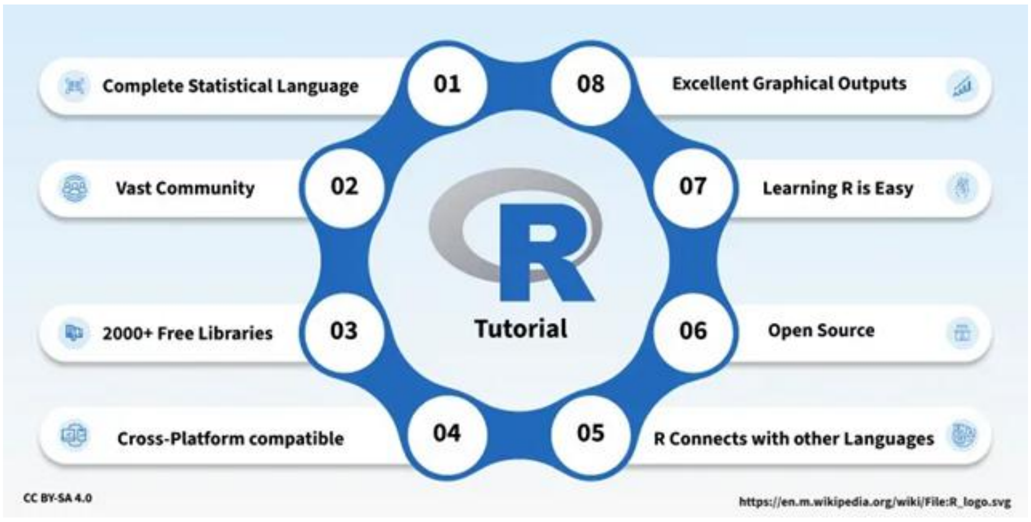
UNIT 1 INTRODUCTION TO R

Introduction - History and overview of R - elements and data structures - Sessions and Functions - Variables - Data Types - Vectors - Scalars - Conclusion - Data Frames - Lists - Matrices - Arrays - Classes - Data input/output - Data storage formats - Subsetting objects - Vectorization.

Introduction - History and overview of R

R is a programming language and software environment that has become the first choice for statistical computing and data analysis. Developed in the early 1990s by Ross Ihaka and Robert Gentleman, R was built to simplify complex data manipulation and create clear, customizable visualizations. Over time, it has gained popularity among statisticians, data scientists and researchers because of its capabilities and the vast array of packages available.


Complete Statistical Language
01
08
Excellent Graphical Outputs
Vast Community
02
07
Learning R is Easy
R
Tutorial
2000+ Free Libraries
03
06
Open Source
Cross-Platform compatible
04
05
R Connects with other Languages
CC BY-SA 4.0
https://en.m.wikipedia.org/wiki/File:R_logo.svg

As data-driven decision-making has grown, R has established itself as an important tool in various industries, including finance and healthcare, due to its ability to handle large datasets and perform in-depth statistical analysis.

Why Choose R Programming?

R is a unique language that offers a wide range of features for data analysis, making it an essential tool for professionals in various fields. Here’s why R is preferred:

- **Free and Open-Source:** R is open to everyone, meaning users can modify, share and distribute their work freely.
- **Designed for Data:** R is built for data analysis, offering a comprehensive set of tools for statistical computing and graphics.

- **Large Package Repository:** The Comprehensive R Archive Network (CRAN) offers thousands of add-on packages for specialized tasks.
- **Cross-Platform Compatibility:** R can work on Windows, Mac and Linux operating systems.
- **Great for Visualization:** With packages like **ggplot2**, R makes it easy to create informative, interactive charts and plots.

Key Features of R

- **Cross-Platform Support:** R works on multiple operating systems, making it versatile for different environments.
- **Interactive Development:** R allows users to interactively experiment with data and see the results immediately.
- **Data Wrangling:** Tools like **dplyr** and **tidyr** help simplify data cleaning and transformation.
- **Statistical Modeling:** R has built-in support for various statistical models like regression, time-series analysis and clustering.
- **Reproducible Research:** With R Markdown, users can combine code, output and narrative in one document, ensuring their analysis is reproducible.

Applications of R

R is used in a variety of fields, including:

- **Data Science and Machine Learning**: R is widely used for data analysis, statistical modeling and machine learning tasks.
- **Finance:** Financial analysts use R for quantitative modeling and risk analysis.
- **Healthcare:** In clinical research, R helps analyze medical data and test hypotheses.
- **Academia:** Researchers and statisticians use R for data analysis and publishing reproducible research.

Data Structures in R Programming

A data structure is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks. Data structures in R programming are tools for holding multiple values.
R’s base data structures are often organized by their dimensionality (1D, 2D or nD) and whether they’re homogeneous (all elements must be of the identical type) or heterogeneous (the elements are often of various types). This gives rise to the six data types which are most frequently utilized in data analysis.

1. Vectors

A vector is an ordered collection of basic data types of a given length. The only key thing here is all the elements of a vector must be of the identical data type e.g homogeneous data structures. Vectors are one-dimensional data structures.

Example:
X = c(1, 3, 5, 7, 8)

print(X)
Output:
[1] 1 3 5 7 8

2. Lists
A list is a generic object consisting of an ordered collection of objects. Lists are heterogeneous data structures. These are also one-dimensional data structures. A list can be a list of vectors, list of matrices, a list of characters and a list of functions and so on.

Example:

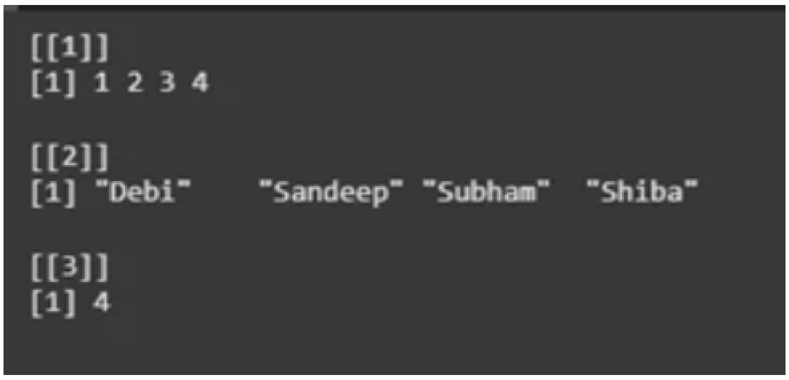
mpId = c(1, 2, 3, 4)

empName = c("Debi", "Sandeep", "Subham", "Shiba")

numberOfEmp = 4

empList = list(empId, empName, numberOfEmp)

print(empList)



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

[[2]]
[1] "Debi"    "Sandeep" "Subham"  "Shiba"

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

3. Data Frames

Data frames are generic data objects of R which are used to store the tabular data. Data frames are the foremost popular data objects in R programming because we are comfortable in seeing the data within the tabular form. They are two-dimensional, heterogeneous data structures. These are lists of vectors of equal lengths.

Data frames have the following constraints placed upon them:

- A data-frame must have column names and every row should have a unique name.
- Each column must have the identical number of items.
- Each item in a single column must be of the same data type.
- Different columns may have different data types.

To create a data frame we use the data.frame() function.

Example:

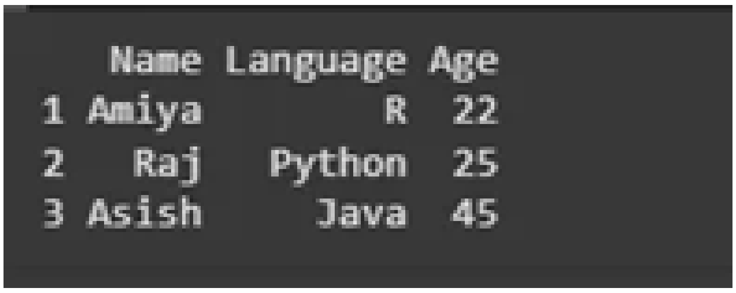
Name = c("Amiya", "Raj", "Asish")

Language = c("R", "Python", "Java")

Age = c(22, 25, 45)

df = data.frame(Name, Language, Age)

print(df)



```
   Name Language Age
1 Amiya        R  22
2   Raj   Python  25
3 Asish     Java  45
```

4. Matrices

A matrix is a rectangular arrangement of numbers in rows and columns. In a matrix, as we know rows are the ones that run horizontally and columns are the ones that run vertically. Matrices are two-dimensional, homogeneous data structures.

Example:

A = matrix(

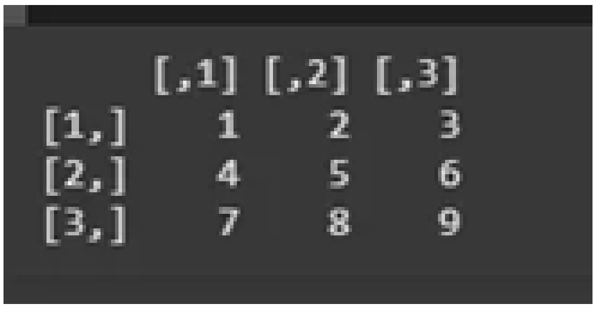
    c(1, 2, 3, 4, 5, 6, 7, 8, 9),

    nrow = 3, ncol = 3,

    byrow = **TRUE**

)

print(A)



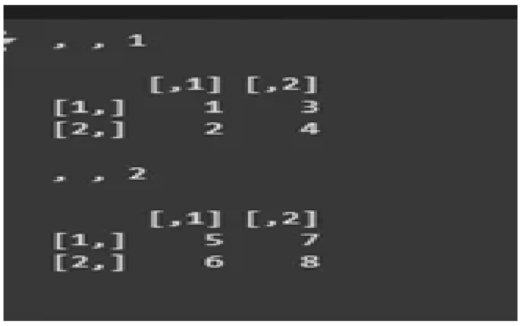
```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

5. Arrays

Arrays are the R data objects which store the data in more than two dimensions. Arrays are n-dimensional data structures. For example, if we create an array of dimensions (2, 3, 3) then it creates 3 rectangular matrices each with 2 rows and 3 columns. They are homogeneous data structures.

Example:

```
A = array(
   c(1, 2, 3, 4, 5, 6, 7, 8),
   dim = c(2, 2, 2)
)
print(A)
```



```
, , 1

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

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

6. Factors

Factors are the data objects which are used to categorize the data and store it as levels. They are useful for storing categorical data. They can store both strings and integers. They are useful to categorize unique values in columns like ("TRUE" or "FALSE") or ("MALE" or "FEMALE"), etc.. They are useful in data analysis for statistical modeling.

Example:

```
# Creating factor using factor()
fac = factor(c("Male", "Female", "Male",
           "Male", "Female", "Male", "Female"))
print(fac)
```

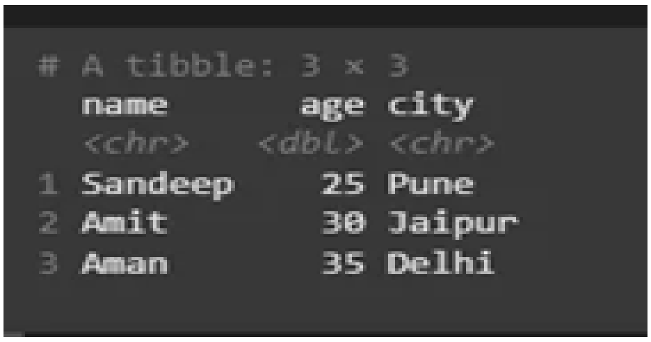
Output:

[1] Male Female Male Male Female Male Female
Levels: Female Male

7. Tibbles

Tibbles are an enhanced version of data frames in R, part of the tidyverse. They offer improved printing, stricter column types, consistent subsetting behavior and allow variables to be referred to as objects. Tibbles provide a modern, user-friendly approach to tabular data in R.

```
library(tibble)


my_data <- tibble(
  name = c("Sandeep", "Amit", "Aman"),
  age = c(25, 30, 35),
  city = c("Pune", "Jaipur", "Delhi")
)


print(my_data)
```



```
# A tibble: 3 × 3
  name      age city
  <chr>   <dbl> <chr>
1 Sandeep    25 Pune
2 Amit       30 Jaipur
3 Aman       35 Delhi
```

Functions in R

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

Syntax:

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

```
Function name=function(ar1,ar2,……)
{
Function body
}
```

Function Components:

The different parts of a function are –

Function Name − This is the actual name of the function. It is stored in R environment as an object with this name.

Arguments − An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.

Function Body − The function body contains a collection of statements that defines what the function does.

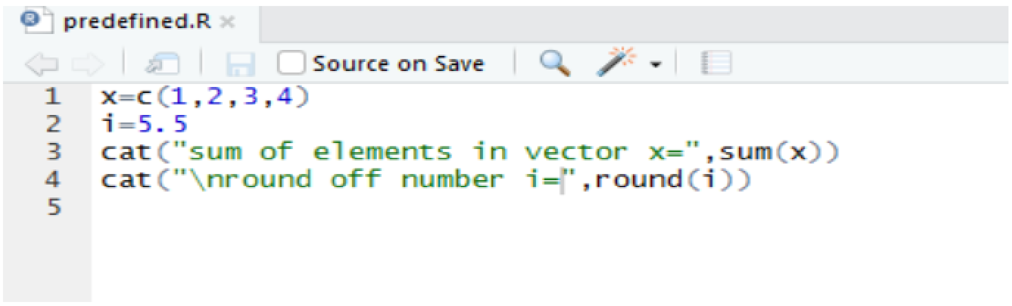
Return Value − The return value of a function is the last expression in the function body to be evaluated

Built-in Functions

R has many in-built functions which can be directly called in the program without defining them first.

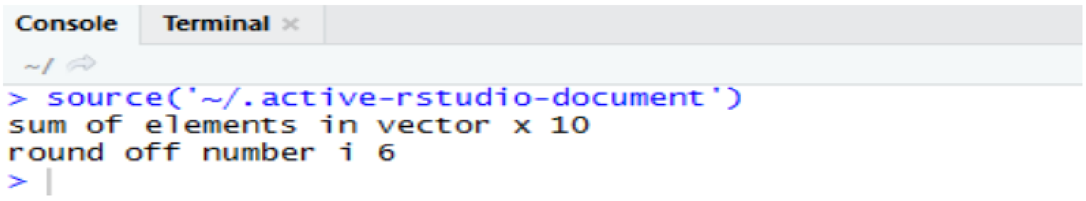
Eg: sum.seq,abs,round .

Example:



```
x=c(1,2,3,4)
i=5.5
cat("sum of elements in vector x=",sum(x))
cat("\nround off number i=",round(i))
```

Output:



```
> source('~/.active-rstudio-document')
sum of elements in vector x 10
round off number i 6
> |
```

Other Built-in Functions in R

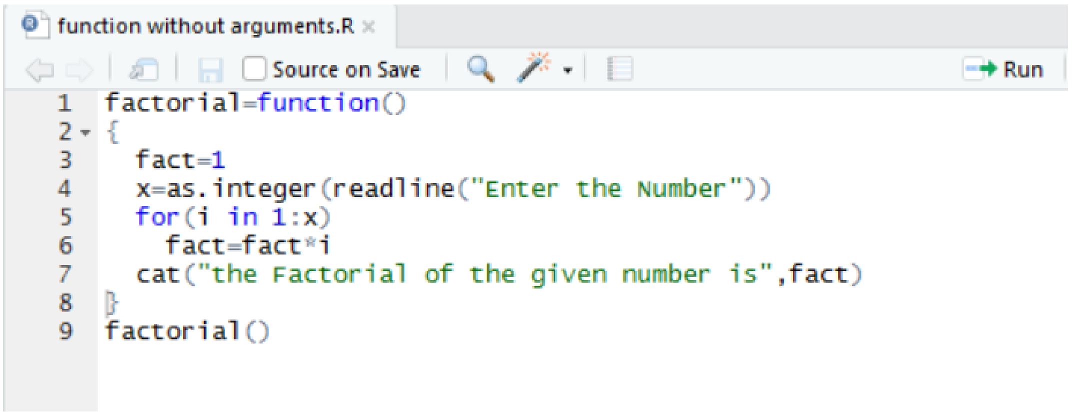
Category	**Function**
Mathematical Functions	abs(), sqrt(), round(), exp(), log(), cos(), sin(), tan()
Statistical Functions	mean(), median(), cor(), var()
Data Manipulation Functions	unique(), subset(), aggregate(), order()
File Input/Output Functions	read.csv(), write.csv(), read.table(), write.table()

User Defined Function:

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

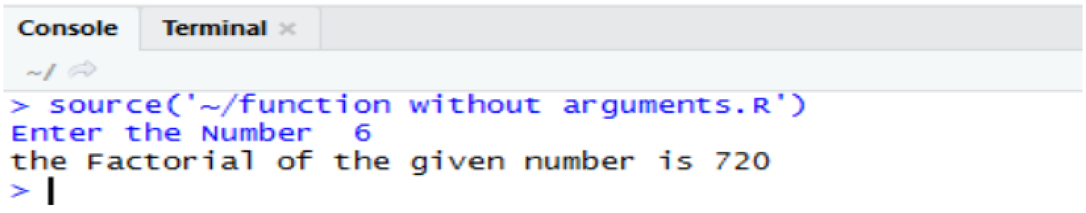
Function without Arguments: Here the function does not receive any arguments.

Example:



```r
factorial=function()
{
  fact=1
  x=as.integer(readline("Enter the Number"))
  for(i in 1:x)
    fact=fact*i
  cat("the Factorial of the given number is",fact)
}
factorial()
```

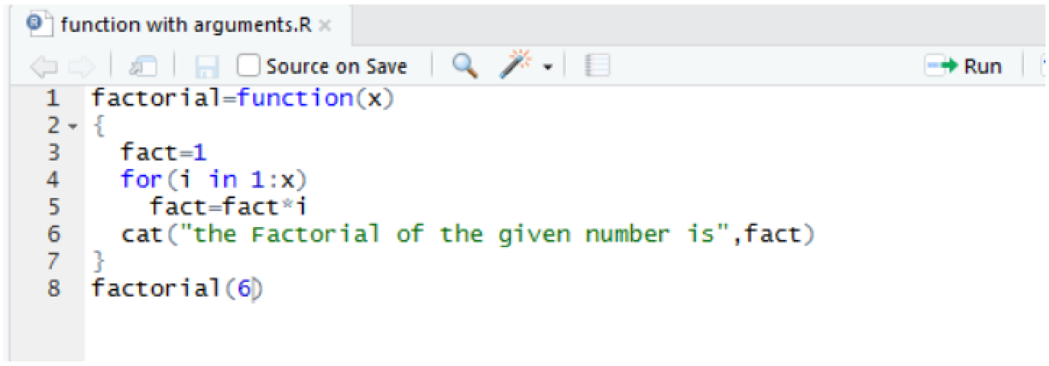
Output:



```
> source('~/function without arguments.R')
Enter the Number  6
the Factorial of the given number is 720
> 
```

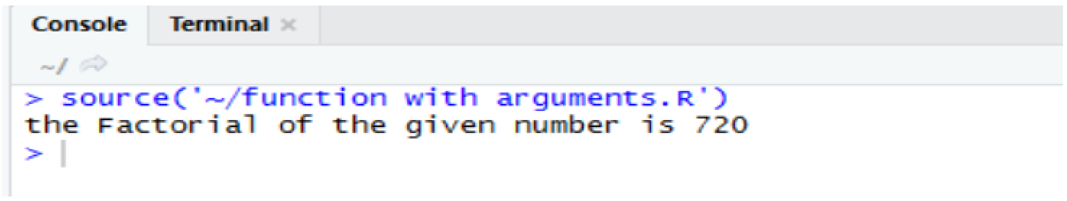
Function with Arguments:

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



```
factorial=function(x)
{
  fact=1
  for(i in 1:x)
    fact=fact*i
  cat("the Factorial of the given number is",fact)
}
factorial(6)
```

Output:

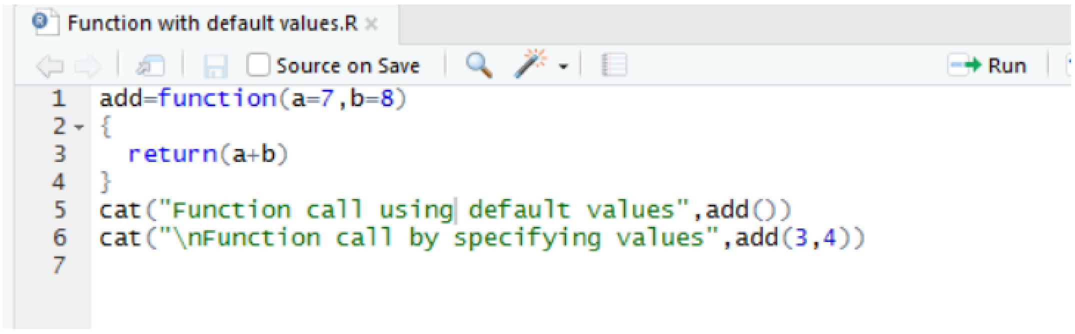


```
~/
> source('~/function with arguments.R')
the Factorial of the given number is 720
>
```

Function with Default Argument

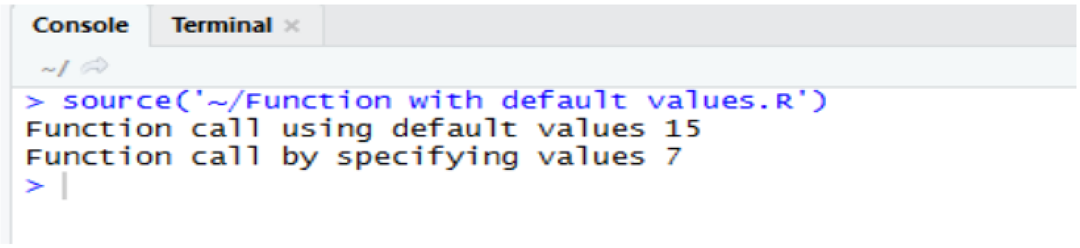
We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non default result.

Example:



```
add=function(a=7,b=8)
{
  return(a+b)
}
cat("Function call using default values",add())
cat("\nFunction call by specifying values",add(3,4))
```

Output:



```
~/
> source('~/Function with default values.R')
Function call using default values 15
Function call by specifying values 7
>
```

LOOPING FUNCTIONS IN R

The for, while loops can often be replaced by looping functions:

lapply:

Definition: Loop over a list and evaluate a function on each element

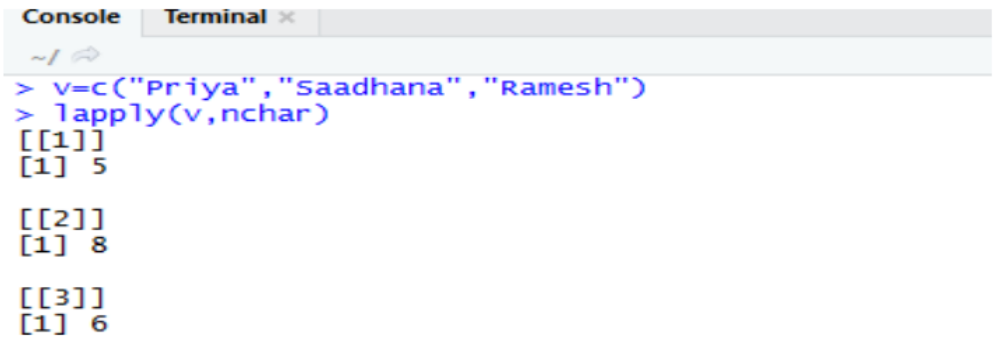
Syntax :

lapply(X, FUN, ...)

X =List

FUN=A

Function …. =other arguments



```
> v=c("Priya","Saadhana","Ramesh")
> lapply(v,nchar)
[[1]]
[1] 5

[[2]]
[1] 8

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

sapply:

Definition : same as lapply but try to simplify the result .

Syntax : lapply(X, FUN, ...)

X

=Li

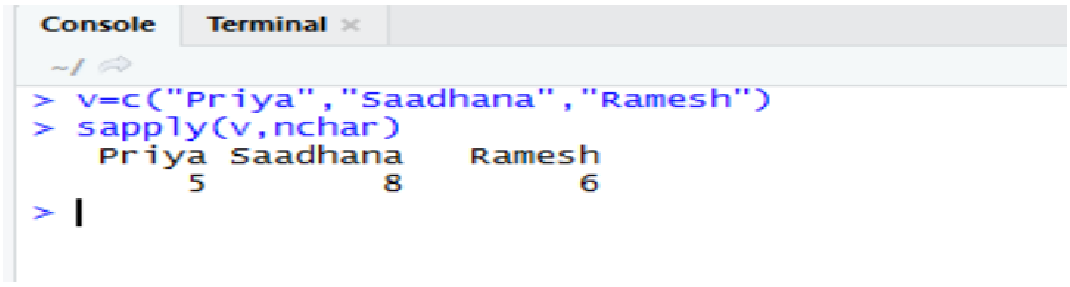
st FUN=A

Function

…. =other arguments

Example:

Example:



```
> v=c("Priya","Saadhana","Ramesh")
> sapply(v,nchar)
   Priya Saadhana   Ramesh
       5        8        6
> 
```

apply:

Definition: apply a function over the margins of an array

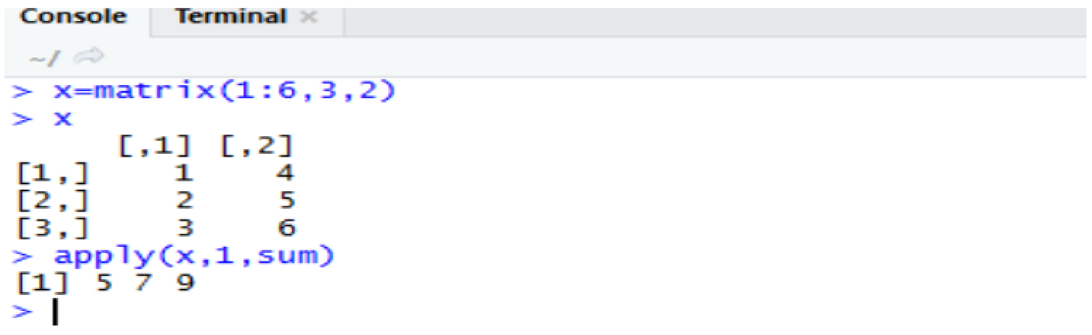
Syntax: apply(X, MARGIN, FUN, ...)

X =An Array

MARGIN =Integer vector indicating which margins should be "retained".

FUN =Function to be applied

. . . = other arguments to be passed to FUN



```
> x=matrix(1:6,3,2)
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> apply(x,1,sum)
[1]  5 7  9
> 
```

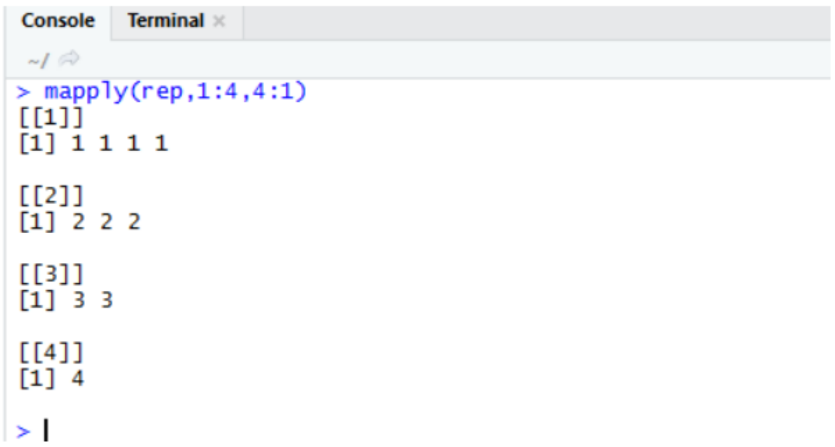
mapply:

Definition: Multivariate version of lapply

Syntax: mapply (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)

FUN	: the function to be applied
. .	: arguments to apply over
MoreArgs	: a list of other arguments to
SIMPLIFY	: logical; whether the result should be simplified to a vector or matrix.

Example:



```
> mapply(rep,1:4,4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4

> 
```

Creating Variables in R

Variables are containers for storing data values.

R does not have a command for declaring a variable. A variable is created the moment you first assign a value to it. To assign a value to a variable, use the <- sign. To output (or print) the variable value, just type the variable name:

Example

```
name <- "John"
age <- 40

name   # output "John"
age    # output 40
```

From the example above, name and age are **variables**, while "John" and 40 are **values**.

Multiple Variables

R allows you to assign the same value to multiple variables in one line:

Example

```
# Assign the same value to multiple variables in one line
var1 <- var2 <- var3 <- "Orange"

# Print variable values
var1
var2
var3
```

R Data Types

Data types in R define the kind of values that variables can hold. Choosing the right data type helps optimize memory usage and computation. Unlike some languages, R does not require explicit data type declarations while variables can change their type dynamically during execution.

R Programming language has the following basic R-data types and the following table shows the data type and the values that each data type can take.

Basic Data Types	**Values**	**Examples**
Numeric	Set of all real numbers	"numeric_value <- 3.14"
Integer	Set of all integers, Z	"integer_value <- 42L"
Logical	TRUE and FALSE	"logical_value <- TRUE"
Complex	Set of complex numbers	"complex_value <- 1 + 2i"

Basic Data Types	Values	Examples
Character	"a", "b", "c", ..., "@", "#", "$", ..., "1", "2", ...etc	"character_value <- "Hello Geeks"
raw	as.raw()	"single_raw <- as.raw(255)"

1. Numeric Data type in R

Decimal values are called numeric in R. It is the default R data type for numbers in R. If we assign a decimal value to a variable x as follows, x will be of numeric type.
Real numbers with a decimal point are represented using this data type in R. It uses a format for double-precision floating-point numbers to represent numerical values.

```
x = 5.6

print(class(x))

print(typeof(x))
```

2. Integer Data type in R

R supports integer data types which are the set of all integers. we can create as well as convert a value into an integer type using the **as.integer()** function.
we can also use the capital 'L' notation as a suffix to denote that a particular value is of the integer R data type.

```
x = as.integer(5)

print(class(x))

print(typeof(x))

y = 5L

print(class(y))

print(typeof(y))
```

Output

```
[1] "integer"

[1] "integer"

[1] "integer"

[1] "integer"
```

3. Logical Data type in R

R has logical data types that take either a value of **true** or **false**. A logical value is often created via a comparison between variables.
Boolean values, which have two possible values, are represented by this R data type: FALSE or TRUE

```
x = 4

y = 3

z = x > y

print(z)

print(class(z))

print(typeof(z))
```

Output

```
[1] TRUE

[1] "logical"

[1] "logical"
```

4. Complex Data type in R

R supports complex data types that are set of all the complex numbers. The complex data type is to store numbers with an imaginary component.

```
x = 4 + 3i

print(class(x))

print(typeof(x))
```

Output

```
[1] "complex"

[1] "complex"
```

5. Character Data type in R

R supports character data types where we have all the alphabets and special characters. It stores character values or strings. Strings in R can contain alphabets, numbers, and symbols. The easiest way to denote that a value is of character type in R data type is to wrap the value inside single or double inverted commas.

```
char = "Geeksforgeeks"
```

```
print(class(char))

print(typeof(char))
```

Output

```
[1] "character"

[1] "character"
```

6. Raw data type in R

To save and work with data at the byte level in R, use the raw data type. By displaying a series of unprocessed bytes, it enables low-level operations on binary data. Here are some speculative data on R's raw data types:

```
x <- as.raw(c(0x1, 0x2, 0x3, 0x4, 0x5))

print(x)
```

Output

```
[1] 01 02 03 04 05
```

Find Data Type of an Object in R

To find the data type of an object we have to use **class()** function. The syntax for doing that is we need to pass the object as an argument to the function **class()** to find the data type of an object.

Syntax

class(object)

Example

```
print(class(TRUE))

print(class(3L))

print(class(10.5))

print(class(1+2i))

print(class("12-04-2020"))
```

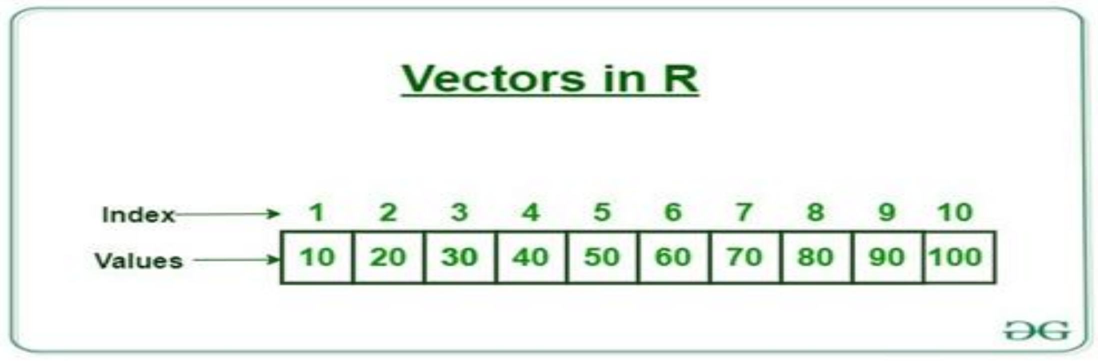
Output

```
[1] "logical"

[1] "integer"

[1] "numeric"

[1] "complex"
```

```
[1] "character"
```

R-Vectors

R Vectors are the same as the arrays in R language which are used to hold multiple data values of the same type. One major key point is that in R Programming Language the indexing of the vector will start from '1' and not from '0'. We can create numeric vectors and character vectors as well.



1. Creating a vector in R

A vector is a basic data structure that represents a one-dimensional array. to create a array we use the **"c"** function which the most common method use in R Programming Language. We can also use **seq()** function or use **colons ":"** also as shown in the example.

```
X<- c(61, 4, 21, 67, 89, 2)

cat('using c function', X, '\n')

Y<- seq(1, 10, length.out = 5)

cat('using seq() function', Y, '\n')

Z<- 2:7

cat('using colon', Z)
```

Output:

```
using c function 61 4 21 67 89 2
using seq() function 1 3.25 5.5 7.75 10
using colon 2 3 4 5 6 7
```

2. Types of R vectors

Vectors are of different types which are used in R. Following are some of the types of vectors:

2.1 Numeric vectors

Numeric vectors are those which contain numeric values such as integer, float, etc. The L suffix in R is used to specify that a number is an integer and not a numeric (floating-point) value.

```
v1 <- c(4, 5, 6, 7)

typeof(v1)

v2 <- c(1L, 4L, 2L, 5L)

typeof(v2)
```

Output:

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

2.2 Character vectors

Character vectors in R contain alphanumeric values and special characters. In R, when a vector contains elements of mixed types (like characters and numbers), R automatically coerces the entire vector to a single type. Since characters are more general than numbers in R, the vector is coerced to a character vector.

```
v1 <- c('geeks', '2', 'hello', 57)

typeof(v1)
```

Output:

```
[1] "character"
```

2.3 Logical vectors

Logical vectors in R contain Boolean values such as TRUE, FALSE and NA for Null values. In R, NA is a special value used to represent missing or undefined data. When used in a logical vector, NA is treated as a logical value because it is specifically designed to work with logical vectors and other types of data.

```
v1 <- c(TRUE, FALSE, TRUE, NA)

typeof(v1)
```

Output:

```
[1] "logical"
```

3. Length of R vector

In R, the length of a vector is determined by the number of elements it contains. we can use the length() function to retrieve the length of a vector.

```
x <- c(1, 2, 3, 4, 5)

length(x)
```

```
y <- c("apple", "banana", "cherry")

length(y)


z <- c(TRUE, FALSE, TRUE, TRUE)

length(z)
```

Output:

```
> length(x)
[1] 5

> length(y)
[1] 3

> length(z)
[1] 4
```

4. Accessing R vector elements

Accessing elements in a vector is the process of performing operation on an individual element of a vector. There are many ways through which we can access the elements of the vector. The most common is using the '[]', symbol.

```
X <- c(2, 5, 18, 1, 12)

cat('Using Subscript operator', X[2], '\n')


Y <- c(4, 8, 2, 1, 17)

cat('Using combine() function', Y[c(4, 1)], '\n')
```

Output:

```
Using Subscript operator 5
Using combine() function 1 4
```

5. Modifying a R vector

Modification of a Vector is the process of applying some operation on an individual element of a vector to change its value in the vector. There are different ways through which we can modify a vector:

```
X <- c(2, 7, 9, 7, 8, 2)


X[3] <- 1

X[2] <- 9

cat('subscript operator', X, '\n')
```

```
X[1:5] <- 0

cat('Logical indexing', X, '\n')


X <- X[c(3, 2, 1)]

cat('combine() function', X)
```

Output:

```
subscript operator 2 9 1 7 8 2
Logical indexing 0 0 0 0 0 2
combine() function 0 0 0
```

6. Deleting a R vector

Deletion of a Vector is the process of deleting all of the elements of the vector. This can be done by assigning it to a NULL value.

```
M <- c(8, 10, 2, 5)

M <- NULL

print(cat('Output vector', M, "\f"))
```

Output:

```
Output vector NULL
```

7. Sorting elements of a R Vector

sort() function is used with the help of which we can sort the values in ascending or descending order.

```
X <- c(8, 2, 7, 1, 11, 2)

A <- sort(X)

cat('ascending order', A, '\n')

B <- sort(X, decreasing = TRUE)

cat('descending order', B)
```

Output:

```
ascending order 1 2 2 7 8 11
descending order 11 8 7 2 2 1
```

R Data Frames

Data Frames are data displayed in a format as a table.

Data Frames can have different types of data inside it. While the first column can be character, the second and third can be numeric or logical. However, each column should have the same type of data.

Use the data.frame() function to create a data frame:

Example

```
# Create a data frame
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

# Print the data frame
Data_Frame
```

Summarize the Data

Use the summary() function to summarize the data from a Data Frame:

Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)
Data_Frame
summary(Data_Frame)
```

Access Items

We can use single brackets [ ], double brackets [[ ]] or $ to access columns from a data frame:

Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

Data_Frame[1]

Data_Frame[["Training"]]
```

```
Data_Frame$Training
```

Add Rows

Use the rbind() function to add new rows in a Data Frame:

Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

# Add a new row
New_row_DF <- rbind(Data_Frame, c("Strength", 110, 110))

# Print the new row
New_row_DF
```

Add Columns

Use the cbind() function to add new columns in a Data Frame:

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

# Add a new column
New_col_DF <- cbind(Data_Frame, Steps = c(1000, 6000, 2000))

# Print the new column
New_col_DF
```

Remove Rows and Columns

Use the c() function to remove rows and columns in a Data Frame:

Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

# Remove the first row and column
```

```
Data_Frame_New <- Data_Frame[-c(1), -c(1)]

# Print the new data frame
Data_Frame_New
```

Amount of Rows and Columns

Use the dim() function to find the amount of rows and columns in a Data Frame:

Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

dim(Data_Frame)
```

You can also use the ncol() function to find the number of columns and nrow() to find the number of rows:

Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

ncol(Data_Frame)
nrow(Data_Frame)
```

Data Frame Length

Use the length() function to find the number of columns in a Data Frame (similar to ncol()):

Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

length(Data_Frame)
```

Combining Data Frames

Use the rbind() function to combine two or more data frames in R vertically:

Example

```
Data_Frame1 <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

Data_Frame2 <- data.frame (
  Training = c("Stamina", "Stamina", "Strength"),
  Pulse = c(140, 150, 160),
  Duration = c(30, 30, 20)
)

New_Data_Frame <- rbind(Data_Frame1, Data_Frame2)
New_Data_Frame
```

And use the cbind() function to combine two or more data frames in R horizontally:

Example

```
Data_Frame3 <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

Data_Frame4 <- data.frame (
  Steps = c(3000, 6000, 2000),
  Calories = c(300, 400, 300)
)

New_Data_Frame1 <- cbind(Data_Frame3, Data_Frame4)
New_Data_Frame1
```

R List

A list in R programming is a generic object consisting of an ordered collection of objects. Lists are **one-dimensional**, **heterogeneous** data structures. The list can be a list of vectors, a list of matrices, a list of characters, a list of functions, and so on. A list in R is created with the use of the list() function.

R allows accessing elements of an R list with the use of the index value. In R, the indexing of a list starts with 1 instead of 0.

1. Creating a List

To create a List in R you need to use the function called "**list()**". We want to build a list of employees with the details. So for this, we want attributes such as ID, employee name, and the number of employees.

Example:

```
empId = c(1, 2, 3, 4)

empName = c("Debi", "Sandeep", "Subham", "Shiba")

numberOfEmp = 4

 empList = list(empId, empName, numberOfEmp)

 print(empList)
```

2. Naming List Components

Naming list components make it easier to access them.

Example:

```
my_named_list <- list(name = "Sudheer", age = 25, city = "Delhi")

print(my_named_list)
```

3. Accessing R List Components

We can access components of an R list in two ways.

3.1. Access components by names:

All the components of a list can be named and we can use those names to access the components of the R list using the dollar command.

Example:

```
empId = c(1, 2, 3, 4)

empName = c("Debi", "Sandeep", "Subham", "Shiba")

numberOfEmp = 4


empList = list(

  "ID" = empId,

  "Names" = empName,

  "Total Staff" = numberOfEmp

  )
```

```
print(empList)


cat("Accessing name components using $ command\n")
print(empList$Names)
```

3.2. Access components by indices:

We can also access the components of the R list using indices. To access the top-level components of a R list we have to use a double slicing operator "[[ ]]" which is two square brackets and if we want to access the lower or inner-level components of a R list we have to use another square bracket "[ ]" along with the double slicing operator "[[ ]]".

Example:

```
empId = c(1, 2, 3, 4)
empName = c("Debi", "Sandeep", "Subham", "Shiba")
numberOfEmp = 4


empList = list(
  "ID" = empId,
  "Names" = empName,
  "Total Staff" = numberOfEmp
)
print(empList)


cat("Accessing name components using indices\n")
print(empList[[2]])


cat("Accessing Sandeep from name using indices\n")
print(empList[[2]][2])


cat("Accessing 4 from ID using indices\n")
print(empList[[1]][4])
```

3.3. Concatenation of lists

Two R lists can be concatenated using the concatenation function. So, when we want to concatenate two lists we have to use the concatenation operator.

Syntax

list = c(list, list1)
list = the original list
list1 = the new list

Example:

```
empId = c(1, 2, 3, 4)

empName = c("Debi", "Sandeep", "Subham", "Shiba")

numberOfEmp = 4


empList = list(

  "ID" = empId,

  "Names" = empName,

  "Total Staff" = numberOfEmp

)

cat("Before concatenation of the new list\n")

print(empList)


empAge = c(34, 23, 18, 45)

empList = c(empName, empAge)


cat("After concatenation of the new list\n")

print(empList)
```

3.4. Adding Item to List

To add an item to the end of list, we can use append() function.

```
my_numbers = c(1,5,6,3)

append(my_numbers, 45)

my_numbers
```

3.5. Deleting Components of a List

To delete components of a R list, first of all, we need to access those components and then insert a negative sign before those components. It indicates that we had to delete that component.

Example:

```
empId = c(1, 2, 3, 4)

empName = c("Debi", "Sandeep", "Subham", "Shiba")

numberOfEmp = 4


empList = list(

  "ID" = empId,

  "Names" = empName,

  "Total Staff" = numberOfEmp

)

cat("Before deletion the list is\n")

print(empList)

cat("After Deleting Total staff components\n")

print(empList[-3])

cat("After Deleting sandeep from name\n")

print(empList[[2]][-2])
```

3.6. Merging list

We can merge the R list by placing all the lists into a single list.

```
lst1 <- list(1,2,3)

lst2 <- list("Sun","Mon","Tue")

new_list <- c(lst1, lst2)

print(new_list)
```

Matrices in R

Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout. We use matrices containing numeric elements to be used in mathematical calculations.

Syntax: The basic syntax for creating a matrix in R is –

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

• data is the input vector which becomes the data elements of the matrix.

• nrow is the number of rows to be created.

• ncol is the number of columns to be created.

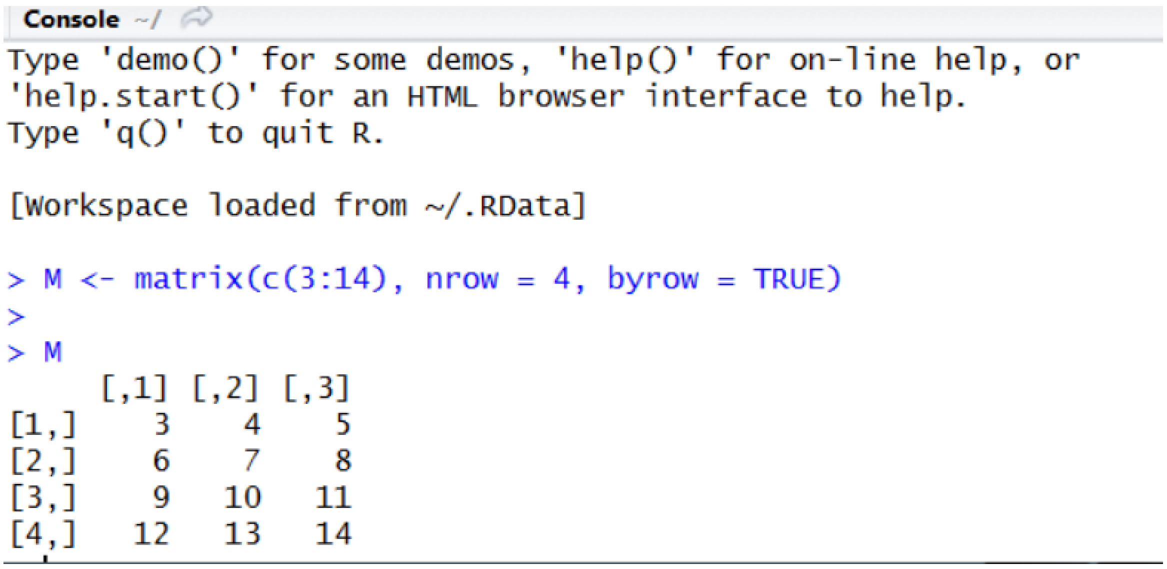
• byrow is a logical clue. If TRUE then the input vector elements are arranged by row.

• dimname is the names assigned to the rows and columns.

Matrix Creation:

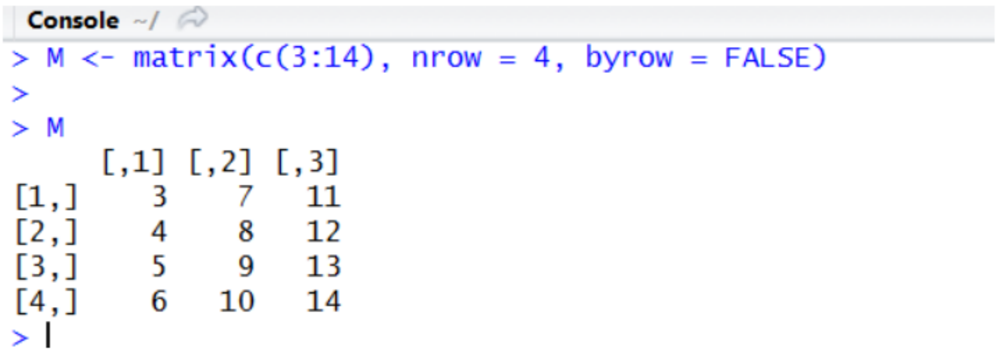
(i)Arrange elements sequentially by row.

Example:



```
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.

[Workspace loaded from ~/.RData]

> M <- matrix(c(3:14), nrow = 4, byrow = TRUE)
>
> M
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    7    8
[3,]    9   10   11
[4,]   12   13   14
```

(ii) Arrange elements sequentially by coloumn.



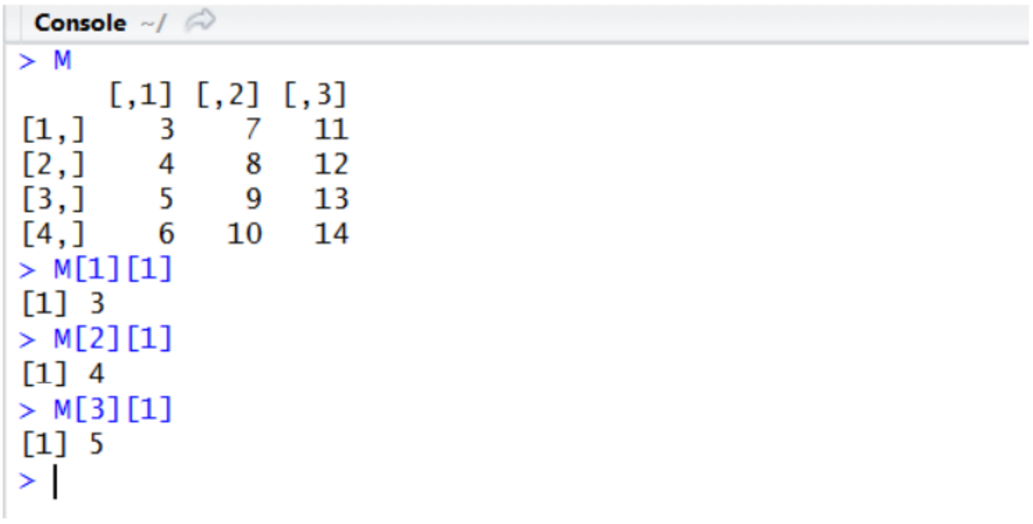
Console ~/

```
> M <- matrix(c(3:14), nrow = 4, byrow = FALSE)
>
> M
     [,1] [,2] [,3]
[1,]    3    7   11
[2,]    4    8   12
[3,]    5    9   13
[4,]    6   10   14
> 
```

Accessing Elements of Matrix:

Elements of a matrix can be accessed by using the column and row indexof the element

Example:

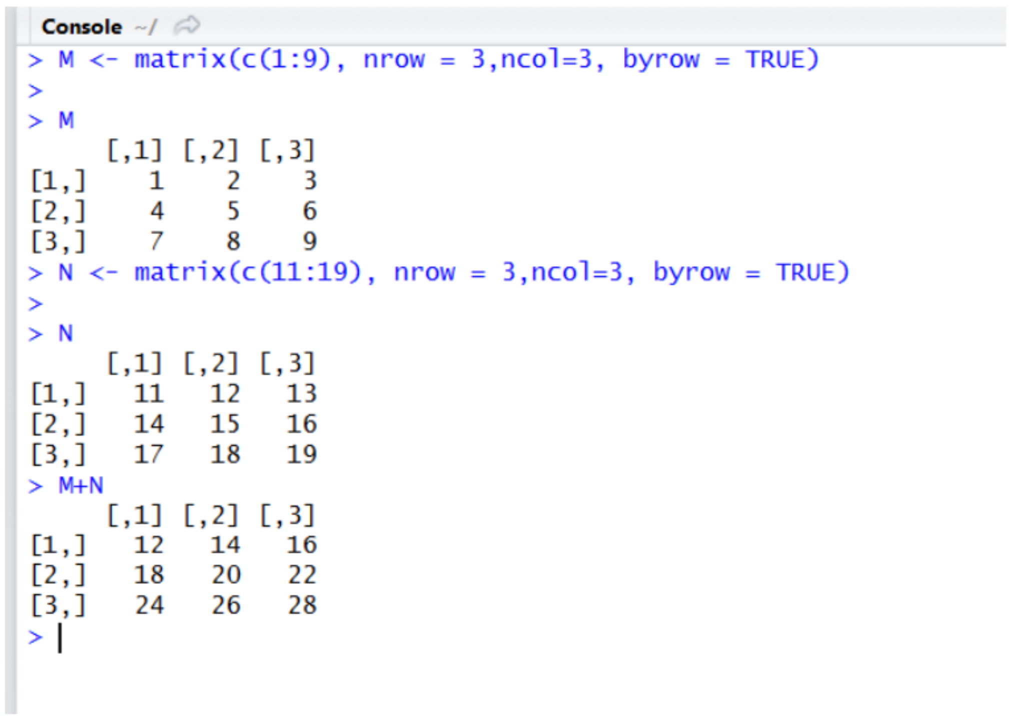


Console ~/

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

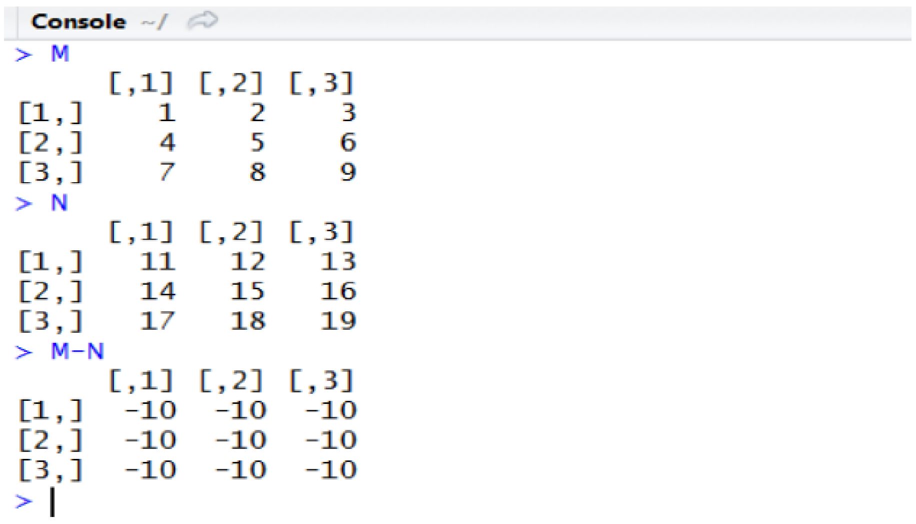
Matrix Operations: Various mathematical operations are performed on the matrices using the Roperators. The result of the operation is also a matrix.The dimensions (number of rows and columns) should be same for the matrices involved in the operation.

Matrix Addition:



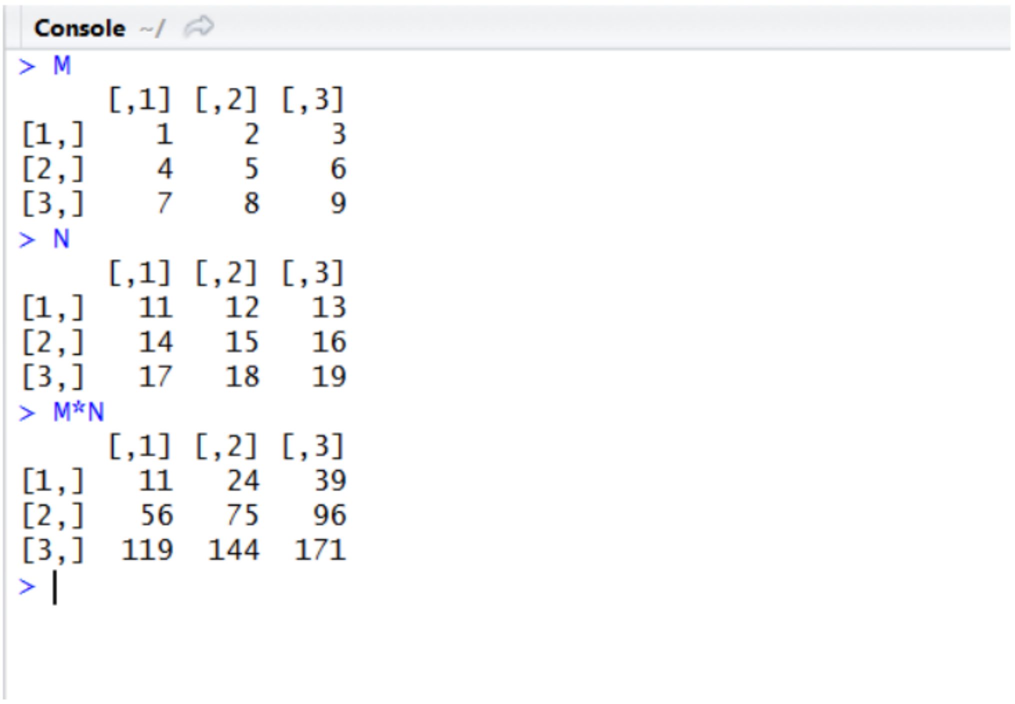
```
> M <- matrix(c(1:9), nrow = 3,ncol=3, byrow = TRUE)
>
> M
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> N <- matrix(c(11:19), nrow = 3,ncol=3, byrow = TRUE)
>
> N
     [,1] [,2] [,3]
[1,]   11   12   13
[2,]   14   15   16
[3,]   17   18   19
> M+N
     [,1] [,2] [,3]
[1,]   12   14   16
[2,]   18   20   22
[3,]   24   26   28
>
```

Matrix Subtraction



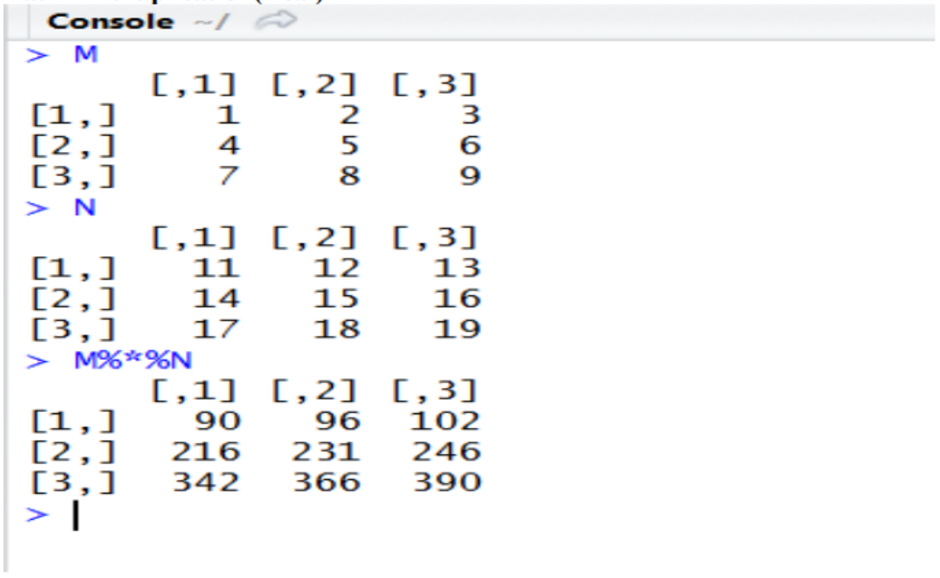
```
> M
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> N
     [,1] [,2] [,3]
[1,]   11   12   13
[2,]   14   15   16
[3,]   17   18   19
> M-N
     [,1] [,2] [,3]
[1,]  -10  -10  -10
[2,]  -10  -10  -10
[3,]  -10  -10  -10
>
```

Matrix Multiplication(Elementwise)



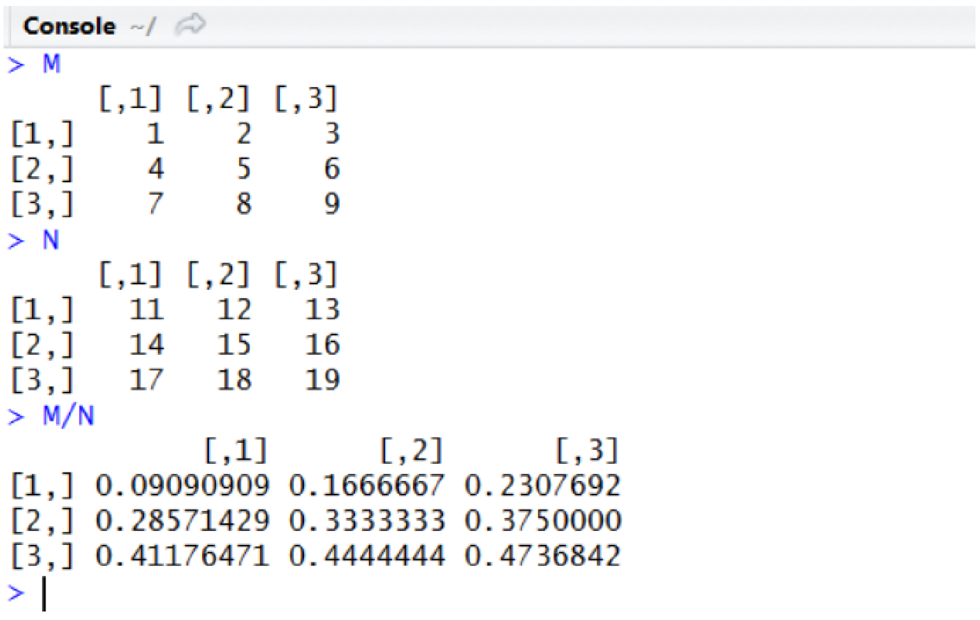
```
> M
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> N
     [,1] [,2] [,3]
[1,]   11   12   13
[2,]   14   15   16
[3,]   17   18   19
> M*N
     [,1] [,2] [,3]
[1,]   11   24   39
[2,]   56   75   96
[3,]  119  144  171
>
```

Matrix Multiplication(Real)



```
> M
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> N
     [,1] [,2] [,3]
[1,]   11   12   13
[2,]   14   15   16
[3,]   17   18   19
> M%*%N
     [,1] [,2] [,3]
[1,]   90   96  102
[2,]  216  231  246
[3,]  342  366  390
>
```

Matrix Division:


Console ~/

```
> M
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> N
     [,1] [,2] [,3]
[1,]   11   12   13
[2,]   14   15   16
[3,]   17   18   19
> M/N
           [,1]      [,2]      [,3]
[1,] 0.09090909 0.1666667 0.2307692
[2,] 0.28571429 0.3333333 0.3750000
[3,] 0.41176471 0.4444444 0.4736842
> |
```

R - Array

Arrays are important data storage structures defined by a fixed number of dimensions. Arrays are used for the allocation of space at contiguous memory locations.

In R Programming Language Uni-dimensional arrays are called vectors with the length being their only dimension. Two-dimensional arrays are called matrices, consisting of fixed numbers of rows and columns. R Arrays consist of all elements of the same data type. Vectors are supplied as input to the function and then create an array based on the number of dimensions.

1. Creating an Array

An R array can be created with the use of array() the function. A list of elements is passed to the array() functions along with the dimensions as required.

Syntax:

array(data, dim = (nrow, ncol, nmat), dimnames=names)

where:

- **nrow:** Number of rows
- **ncol:** Number of columns
- **nmat:** Number of matrices of dimensions nrow * ncol
- **dimnames:** Default value = NULL.

Otherwise, a list has to be specified which has a name for each component of the dimension. Each component is either a null or a vector of length equal to the dim value of that corresponding dimension.

1.1 Uni-Dimensional Array

A vector is a uni-dimensional array, which is specified by a single dimension, length. A Vector can be created using c() function. A list of values is passed to the c() function to create a vector.

vec1 <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)

print (vec1)

cat ("Length of vector : ", length(vec1))

Output:

[1] 1 2 3 4 5 6 7 8 9
Length of vector : 9

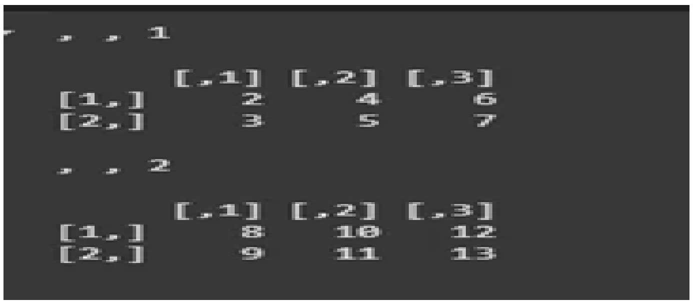
1.2. Multi-Dimensional Array

A two-dimensional matrix is an array specified by a fixed number of rows and columns, each containing the same data type. A matrix is created by using **array()** function to which the values and the dimensions are passed.

arr = array(2:13, dim = c(2, 3, 2))

print(arr)

Output:



```
, , 1

     [,1] [,2] [,3]
[1,]    2    4    6
[2,]    3    5    7

, , 2

     [,1] [,2] [,3]
[1,]    8   10   12
[2,]    9   11   13
```

Naming of Arrays

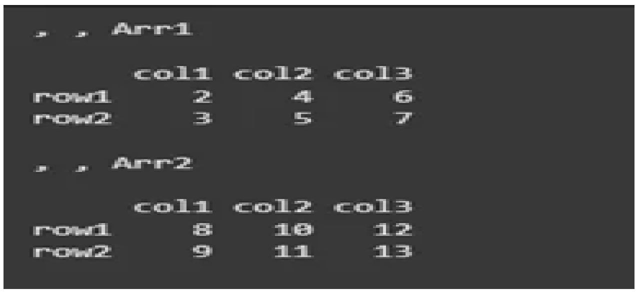
The row names, column names and matrices names are specified as a vector of the number of rows, number of columns and number of matrices respectively. By default, the rows, columns and matrices are named by their index values.

row_names <- c("row1", "row2")

col_names <- c("col1", "col2", "col3")

mat_names <- c("Arr1", "Arr2")

```
arr = array(2:14, dim = c(2, 3, 2),

            dimnames = list(row_names,

                            col_names, mat_names))

print (arr)
```



```
, , Arr1

     col1 col2 col3
row1    2    4    6
row2    3    5    7

, , Arr2

     col1 col2 col3
row1    8   10   12
row2    9   11   13
```

Accessing arrays

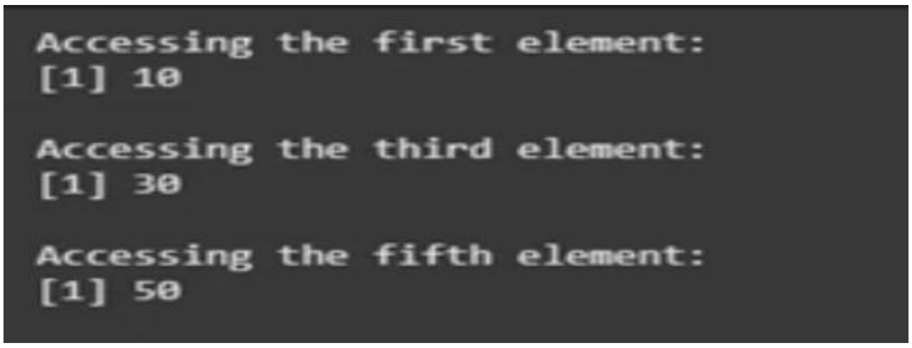
The R arrays can be accessed by using indices for different dimensions separated by commas. Different components can be specified by any combination of elements' names or positions.

4.1. Accessing Uni-Dimensional Array

The elements can be accessed by using indexes of the corresponding elements.

```
uni_array <- c(10, 20, 30, 40, 50)


cat("Accessing the first element:\n")

print(uni_array[1])


cat("\nAccessing the third element:\n")

print(uni_array[3])


cat("\nAccessing the fifth element:\n")

print(uni_array[5])
```

Output:

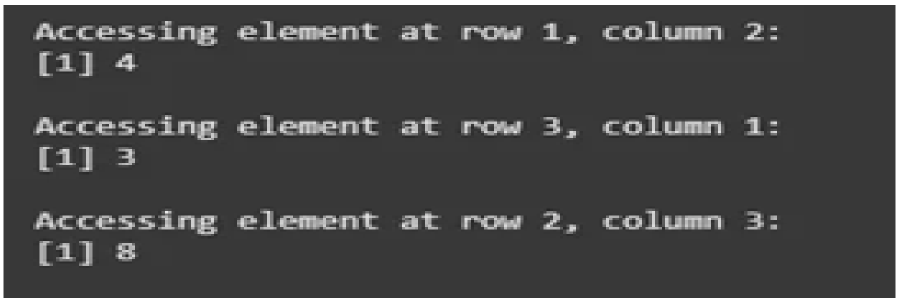


```
Accessing the first element:
[1] 10

Accessing the third element:
[1] 30

Accessing the fifth element:
[1] 50
```

Accessing Multi-Dimensional Array

The elements can be accessed by using indexes of the corresponding elements.

```
multi_array <- array(1:9, dim = c(3, 3))


cat("Accessing element at row 1, column 2:\n")
print(multi_array[1, 2])


cat("\nAccessing element at row 3, column 1:\n")
print(multi_array[3, 1])


cat("\nAccessing element at row 2, column 3:\n")
print(multi_array[2, 3])
```



```
Accessing element at row 1, column 2:
[1] 4

Accessing element at row 3, column 1:
[1] 3

Accessing element at row 2, column 3:
[1] 8
```

Classes in R Programming

Classes and Objects are core concepts in Object-Oriented Programming (OOP), modeled after real-world entities. In R, everything is treated as an object. An object is a data structure with defined attributes and methods. A class is a blueprint that defines a set of properties and methods shared by all objects of that type.

R has a unique three-class system: S3, S4, and Reference Classes. Each of these class systems has distinct characteristics and is used to define and manage objects and their methods effectively.

1. S3 Class

S3 is the most widely used OOP system in R, but it lacks a formal definition and structure. An object of this type can be created simply by adding an attribute to it.

Example:

```
movieList <- list(name = "Iron man", leadActor = "Robert Downey Jr")


class(movieList) <- "movie"


movieList
```

Output:

```
$name
[1] "Iron man"

$leadActor
[1] "Robert Downey Jr"

attr(,"class")
[1] "movie"
```

2. S4 Class

Programmers familiar with languages like C++ or Java may find S3 significantly different from their typical concept of classes, as it lacks the structure usually associated with classes. S4 improves upon S3 by providing a more formal definition for objects and offering a clear structure for managing them.

Example:

As shown in the example, **setClass()** is used to define a class and **new()** is used to create the objects.

```
library(methods)

setClass("movies", slots=list(name="character", leadActor = "character"))

movieList <- new("movies", name="Iron man", leadActor = "Robert Downey Jr")

movieList
```

Output:

```
An object of class "movies"
Slot "name":
```

```
[1] "Iron man"
Slot "leadActor":
[1] "Robert Downey Jr"
```

3. Reference Class

Reference Classes are an improvement over S4 Classes. In this system, methods are associated with the classes, making them more similar to object-oriented classes in other languages. Defining a Reference Class is similar to defining an S4 class. Instead of ***setClass()****, we use* ***setRefClass()****, and instead of "****slots****," we use "****fields****."*

Example:

```
library(methods)


moviess <- setRefClass("moviess",

                    fields = list(

                      name = "character",

                      leadActor = "character",

                      rating = "numeric"

                    ))


movieList <- new("moviess", name = "Iron Man", leadActor = "Robert Downey Jr", rating =
7)


movieList
```

Output:

```
Reference class object of class "moviess"
Field "name":
[1] "Iron Man"
Field "leadActor":
[1] "Robert Downey Jr"
Field "rating":
[1] 7
```

Data In and Out

There are a few principal functions reading data into R.

- *read.table(), read.csv(): for reading tabular data*

- *readLines( ) : for reading lines of a text file*
- *source () : for reading in R code files (inverse of dump)*
- *dget () : for reading in R code files (inverse of dput)*
- *load () : for reading in saved workspaces*
- *unserialize () : for reading single R objects in binary form*

There are analogous functions for writing data to files

- *write.table() : for writing tabular data to text files (i.e. CSV) or connections*
- *writeLines() : for writing character data line-by-line to a file or connection*
- *dump() : for dumping a textual representation of multiple R objects*
- *dput() : for outputting a textual representation of an R object*
- *save() : for saving an arbitrary number of R objects in binary format (possiblycompressed) to a file.*
- *Serialize() :for converting an R object into a binary format for outputting to a connection (orfile).*

Subsetting in R Programming

In R Programming Language, subsetting allows the user to access elements from an object. It takes out a portion from the object based on the condition provided. There are 4 ways of subsetting in R programming. Each of the methods depends on the usability of the user and the type of object. For example, if there is a dataframe with many columns such as states, country, and population and suppose the user wants to extract states from it, then subsetting is used to do this operation. In this article, let us discuss the implementation of different types of subsetting in R programming.

R - subsetting

Method 1: Subsetting in R Using [ ] Operator

Using the '[ ]' operator, elements of vectors and observations from data frames can be accessed. To neglect some indexes, '-' is used to access all other indexes of vector or data frame.

Example 1:

In this example, let us create a vector and perform subsetting using the [ ] operator.

```
# Create vector

x <- 1:15

# Print vector

cat("Original vector: ", x, "\n")
```

```
# Subsetting vector
cat("First 5 values of vector: ", x[1:5], "\n")
cat("Without values present at index 1, 2 and 3: ",
                      x[-c(1, 2, 3)], "\n")
```

Output:

```
Original vector:  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
First 5 values of vector:  1 2 3 4 5
Without values present at index 1, 2 and 3:  4 5 6 7 8 9 10 11 12 13 14 15
```

Method 2: Subsetting in R Using [[ ]] Operator

[[ ]] operator is used for subsetting of list-objects. This operator is the same as [ ] operator but the only difference is that [[ ]] selects only one element whereas [ ] operator can select more than 1 element in a single command.

Example 1: In this example, let us create a list and select the elements using [[]] operator.

```
# Create list
ls <- list(a = 1, b = 2, c = 10, d = 20)


# Print list
cat("Original List: \n")
print(ls)


# Select first element of list
cat("First element of list: ", ls[[1]], "\n")
```

Output:

```
Original List:
$a
[1] 1
$b
[1] 2
$c
[1] 10
$d
```

```
[1] 20
First element of list:  1
```

Method 3: Subsetting in R Using $ Operator

$ operator can be used for lists and data frames in R. Unlike [ ] operator, it selects only a single observation at a time. It can be used to access an element in named list or a column in data frame. $ operator is only applicable for recursive objects or list-like objects.

Example 1: In this example, let us create a named list and access the elements using $ operator

```
# Create list
ls <- list(a = 1, b = 2, c = "Hello", d = "GFG")


# Print list
cat("Original list:\n")
print(ls)


# Print "GFG" using $ operator
cat("Using $ operator:\n")
print(ls$d)
```

Output:

```
Original list:
$a
[1] 1
$b
[1] 2
$c
[1] "Hello"
$d
[1] "GFG"
Using $ operator:
[1] "GFG"
```

Vectorized IF Statement in R

In R Language vectorized operations are a powerful feature that allows you to apply functions or operations over entire vectors at once, rather than looping through each element individually. The ifelse() function is a primary tool for creating vectorized if statements in R Programming Language.

Introduction to Vectorized ifelse()

The ifelse() function in R is used to create vectorized conditional statements. It applies a test condition across each element of a vector and returns one value for TRUE elements and another for FALSE elements. Vectorization in R means that operations are applied element-wise across entire vectors or arrays without the need for explicit loops. This leads to more concise and efficient code.

ifelse(test, yes, no)

Where,

- ***test:*** *A logical vector.*
- ***yes:*** *Values returned for TRUE elements.*
- ***no:*** *Values returned for FALSE elements.*
- **Example of Vectorized ifelse()**
- Let's see how ifelse() works with a simple example.

```
# Define a numeric vector
numbers <- c(10, 15, 20, 25, 30)

# Apply vectorized if statement
result <- ifelse(numbers > 20, "Above 20", "20 or Below")

# Print the result
print(result)
```

- **Output:**

```
[1] "20 or Below" "20 or Below" "20 or Below" "Above 20"    "Above 20"
```

Vectorized ifelse() with More Complex Conditions

You can use more complex conditions within ifelse() to handle a variety of scenarios.

```
# Define a numeric vector

numbers <- c(5, 10, 15, 20, 25)


# Apply vectorized if statement with multiple conditions

result <- ifelse(numbers < 10, "Less than 10",

        ifelse(numbers <= 20, "Between 10 and 20", "Greater than 20"))
```

```
# Print the result
print(result)
```

Output:

```
[1] "Less than 10"      "Between 10 and 20" "Between 10 and 20" "Between 10 and 20"
[5] "Greater than 20"
```

Conclusion

The ifelse() function in R is a powerful tool for creating vectorized if statements, allowing you to apply conditional logic across entire vectors efficiently. It’s particularly useful for simple, element-wise checks and transformations. For more complex operations, alternative approaches like loops or lapply() might be more appropriate. Understanding when and how to use ifelse() is key to writing efficient and effective R code.