8     Ginni    964 2014-06-17     Financ
  NA.     name salary start_date       dept
1   1    Raman 623.30 2012-01-01 Operations
2   2    Rafia 915.20 2013-09-23         IT
3   3 Himanshu 611.00 2014-11-15         HR
4   4  jasmine 729.00 2014-05-11         IT
5   5     Yash 843.25 2015-03-27    Finance

C:\Users\ajeet\R>
```

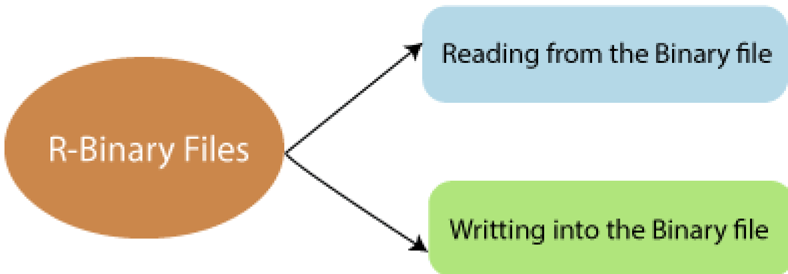
R Binary File

A binary file is a file which contains information present only in the form of bits and bytes(0's and 1's). They are not human-readable because the bytes translate into characters and symbols that contain many other non-printable characters. If we will read a binary file using any text editor, it will show the characters like ð and Ø.

The code is relatively very easy to read binary data into R. To read binary data, we must know how a piece of information has been parsed into binary.

The binary file must be read by specific programs to be useful. For example, the binary file of a Microsoft Word program can only be read by the Word program in a human-readable form. It indicates that, in addition to human-readable text, there is a lot of information such as character formatting and page numbers, etc., which are also stored with alphanumeric characters. And finally, a binary file is a contiguous sequence of bytes. The line break we see in a text file is a character joining the first line to the next line.

Sometimes, the data generated by other programs need to be processed by R as a binary file. Also, R needs to create binary files that can be shared with other programs. There are two functions writeBin () and readBin () for creating and reading binary files in R.



Writing the Binary File

Like CSV and Excel files, we can also write into a binary file. R provides a writeBin() function for writing the data into a binary file. There is the following syntax of writeBin() function:

```
1. writeBin(object,con)
```

Here,

- The ?con' is the connection object which is used to write the binary file.
- The ?object' is the binary file in which we write our data.

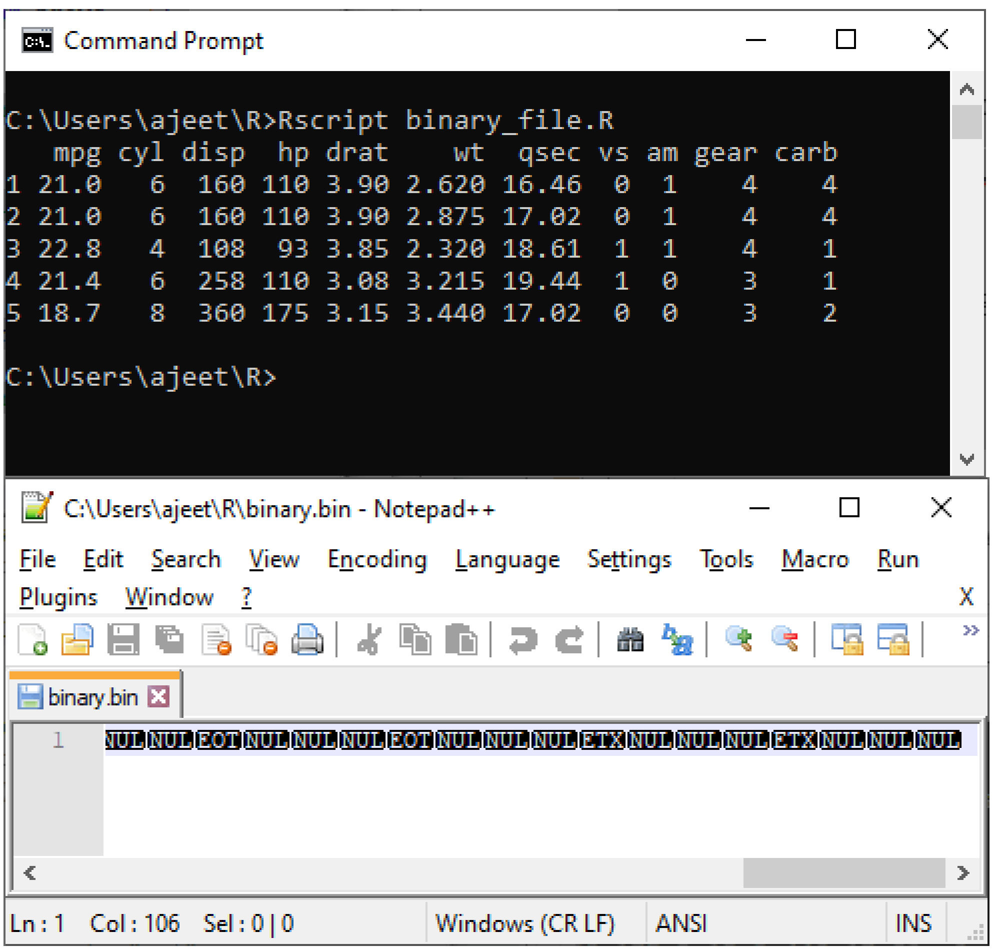
Let's see an example to understand how this function is used to write data into a file in binary format. In the following example, we will use R inbuilt data "mtcars." We will create a CSV file from it and convert it into a binary file.

Example

```
1. # Reading the "mtcars" data frame as a csv file and will store only the columns "cyl", "am
   " and "gear".
2. write.table(mtcars, file = "mtcars.csv",row.names = FALSE, na = "",
3. col.names = TRUE, sep = ",")
```

```
# Storing 5 records from the csv file as a new data frame.
new.mtcars <- read.table("mtcars.csv",sep = ",",header = TRUE,nrows = 5)
new.mtcars
# Creating a connection object to write the binary file using mode "wb".
write.filename = file("/Users/ajeet/R/binary.bin", "wb")
 # Writing the column names of the data frame to the connection object.
writeBin(colnames(new.mtcars), write.filename)
# Writing the records in each of the column to the file.
writeBin(c(new.mtcars$cyl,new.mtcars$am,new.mtcars$gear), write.filename)
 # Closing the file for writing so that other programs can read it.
close(write.filename)
```

Output



Reading the Binary File

We can also read our binary file which we have created before. For this purpose, R provides a readBin() function for reading the data from a binary file.

There is the following syntax of readbin() function:

```
readBin(con,what,n)
```

Here,

- The ?con' is the connection object which is used to read the binary file.
- The ?what' is the mode such as character, integer, etc. which represent the bytes to be read.
- The ?n' is the number of bytes which we want to read from the binary file.

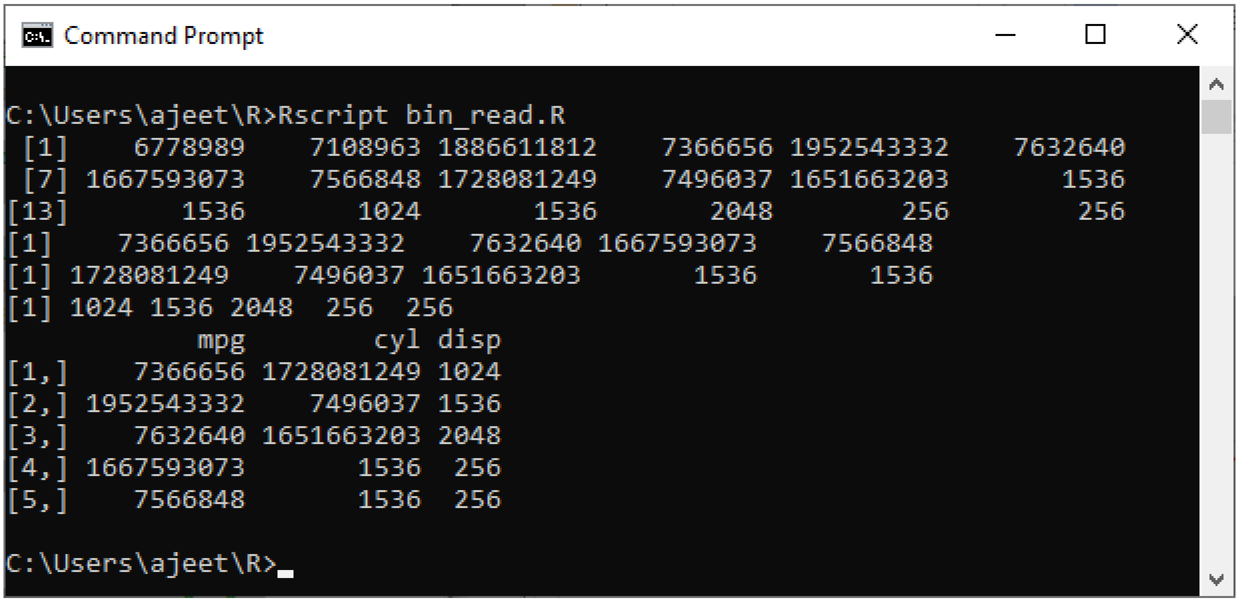
Let's see an example in which we read our binary data from binary.bin file.

Example

```
# Creating a connection object to read the file in binary mode using "rb".
read.filename <- file("/Users/ajeet/R/binary.bin", "rb")
# Reading the column names. n = 3 as we have 3 columns.
column.names <- readBin(read.filename, character(),  n = 3)
# Reading the column values. n = 18 as we have 3 column names and 15 values.
read.filename <- file("/Users/ajeet/R/binary.bin", "rb")
bin_data <- readBin(read.filename, integer(),  n = 18)
# Printing the data.
print(bin_data)
 # Reading the values from 4th byte to 8th byte, which represents "cyl."
cyl_data = bin_data[4:8]
print(cyl_data)
  # Reading the values form 9th byte to 13th byte which represents "am".
am_data = bin_data[9:13]
print(am_data)
# Reading the values form 9th byte to 13th byte which represents "gear".
gear_data = bin_data[14:18]
print(gear_data)
```

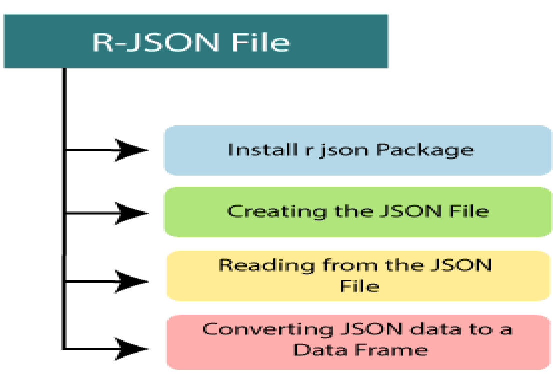
```
# Combining all the read values to a dat frame.
final_data = cbind(cyl_data, am_data, gear_data)
colnames(final_data) = column.names
print(final_data)
```

Output



R JSON File

JSON stands for JavaScript Object Notation. The JSON file contains the data as text in a human-readable format. Like other files, we can also read and write into the JSON files. For this purpose, R provides a package named rjson, which we have to install with the help of the familiar command **install.packages**.

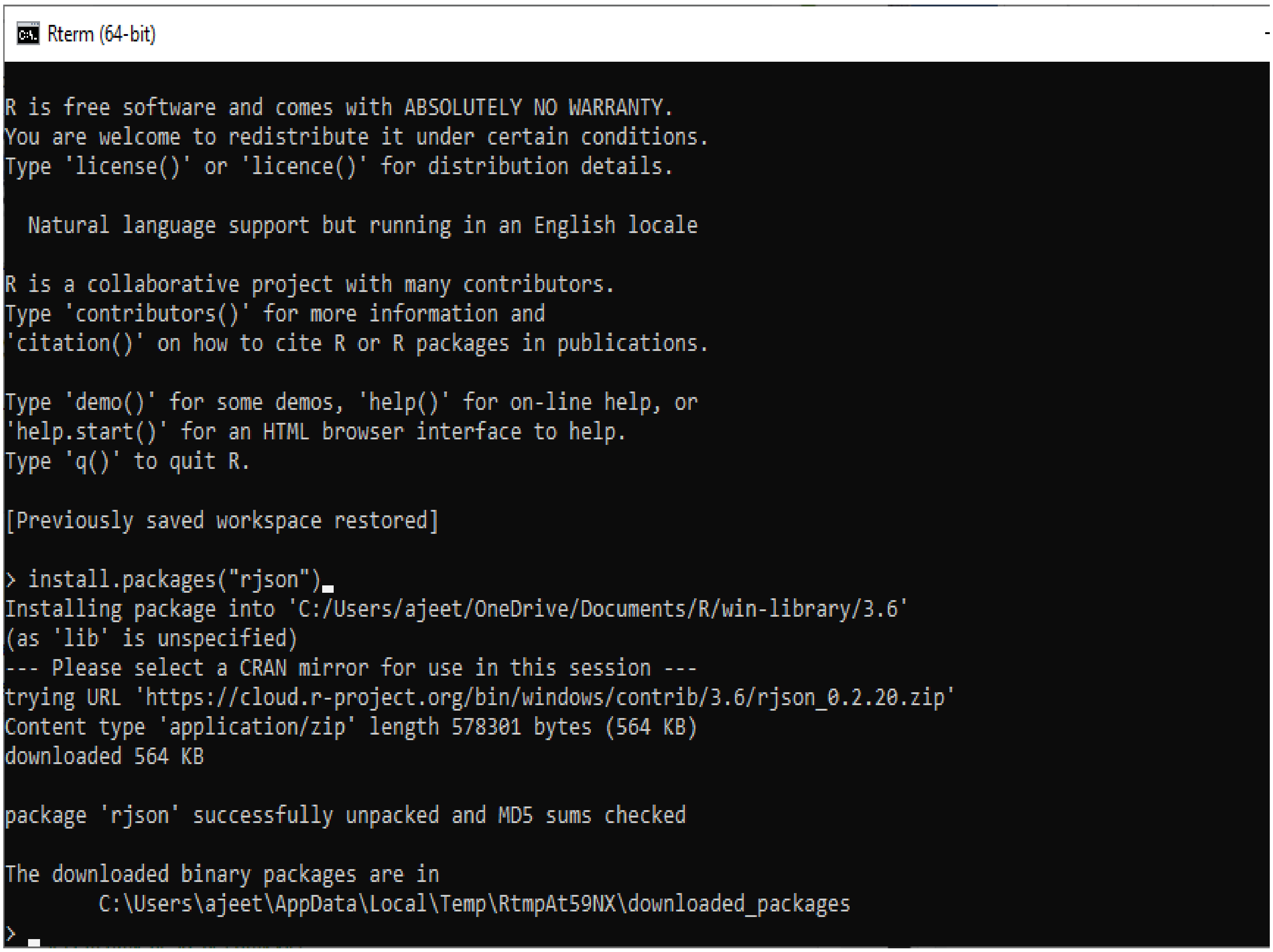


Install rjson package

By running the following command into the R console, we will install the rjson package into our current working directory.

1. install.packages("rjson")

Output



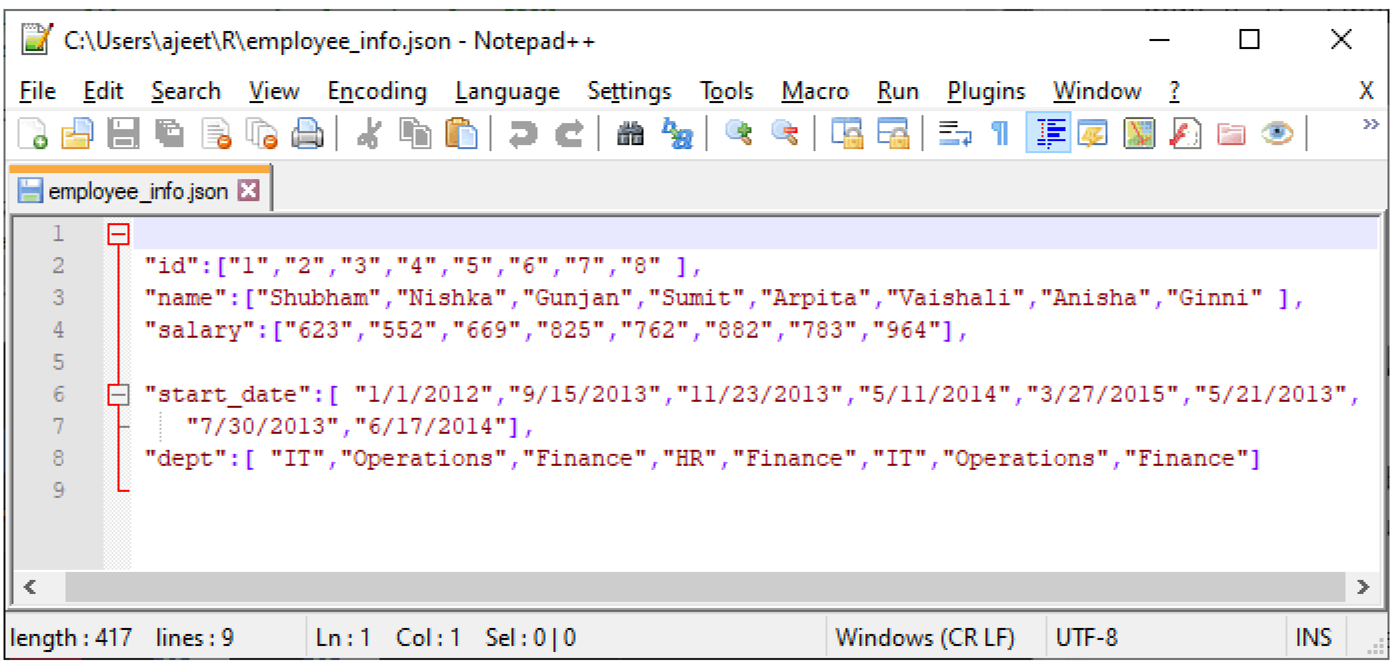
Creating a JSON file

The extension of JSON file is .json. To create the JSON file, we will save the following data as employee_info.json. We can write the information of employees in any text editor with its appropriate rule of writing the JSON file. In JSON files, the information contains in between the curly braces({}).

Example: employee_info.json

```
{
  "id":["1","2","3","4","5","6","7","8" ],
  "name":["Shubham","Nishka","Gunjan","Sumit","Arpita","Vaishali","Anisha","Ginni" ],
  "salary":["623","552","669","825","762","882","783","964"],

   "start_date":[ "1/1/2012","9/15/2013","11/23/2013","5/11/2014","3/27/2015","5/21/2013",
      "7/30/2013","6/17/2014"],
  "dept":[ "IT","Operations","Finance","HR","Finance","IT","Operations","Finance"]
}
```

Output



Read the JSON file

Reading the JSON file in R is a very easy and effective process. R provide from JSON() function to extract data from a JSON file. This function, by default, extracts the data in the form of a list. This function takes the JSON file and returns the records which are contained in it.

Let's see an example to understand how fromJSON() function is used to extract data and print the result in the form of a list. We will consider the employee_info.json file which we have created before.

Example

1. # Loading the package which is required to read JSON files.
2. library("rjson")
3. # Giving the input file name to the function fromJSON.
4. result <- fromJSON(file = "employee_info.json")
5. # Printing the result.
6. print(result)

Output

```
Command Prompt
C:\Users\ajeet\R>Rscript json.R
$id
[1] "1" "2" "3" "4" "5" "6" "7" "8"

$name
[1] "Shubham"  "Nishka"   "Gunjan"   "Sumit"    "Arpita"   "Vaishali" "Anis
ha"
[8] "Ginni"

$salary
[1] "623" "552" "669" "825" "762" "882" "783" "964"

$start_date
[1] "1/1/2012"   "9/15/2013"  "11/23/2013" "5/11/2014"  "3/27/2015"
[6] "5/21/2013"  "7/30/2013"  "6/17/2014"

$dept
[1] "IT"         "Operations" "Finance"    "HR"         "Finance"
```

Converting JSON data to a Data Frame

R provide, as.data.frame() function to convert the extracted data into data frame. For further analysis, data analysts use this function. Let's start an example to see how this function is used, and in our example, we will consider our employee_info.json file.

Example

1. # Loading the package which is required to read JSON files.
2. library("rjson")
3. # Giving the input file name to the function fromJSON.
4. result <- fromJSON(file = "employee_info.json")
5. # Converting the JSON record to a data frame.
6. data_frame <- as.data.frame(result)
7. #Printing JSON data frame
8. print(data_frame)

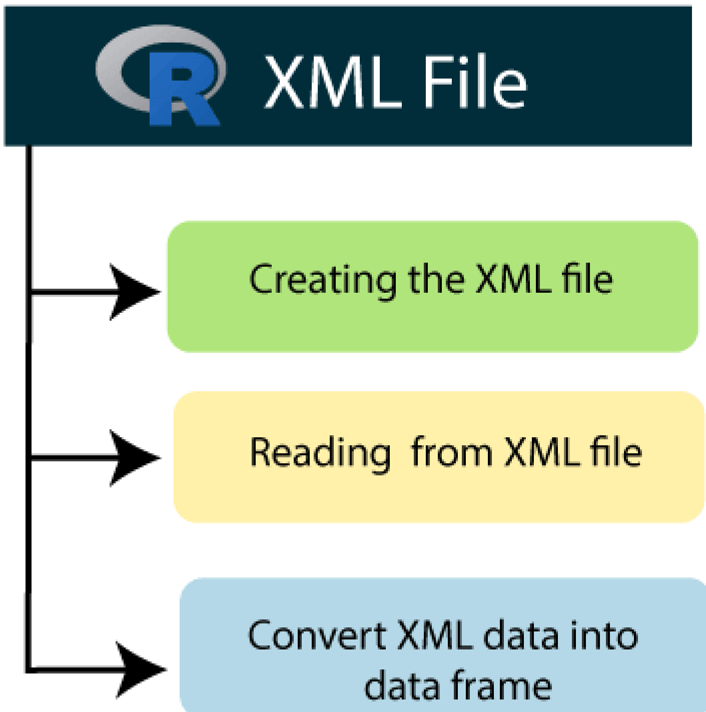
Output

```
C:\Users\ajeet\R>Rscript json.R
  id     name salary start_date       dept
1  1  Shubham    623   1/1/2012         IT
2  2   Nishka    552  9/15/2013 Operations
3  3   Gunjan    669 11/23/2013    Finance
4  4    Sumit    825  5/11/2014         HR
5  5   Arpita    762  3/27/2015    Finance
6  6 Vaishali    882  5/21/2013         IT
7  7   Anisha    783  7/30/2013 Operations
8  8    Ginni    964  6/17/2014    Finance

C:\Users\ajeet\R>
```

R XML File

Like HTML, XML is also a markup language which stands for Extensible Markup Language. It is developed by World Wide Web Consortium(W3C) to define the syntax for encoding documents which both humans and machine can read. This file contains markup tags. There is a difference between HTML and XML. In HTML, the markup tag describes the structure of the page, and in xml, it describes the meaning of the data contained in the file. In R, we can read the xml files by installing "XML" package into the R environment. This package will be installed with the help of the familiar command i.e., install.packages.

1. install.packages("XML")



Creating XML File

We will create an xml file with the help of the given data. We will save the following data with the .xml file extension to create an xml file. XML tags describe the meaning of data, so that data contained in such tags can easily tell or explain about the data.

Example: xml_data.xml

```
1. <records>
2. <employee_info>
3. <id>1</id>
4. <name>Shubham</name>
5. <salary>623</salary>
```

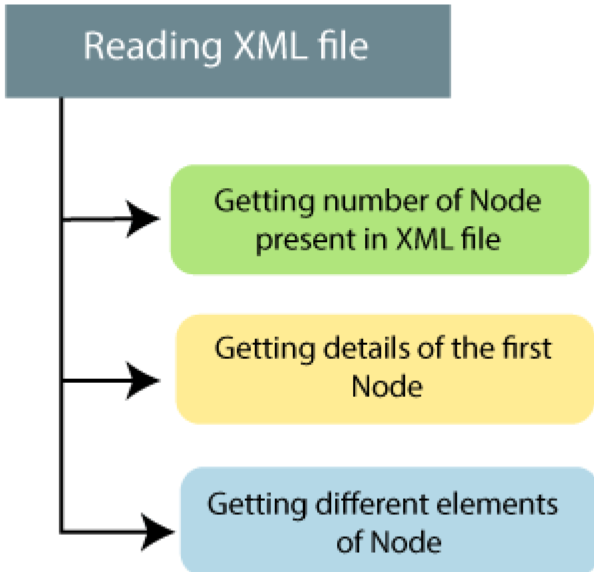
```xml
<date>1/1/2012</date>
<dept>IT</dept>
</employee_info>
<employee_info>
<id>2</id>
<name>Nishka</name>
<salary>552</salary>
<date>1/1/2012</date>
<dept>IT</dept>
</employee_info>
<employee_info>
<id>1</id>
<name>Gunjan</name>
<salary>669</salary>
<date>1/1/2012</date>
<dept>IT</dept>
</employee_info>
<employee_info>
<id>1</id>
<name>Sumit</name>
<salary>825</salary>
<date>1/1/2012</date>
<dept>IT</dept>
</employee_info>
<employee_info>
<id>1</id>
<name>Arpita</name>
<salary>762</salary>
<date>1/1/2012</date>
<dept>IT</dept>
```

```
</employee_info>

<employee_info>
<id>1</id>
<name>Vaishali</name>
<salary>882</salary>
<date>1/1/2012</date>
<dept>IT</dept>
</employee_info>

<employee_info>
<id>1</id>
<name>Anisha</name>
<salary>783</salary>
<date>1/1/2012</date>
<dept>IT</dept>
</employee_info>

<employee_info>
<id>1</id>
<name>Ginni</name>
<salary>964</salary>
<date>1/1/2012</date>
<dept>IT</dept>
</employee_info>

</records>
```

Reading XML File

In R, we can easily read an xml file with the help of xmlParse() function. This function is stored as a list in R. To use this function, we first need to load the xml package with the help of the library() function. Apart from the xml package, we also need to load one additional package named methods.

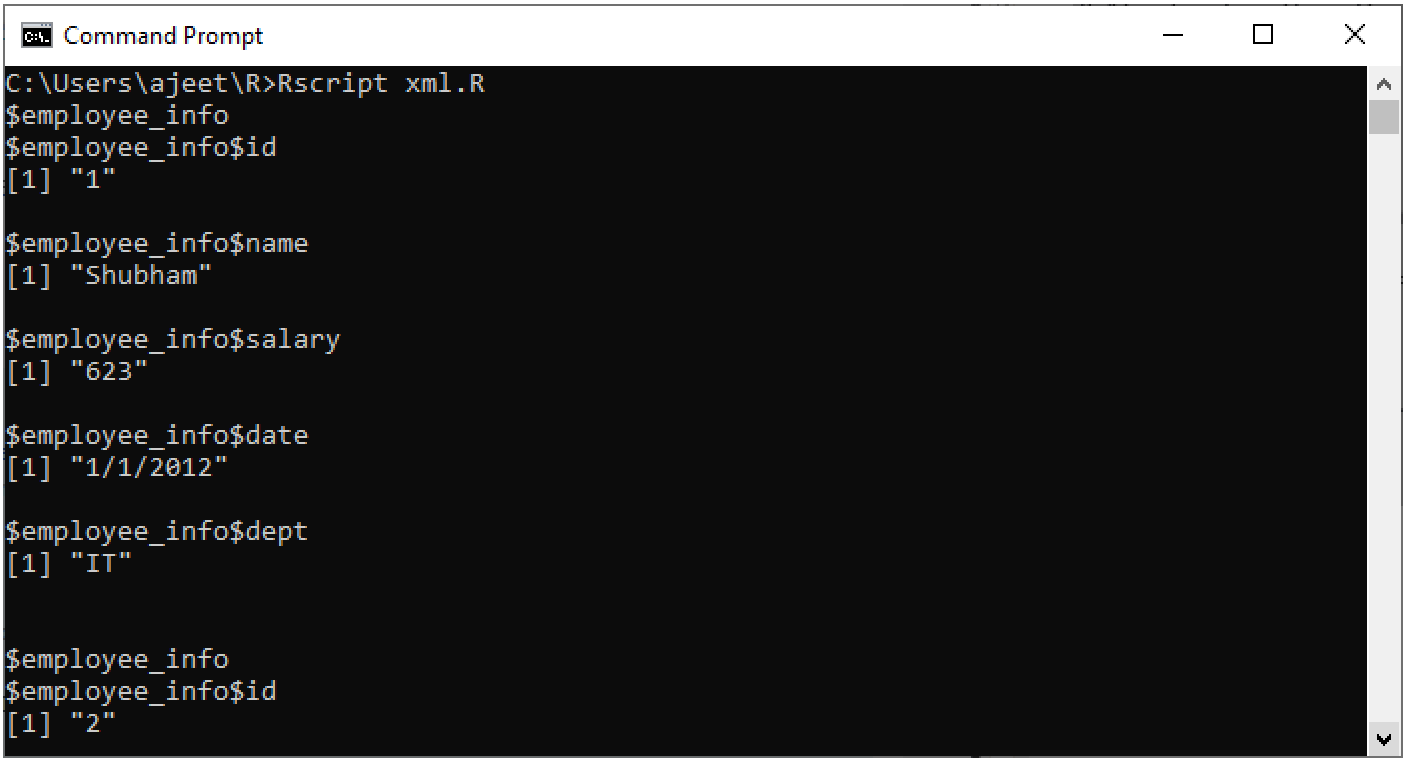
Let's see an example to understand the working of xmlParse() function in which we read our xml_data.xml file.



Example: Reading xml data in the form of a list.

```
# Loading the package required to read XML files.
library("XML")
# Also loading the other required package.
library("methods")
# Giving the input file name to the function.
result <- xmlParse(file = "xml_data.xml")
xml_data <- xmlToList(result)
print(xml_data)
```

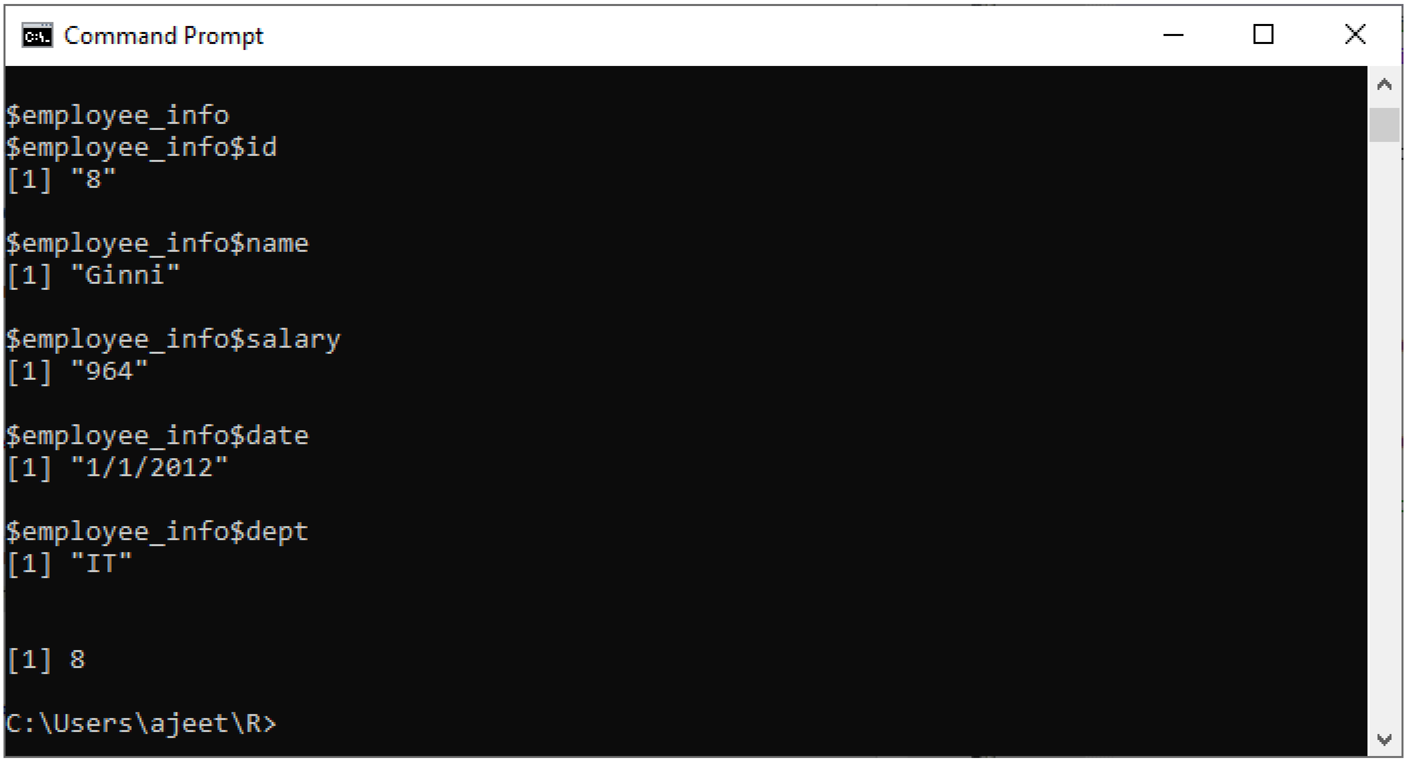
Output



Example: Getting number of nodes present in xml file.

```
# Loading the package required to read XML files.
library("XML")
# Also loading the other required package.
library("methods")
# Giving the input file name to the function.
result <- xmlParse(file = "xml_data.xml")
#Converting the data into list
xml_data <- xmlToList(result)
#Printing the data
print(xml_data)
# Exracting the root node form the xml file.
root_node <- xmlRoot(result)
# Finding the number of nodes in the root.
root_size <- xmlSize(root_node)
# Printing the result.
print(root_size)
```

Output

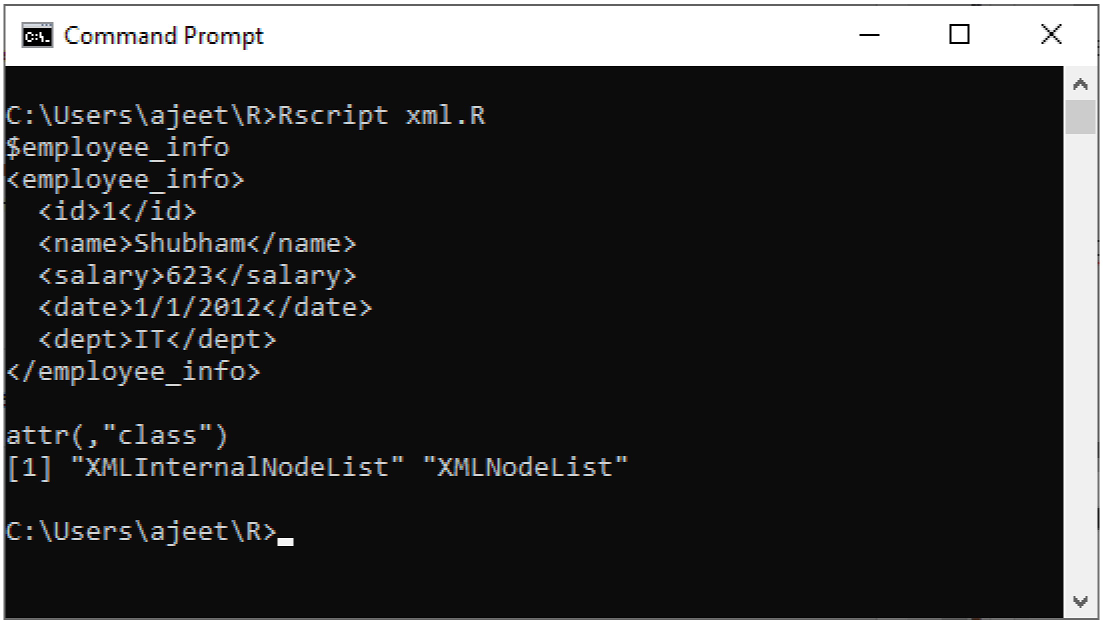


```
$employee_info
$employee_info$id
[1] "8"

$employee_info$name
[1] "Ginni"

$employee_info$salary
[1] "964"

$employee_info$date
[1] "1/1/2012"

$employee_info$dept
[1] "IT"


[1] 8

C:\Users\ajeet\R>
```

Example: Getting details of the first node in xml.

```
# Loading the package required to read XML files.
library("XML")
# Also loading the other required package.
library("methods")
# Giving the input file name to the function.
result <- xmlParse(file = "xml_data.xml")
# Exracting the root node form the xml file.
root_node <- xmlRoot(result)
# Printing the result.
print(root_node[1])
```

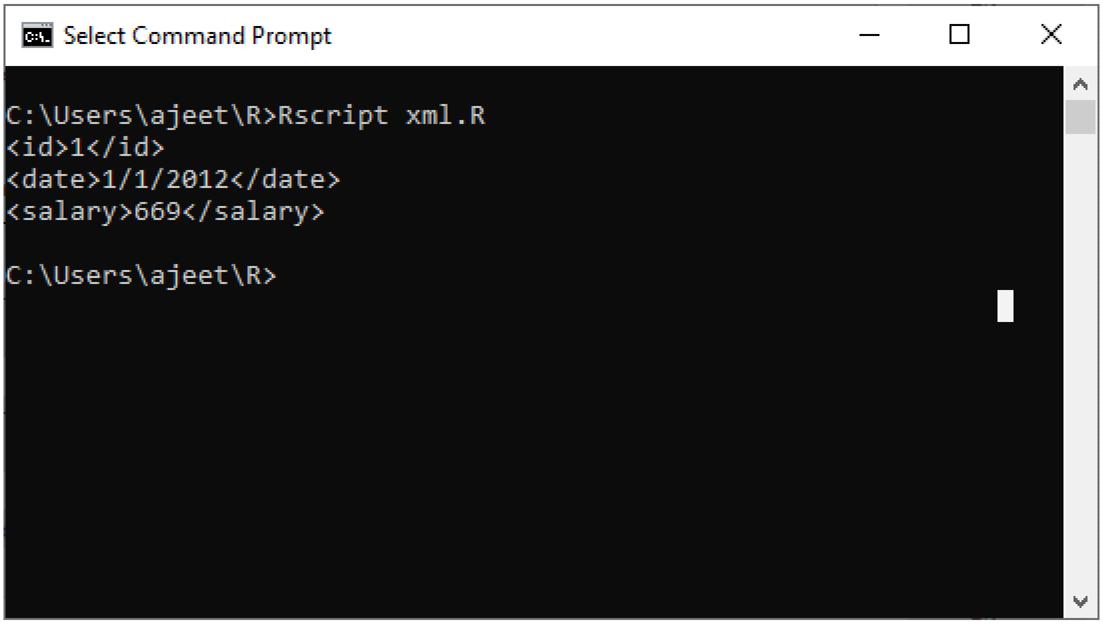
Output



Example: Getting details of different elements of a node.

```
# Loading the package required to read XML files.
library("XML")
# Also loading the other required package.
library("methods")
# Giving the input file name to the function.
result <- xmlParse(file = "xml_data.xml")
# Exracting the root node form the xml file.
root_node <- xmlRoot(result)
# Getting the first element of the first node.
print(root_node[[1]][[1]])
# Getting the fourth element of the first node.
print(root_node[[1]][[4]])
# Getting the third element of the third node.
print(root_node[[3]][[3]])
```

Output



How to convert xml data into a data frame

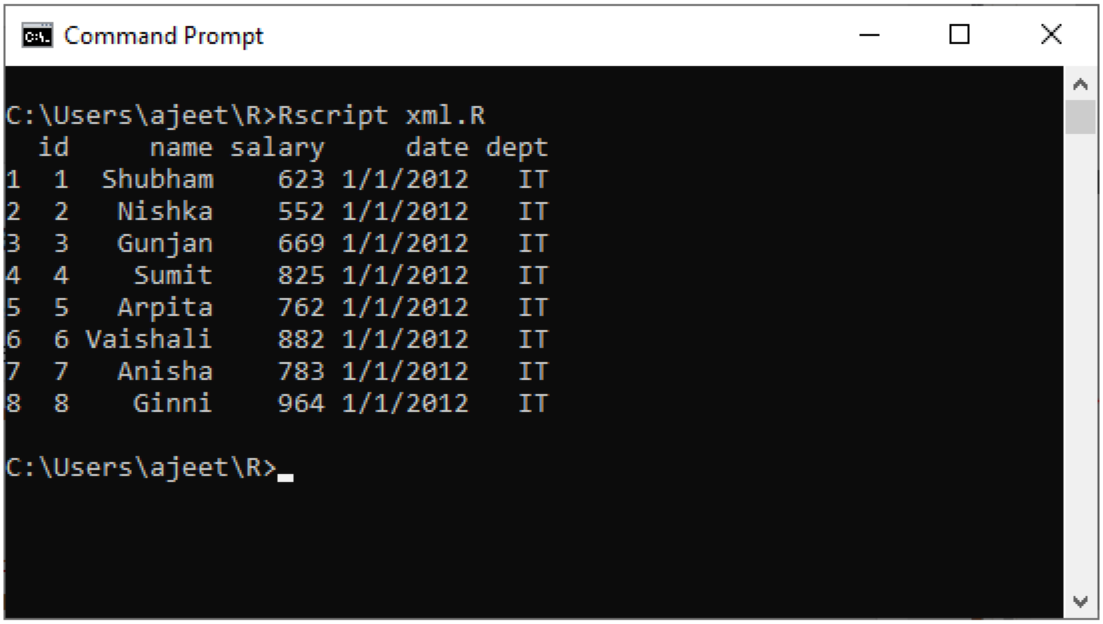
It's not easy to handle data effectively in large files. For this purpose, we read the data in the xml file as a data frame. Then this data frame is processed by the data analyst. R provide xmlToDataFrame() function to extract the information in the form of Data Frame.

Let's see an example to understand how this function is used and processed:

Example

```
1. # Loading the package required to read XML files.
2. library("XML")
3. # Also loading the other required package.
4. library("methods")
5. # Giving the input file name to the function xmlToDataFrame.
6. data_frame <- xmlToDataFrame("xml_data.xml")
7. #Printing the result
8. print(data_frame)
```

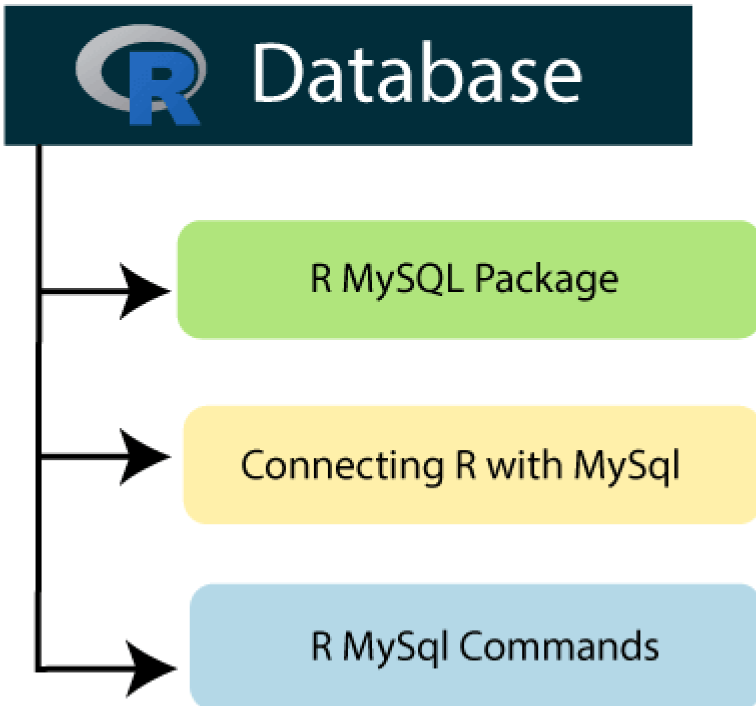
Output



R Database

In the relational database management system, the data is stored in a normalized format. Therefore, to complete statistical computing, we need very advanced and complex SQL queries. The large and huge data which is present in the form of tables require SQL queries to extract the data from it.

R can easily connect with many of the relational databases like MySql, SQL Server, Oracle, etc. When we extract the information from these databases, by default, the information is extracted in the form of data frame. Once, the data comes from the database to the R environment; it will become a normal R dataset. The data analyst can easily analyze or manipulate the data with the help of all the powerful packages and functions.



RMySQL Package

RMySQL package is one of the most important built-in package of R. This package provides native connectivity between the R and MySql database. In R, to work with MySql database, we first have to install the RMySQL package with the help of the familiar command, which is as follows:

1. install.packages("RMySQL")

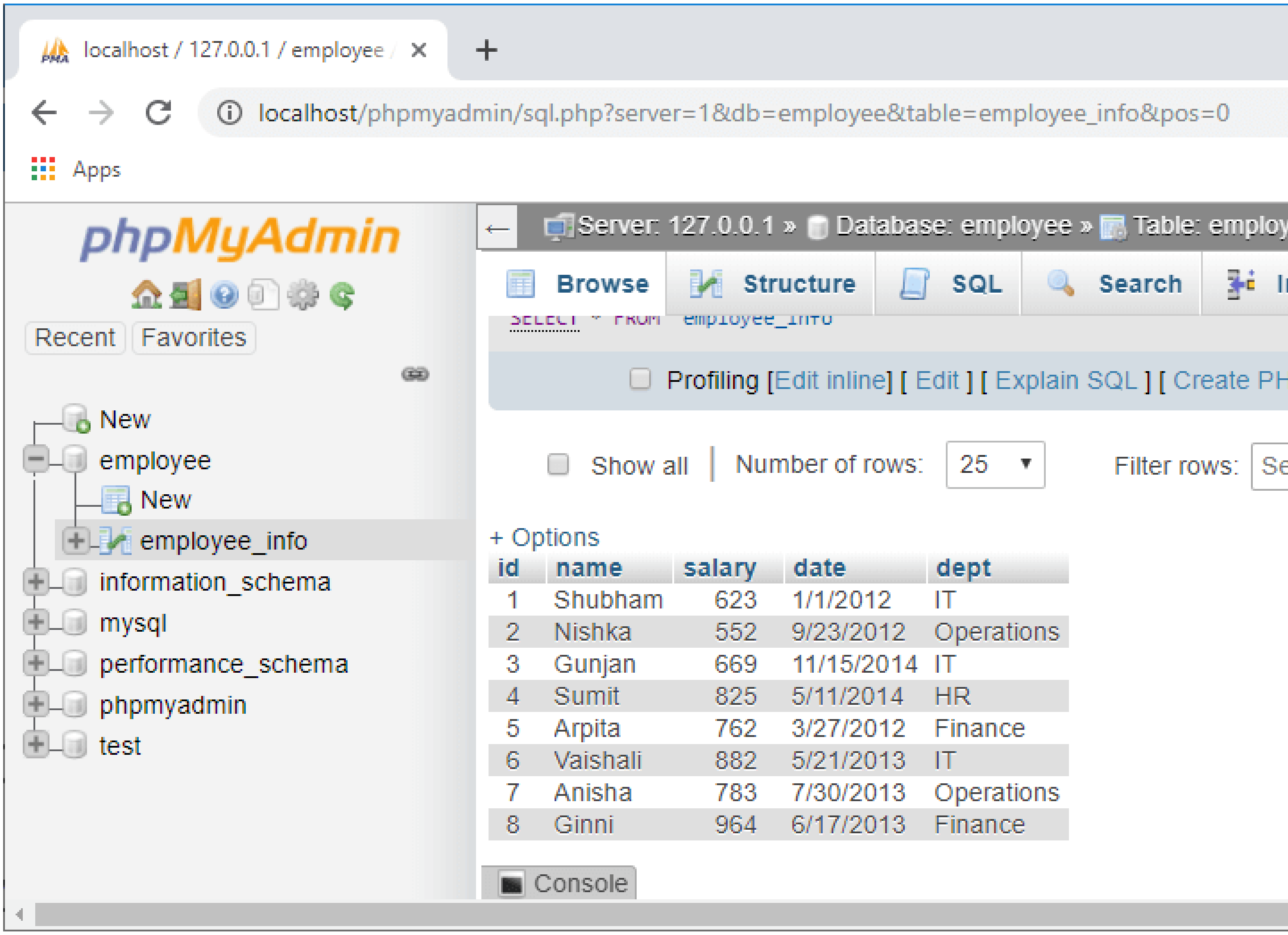
When we run the above command in the R environment, it will start downloading the package RMySQL.

Output

Rterm (64-bit)

```
'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.

[Previously saved workspace restored]

> install.packages("RMySQL")
Installing package into 'C:/Users/ajeet/OneDrive/Documents/R/win-library/3.6'
(as 'lib' is unspecified)
--- Please select a CRAN mirror for use in this session ---
also installing the dependency 'DBI'

trying URL 'https://cloud.r-project.org/bin/windows/contrib/3.6/DBI_1.0.0.zip'
Content type 'application/zip' length 889106 bytes (868 KB)
downloaded 868 KB

trying URL 'https://cloud.r-project.org/bin/windows/contrib/3.6/RMySQL_0.10.17.zip'
Content type 'application/zip' length 3195872 bytes (3.0 MB)
downloaded 3.0 MB

package 'DBI' successfully unpacked and MD5 sums checked
package 'RMySQL' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\ajeet\AppData\Local\Temp\RtmpeY4LXs\downloaded_packages
>

Save workspace image? [y/n/c]:
```

We have created a database employee in which there is a table employee_info, which has the following record.



We will use the data which we have mentioned above in our upcoming topics.

Create a connection between R and MySql

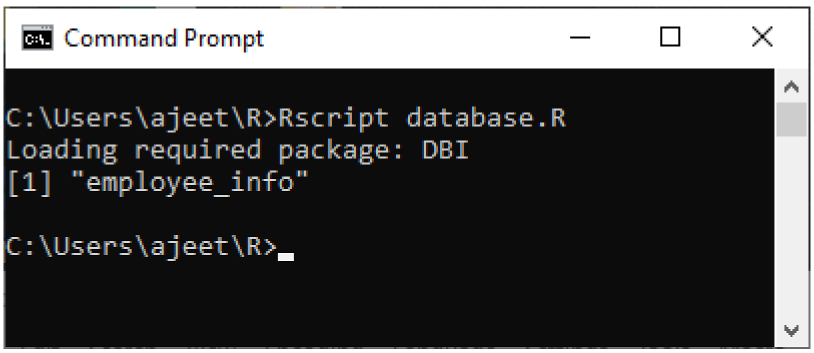
To work with MySql database, it is required to create a connection object between R and the database. For creating a connection, R provides **dbConnect()** function. This function takes the username, password, database name, and host name as input parameters. Let's see an example to understand how the **dbConnect()** function is used to connect with the database.

Example

```
#Loading RMySQL package into R
library("RMySQL")
```

```
# Creating a connection Object to MySQL database.
# Conneting with database named "employee" which we have created befoe with the he
lpof XAMPP server.
mysql_connect = dbConnect(MySQL(), user = 'root', password = '', dbname = 'employee',
  host = 'localhost')

# Listing the tables available in this database.
dbListTables(mysql_connect)
```

Output



R MySQL Commands

In R, we can perform all the SQL commands like insert, delete, update, etc. For performing the query on the database, R provides the dbSendQuery() function. The query is executed in MySQL, and the result set is returned using the R fetch () function. Finally, it is stored in R as a data frame. Let's see the example of each and every SQL command to understand how dbSendQuery() and fetch() functions are used.

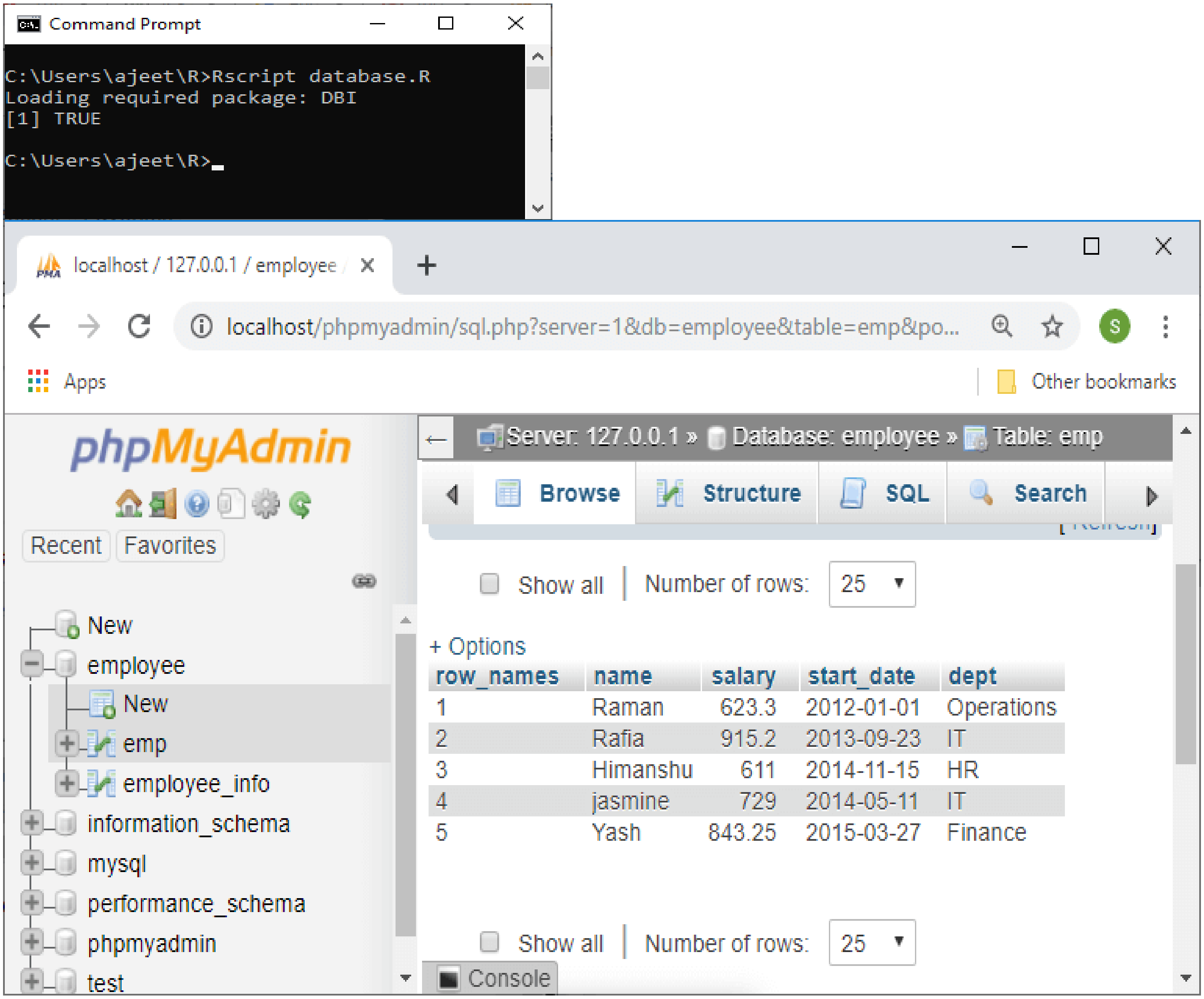
Create Table

R provides an additional function to create a table into the database i.e., dbWriteTable(). This function creates a table in the database; if it does not exist else, it will overwrite the table. This function takes the data frame as an input.

Example

```
#Loading RMySQL package into R
library("RMySQL")
# Creating a connection Object to MySQL database.
# Conneting with database named "employee" which we have created befoe with the helpof XAMPP server.
mysql_connect = dbConnect(MySQL(), user = 'root', password = '', dbname = 'employee',
  host = 'localhost')
#Creating data frame to create a table
emp.data<- data.frame(
name = c("Raman","Rafia","Himanshu","jasmine","Yash"),
salary = c(623.3,915.2,611.0,729.0,843.25),
start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11","2015-03-27")),
dept = c("Operations","IT","HR","IT","Finance"),
stringsAsFactors = FALSE
)
 # All the rows of emp.data are taken inot MySql.
dbWriteTable(mysql_connect, "emp", emp.data[, ], overwrite = TRUE)
```

Output



Select

We can simply select the record from the table with the help of the fetch() and dbSendQuery() function. Let's see an example to understand how to select query works with these two functions.

Example

1. #Loading RMySQL package into R
2. library("RMySQL")
3. # Creating a connection Object to MySQL database.

4. # Conneting with database named "employee" which we have created befoe with the he lpof XAMPP server.
5. mysql_connect = dbConnect(MySQL(), user = 'root', password = '', dbname = 'employee',
6.   host = 'localhost')
7. # selecting the record from employee_info table.
8. record = dbSendQuery(mysql_connect, "select * from employee_info")
9. # Storing the result in a R data frame object. n = 6 is used to fetch first 6 rows.
10. data_frame = fetch(record, n = 6)
11. print(data_frame)

Output

Command Prompt

```
C:\Users\ajeet\R>Rscript database.R
Loading required package: DBI
  id     name salary       date       dept
1  1  Shubham    623   1/1/2012         IT
2  2   Nishka    552  9/23/2012 Operations
3  3   Gunjan    669 11/15/2014         IT
4  4    Sumit    825  5/11/2014         HR
5  5   Arpita    762  3/27/2012    Finance
6  6 Vaishali    882  5/21/2013         IT

C:\Users\ajeet\R>_
```

Select with where clause

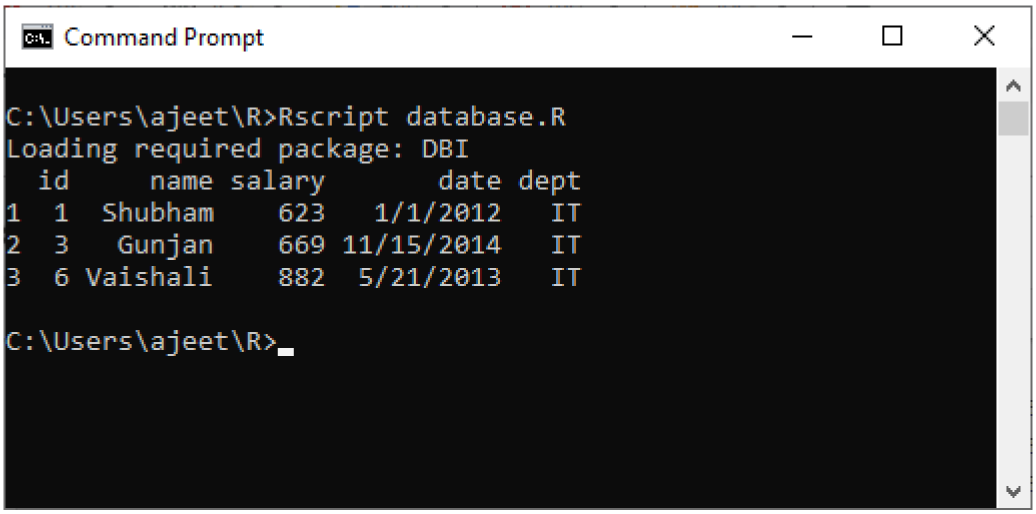
We can select the specific record from the table with the help of the fetch() and dbSendQuery() function. Let's see an example to understand how to select query works with where clause and these two functions.

Example

1. #Loading RMySQL package into R
2. library("RMySQL")
3. # Creating a connection Object to MySQL database.
4. # Conneting with database named "employee" which we have created befoe with the he lpof XAMPP server.

```
mysql_connect = dbConnect(MySQL(), user = 'root', password = '', dbname = 'employee',
  host = 'localhost')
# selecting the specific record from employee_info table.
record = dbSendQuery(mysql_connect, "select * from employee_info where dept='IT'")
# Fetching all the records(with n = -1) and storing it as a data frame.
data_frame = fetch(record, n = -1)
print(data_frame)
```

Output



Insert command

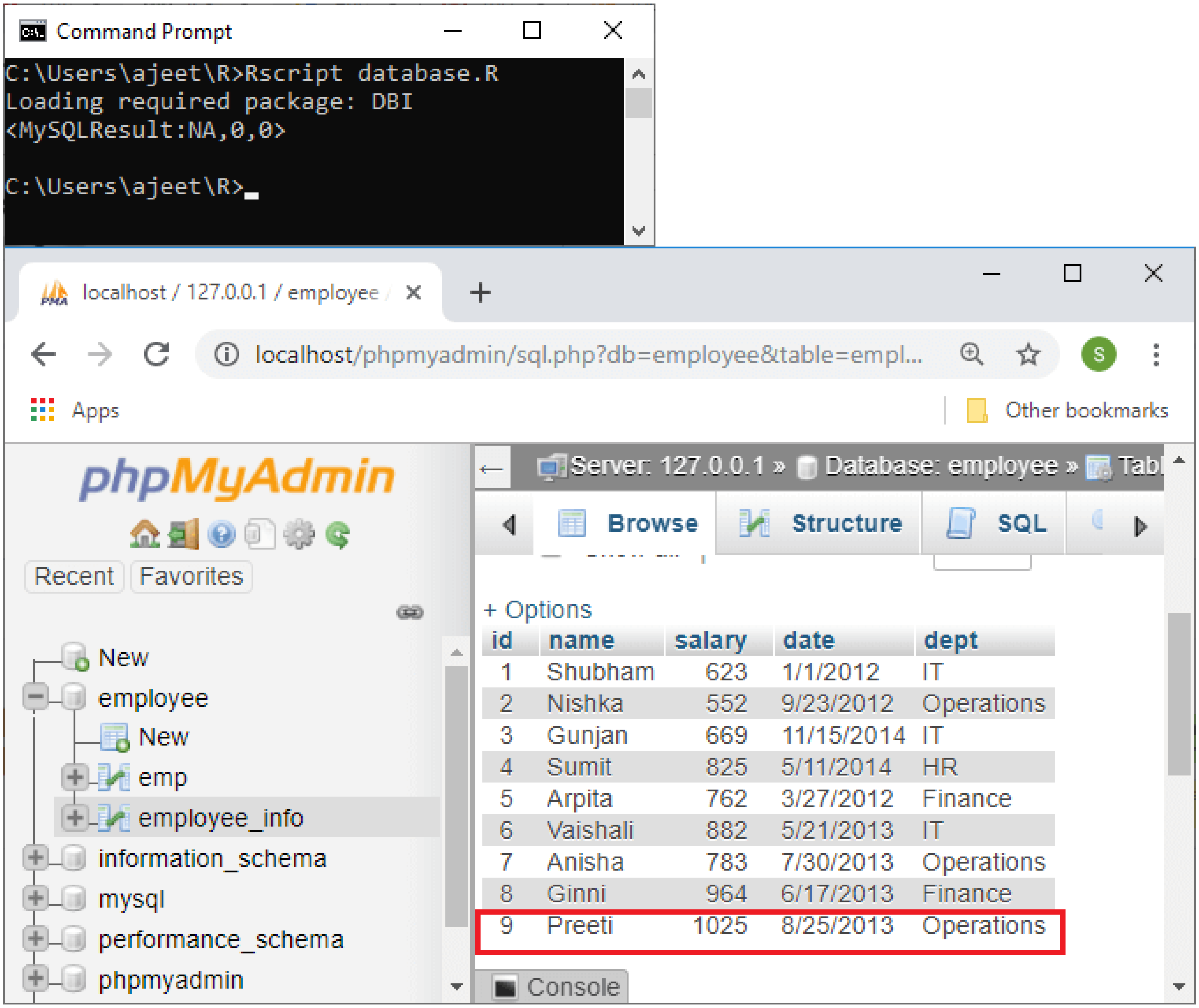
We can insert the data into tables with the help of the familiar method dbSendQuery() function.

Example

```
#Loading RMySQL package into R
library("RMySQL")
# Creating a connection Object to MySQL database.
# Conneting with database named "employee" which we have created befoe with the helpof XAMPP server.
mysql_connect = dbConnect(MySQL(), user = 'root', password = '', dbname = 'employee',
  host = 'localhost')
# Inserting record into employee_info table.
```

8. dbSendQuery(mysql_connect, "insert into employee_info values(9,'Preeti',1025,'8/25/2013','Operations')")

Output



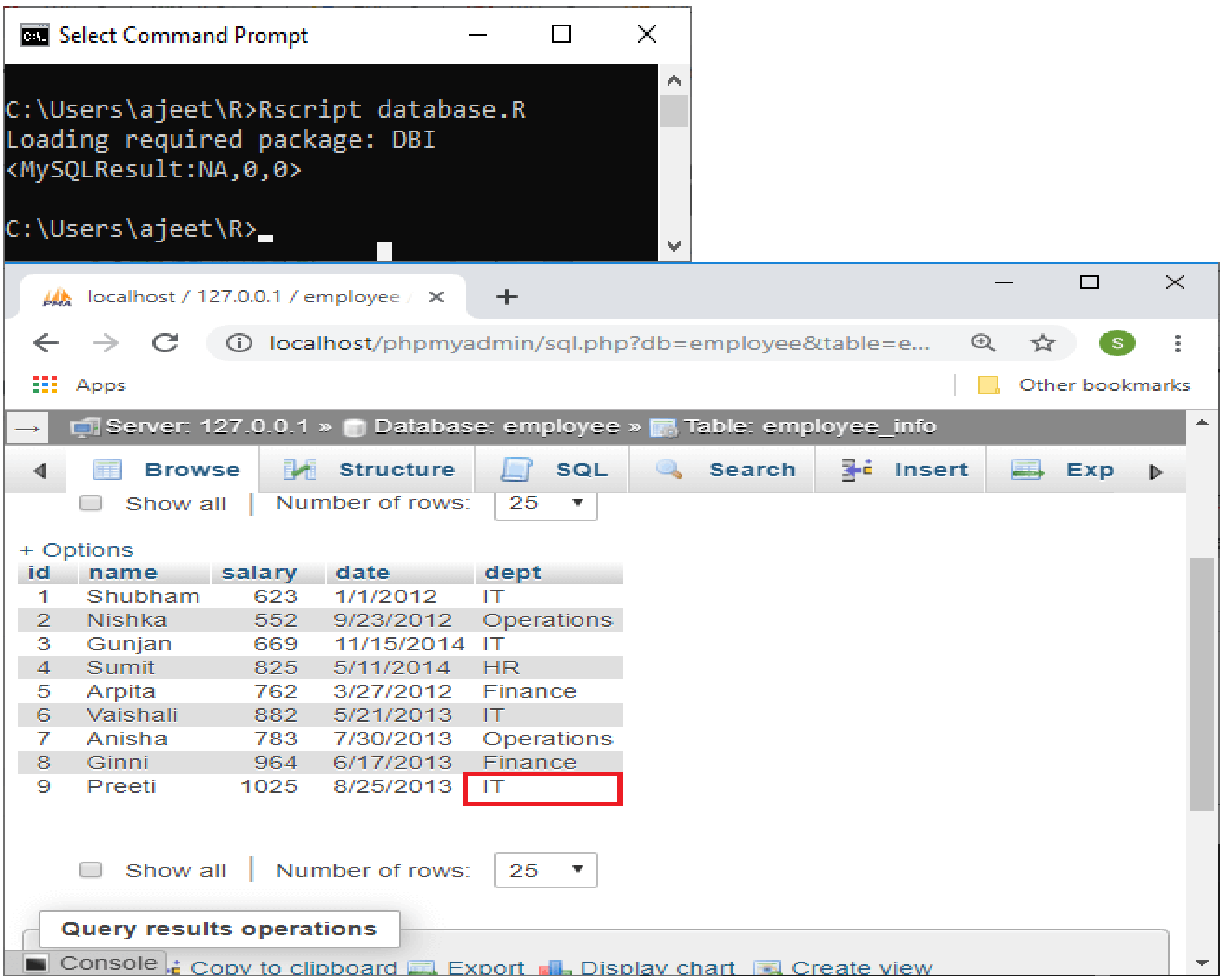
Update command

Updating a record in the table is much easier. For this purpose, we have to pass the update query to the dbSendQuery() function.

Example

```
#Loading RMySQL package into R
library("RMySQL")
# Creating a connection Object to MySQL database.
# Conneting with database named "employee" which we have created befoe with the help of XAMPP server.
mysql_connect = dbConnect(MySQL(), user = 'root', password = '', dbname = 'employee',
  host = 'localhost')
# Updating the record in employee_info table.
dbSendQuery(mysql_connect, "update employee_info set dept='IT' where id=9")
```

Output



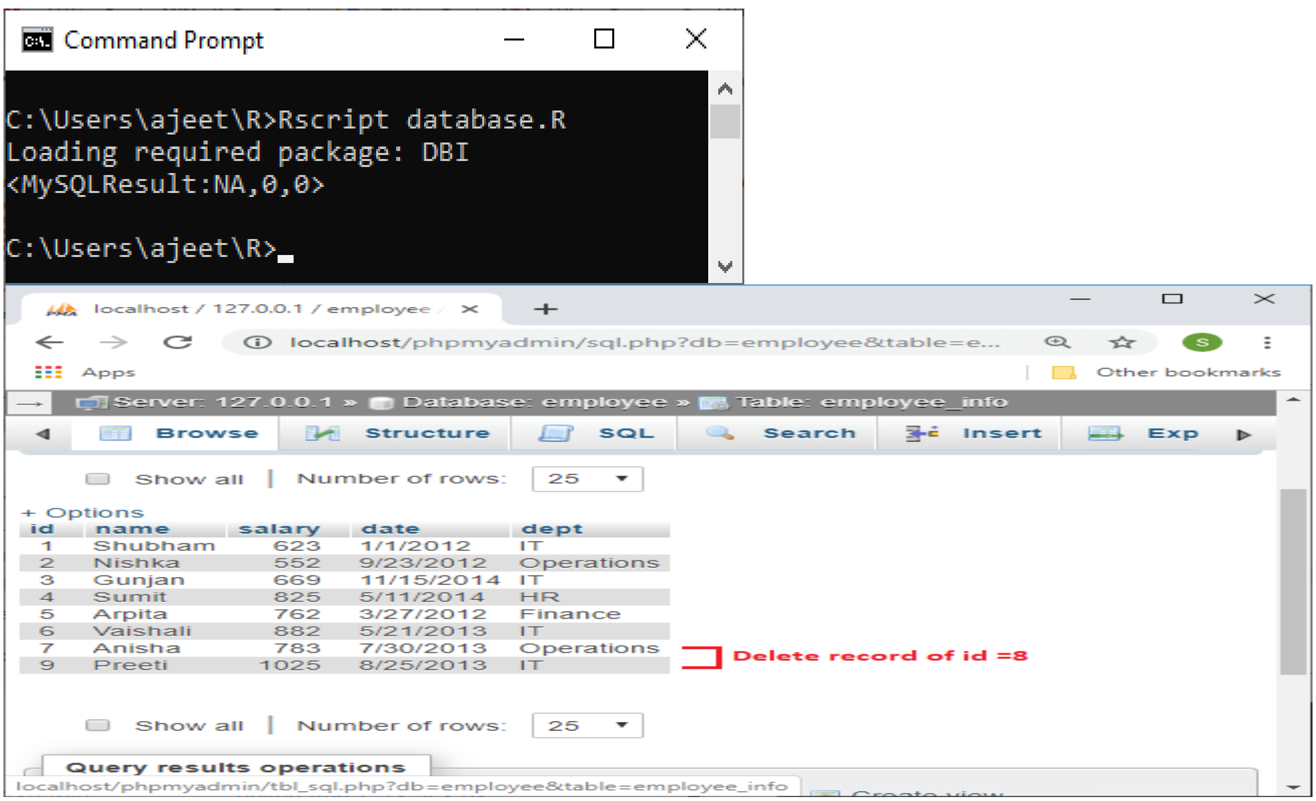
Delete command

Below is an example in which we delete a specific row from the table by passing the delete query in the dbSendQuery() function.

Example

1. #Loading RMySQL package into R
2. library("RMySQL")
3. # Creating a connection Object to MySQL database.
4. # Conneting with database named "employee" which we have created befoe with the he lpof XAMPP server.
5. mysql_connect = dbConnect(MySQL(), user = 'root', password = '', dbname = 'employee',
6.   host = 'localhost')
7. # Deleting the specific record from employee_info table.
8. dbSendQuery(mysql_connect, "delete from employee_info where id=8")

Output



Drop command

Below is an example in which we drop a table from the database by passing the appropriate drop query in the dbSendQuery() function.

Example

1. #Loading RMySQL package into R
2. library("RMySQL")
3. # Creating a connection Object to MySQL database.
4. # Conneting with database named "employee" which we have created befoe with the he lpof XAMPP server.
5. mysql_connect = dbConnect(MySQL(), user = 'root', password = '', dbname = 'employee',
6. host = 'localhost')
7. # Dropping the specific table from the employee database.
8. dbSendQuery(mysql_connect, "drop table if exists emp")

Output