Unit-1

Introduction to R- Programming

Introduction:

- R is a **programming language** and software environment for **statistical analysis**,**graphics representation and reporting**.R was created by **Ross Ihaka** and **Robert Gentleman** at the University of Auckland, NewZealand, and is currently developed by the R Development Core Team.
- R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac.
- This programming language was named R,based on the first letter of first name of the two R authors (Robert Gentleman and Ross Ihaka),and partly a play on the name of theBell Labs Language S.
- R is the most popular data analytics tool as it is open-source, flexible, offers multiple packages and has a huge community.

Why Do We Need Analytics?

Before an answer to above question, let us see some of the problems and their solutions in R in multiple domains.

Banking:

Large amount of customer data is generated every day in Banks. While dealing with millions of customers on regular basis, it becomes hard to track their mortgages.

Solution:

R builds a custom model that maintains the loans provided to every individual customer which helps us to decide the amount to be paid by the customer over time.

Insurance:

Insurance extensively depends on forecasting. It is difficult to decide which policy to accept or reject.

Solution:

By using the continuous credit report as input, we can create a model in R that will not only assess risk appetite but also make a predictive forecast as well.

Healthcare:

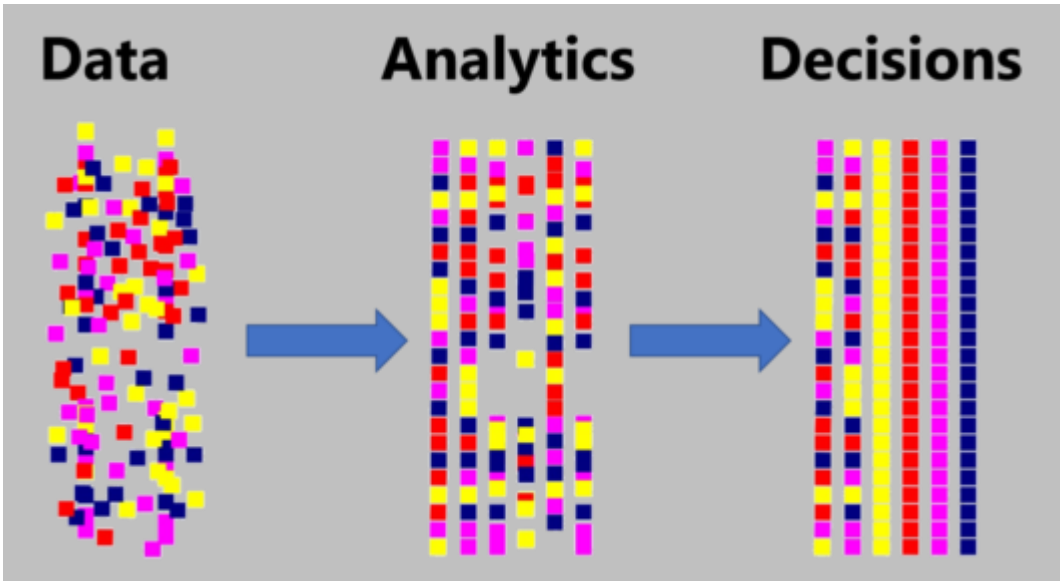
Every year millions of people are admitted in hospital and billions are spent annually just in the admission process.

Solution:

Given the patient history and medical history, a predictive model can be built to identify who is at risk for hospitalization and to what extent the medical equipment should be scaled.

What is Business Analytics?

Business analytics is a process of examining large sets of data and achieving hidden patterns, correlations and other insights. It basically helps you understand all the data that you have gathered, be it organizational data, market or product research data or any other kind of data. It becomes easy for you to make better decisions, better products, better marketing strategies etc. Refer to the below image for better understanding:



If you look at the above figure, your data in the first image is scattered. Now, if you want something specific such as a particular record in a database, it becomes cumbersome. To simplify this, you need analysis. With analysis, it becomes easy to strike a correlation between the data. Once you have established what to do, it becomes quite easy for you to make decisions such as, which path you want to follow or in terms of business analytics, which path will lead to the betterment of your organization.

But you can't expect people in the chain above to always understand the raw data that you are providing them after analytics. So to overcome this gap, we have a concept of *data visualization*.

Data visualization: Data visualization is a visual access to huge amounts of data that you have generated after analytics. The human mind processes visual images and visual graphics are better than compare to raw data. It's always easy for us to understand a pie chart or a bar graph compare to raw numbers. Now you may be wondering how you can achieve this data visualization from the data you have already analyzed.

There are various tools available in the market for Data Visualization:R, Power BI, Spark, Qlikview etc.

Why R?

R is a programming and statistical language.

R is used for data Analysis and Visualization.

R is simple and easy to learn, read and write.

R is an example of a FLOSS (Free Libre and Open Source Software) where one can freely distribute copies of this software, read its source code, modify it, etc.

Who uses R?

- The Consumer Financial Protection Bureau uses R for data analysis
- Statisticians at John Deere use R for time series modeling and geospatial analysis in a reliable and reproducible way.
- Bank of America uses R for reporting.
- R is part of technology stack behind Foursquare's famed recommendation engine.
- ANZ, the fourth largest bank in Australia, using R for credit risk analysis.
- Google uses R to predict Economic Activity.
- Mozilla, the foundation responsible for the Firefox web browser, uses R to visualize Web activity.

Evolution of R

- R is an implementation of S programming language which was created by John Chambers at Bell Labs.
- R was initially written by Ross Ihaka and Robert Gentleman at the Department of Statistics of the University of Auckland in Auckland, New Zealand.
- R made its first public appearance in 1993.
- A large group of individuals has contributed to R by sending code and bug reports.Since mid-1997 there has been a core group (the "R Core Team") who can modify the R source code archive.
- In the year 2000 R 1.0.0 released.
- R 3.0.0 was released in 2013.

Features of R:

- R supports procedural programming with functions and **object-oriented programming** with generic functions. Procedural programming includes procedure, records, modules, and procedure calls. While object-oriented programming language includes class, objects, and functions.
- **Packages** are part of R programming. Hence, they are useful in collecting sets of **R functions** into a single unit.
- R is a well-developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.
- R has an effective data handling and storage facility,
- R provides a suite of operators for calculations on arrays, lists, vectors and matrices.
- R provides a large, coherent and integrated collection of tools for data analysis. It provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.
- Rs programming features include database input, exporting data, viewing data, variable labels, missing data, etc.
- R is an interpreted language. So we can access it through command line interpreter.
- R supports **matrix** arithmetic.
- R, SAS, and SPSS are three statistical languages. Of these three statistical languages, R is the only an open source.

As a conclusion, R is world's most widely used statistics programming language. It is a good choice of data scientists and supported by a vibrant and talented community of contributors.

The prominent editors available for R programming language are:

- **RGUI**(R graphical user interface) - Rstudio – Studio R offers a richer editing environment than RGUI and makes some common tasks easier and more fun.
- **RStudio -** RStudio is an integrated development environment (IDE) for R language. RStudio is a code editor and development environment, with some nice features that make code development in R easy and fun.
- **R Command Prompt**

Once you have R environment setup, then it's easy to start your R command prompt by just clicking on R Software icon. This will launch R interpreter and you will get a prompt > where

you can start typing your programs or commands.

>x=6

> print(x)

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

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

Save the above code in a file test.R. Execute by opening that script in R editor, select all (Ctrl +A) and click on run line or selection (Ctrl+R) option in Edit menu of R console.

When we run the above program, it produces the following result.

[1] "Hello, World!"

It is **Very Important** to understand because these are the objects you will manipulate on a day-to-day basis in R. Dealing with object conversions is one of the most common sources of frustration for beginners.

To understand computations in R, two slogans are helpful:

- Everything that exists is an object.
- Everything that happens is a function call.

Variables, Datatypes in R:

Everything in R is an object.R has 6 atomic vector types.

- character
- numeric (real or decimal)
- integer
- logical
- complex

By *atomic*, we mean the vector only holds data of a single type.

- **character**: "a", "swc"
- **numeric**: 2, 15.5
- **integer**: 2L (the L tells R to store this as an integer)
- **logical**: TRUE, FALSE
- **complex**: 1+4i (complex numbers with real and imaginary parts)

R provides many functions to examine features of vectors and other objects, for example

- class() - what kind of object is it (high-level)?
- typeof() - what is the object's data type (low-level)?
- length() - how long is it? What about two dimensional objects?
- attributes() - does it have any metadata?

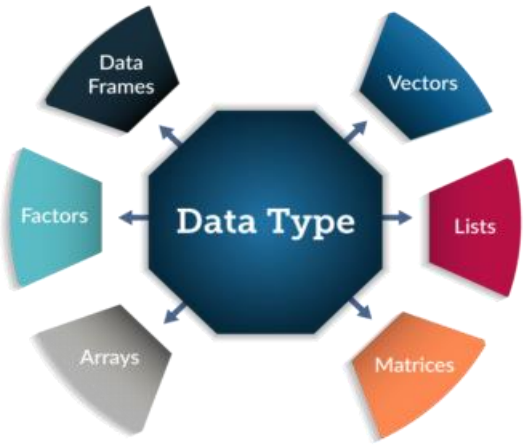
Basics types of data

- ✓ 4.5 is a decimal value called numeric.
- ✓ 4 is a natural value called integers. Integers are also numeric.
- ✓ TRUE or FALSE is a Boolean value called logical.
- ✓ The value inside " " or ' ' are text (string). They are called characters.
- ✓ 3+4i is a complex type of data.

Data Objects in R:

Data types are used to store information. In R, we do not need to declare a variable as some data type. The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable.There are mainly six data types present in R:

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



Scalar: Scalar variable A scalar is a single number. The following code creates. a scalar variable with the numeric value 5: x = 5. Vector variable A vector is a sequence of numbers.

1. Vector: A Vector is a sequence of data elements of the same basic type.

Example 1:

```
>vtr = c(1, 3, 5 ,7 9)          or        >vtr<- c (1, 3, 5 ,7 9)
>print(vtr)
o/p:    [1] 1 3 5 7 9
```

Example 2:creating sequence vector by using**colon operator.**

```
>v = 2:12
> print(v)
o/p:   [1]  2  3  4  5  6  7  8  9 10 11 12

> v = 3.5:10.5
> v
o/p:   [1]  3.5  4.5  5.5  6.5  7.5  8.5  9.5 10.5
```

Example 3: If the final element specified does not belong to the sequence then it is discarded.

```
> v <- 3.8:11
> v
[1]  3.8  4.8  5.8  6.8  7.8  8.8  9.8 10.8
```

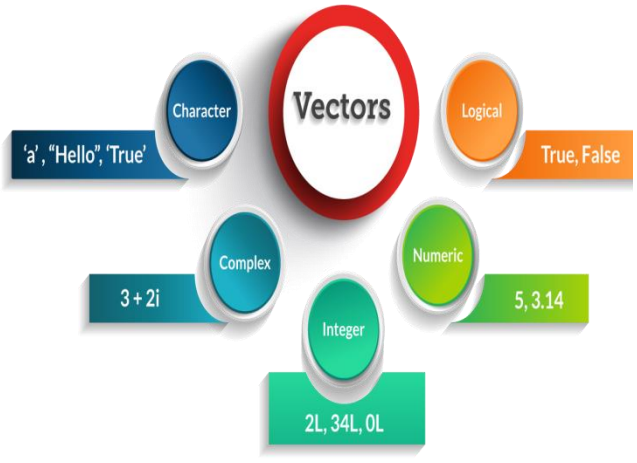
Example 4: using **seq()**(sequence) function, create vector from 1 to9 increment by 2.

```
>a=seq(1,10,by=2)
> a
o/p:   [1] 1 3 5 7 9
```

Example 4:Accessing vector elements by its position (index).

```
> day = c("Mon","Tue","Wed","Thurs","Fri","Sat","sun")
> print(day[3])
o/p:      [1] "Wed"
> weekend=day[c(6,7)]
> weekend
o/p:     [1] "Sat" "sun".
> print(day[c(-2,-3)])            // Negative indexing
o/p:     [1] "Mon"   "Thurs" "Fri"   "Sat"   "sun"
```

There are 5 Atomic vectors, also termed as five classes of vectors.



1. # Atomic vector of type **character**.
   Ex: v = "TRUE"
   print(class(v))
   o/pt: [1] "character"

2. #Atomic vector of type **numeric**.
   Ex: v = 23.5
   print(class(v))
   o/p: [1] "numeric"

3. # Atomic vector of type **integer.**
   Ex: v = 2L
   print(class(v))
   o/p: [1] "integer"

4. # Atomic vector of type **logical.**
   Ex: v = TRUE
   print(class(v))
   o/p: [1] "logical"

5. # Atomic vector of type **complex**.
   Ex: v = 2+5i
   print(class(v))
   o/p: [1] "complex"

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

Example 1:

```
>n = c(2, 3, 5)
>s = c("aa", "bb", "cc", "dd", "ee")
>x = list(n, s, TRUE)
>x
O/p –
        [[1]]
        [1] 2 3 5
        [[2]]
        [1] "aa"  "bb"  "cc" "dd" "ee"
        [[3]]
        [1] TRUE
```

Example 2: Accessing the elements of list

```
>mylist = list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2), list("green",12.3))
> print(mylist[1])
[[1]]
o/p:   [1] "Jan" "Feb" "Mar"
> print(mylist[c(1,3)])
o/p:   [[1]]
[1] "Jan" "Feb" "Mar"

[[2]]
[[2]][[1]]
[1] "green"

[[2]][[2]]
[1] 12.3
```

3. Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout. A Matrix is created using the **matrix() function.**

Example: matrix (data, nrow, ncol, byrow, dimnames) where,

- ✓ 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.

Examples 1:>Mat = matrix(c(1:16), nrow = 4, ncol = 4)
>Mat

Output :

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

Examples 2:using **byrow=TRUE/FALSE**

```
# Create a matrix.
>M = matrix( c('a','a','b','c','b','a'), nrow = 2, ncol = 3, byrow = TRUE)
>print(M)
o/p: [,1] [,2] [,3]
     [1,] "a"  "a"  "b"
     [2,] "c"  "b"  "a"


> M = matrix( c('a','a','b','c','b','a'),nrow=3,byrow=FALSE)
> M
   [,1] [,2]
[1,] "a"  "c"
[2,] "a"  "b"
[3,] "b"  "a"
```

Examples 3:Accessing the elements of matrix

> print(Mat[1,3]) to access 1st row third element.
o/p: [1] 9
> print(Mat[,3]) to access 3rd column
o/p: [1] 9 10 11 12
> print(Mat[1,]) to access 1st row
o/p: [1] 1 5 9 13

Examples 4:giving row names, column names of a matrix

```
>rownames = c("row1", "row2", "row3", "row4")
>colnames = c("col1", "col2", "col3")
> P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames, colnames))
> P
o/p:  col1 col2 col3
row1    3    4    5
row2    6    7    8
row3    9   10   11
row4   12   13   14

> P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames =list(c('a','b','c','d'),c(1,2,3)))
```

```
> P
o/p:  1  2  3
      a  3  4  5
      b  6  7  8
      c  9 10 11
      d 12 13 14

> P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames =list(1:4,5:7))
> P
o/p:  5  6  7
      1  3  4  5
      2  6  7  8
      3  9 10 11
      4 12 13 14
```

4. **Arrays:** Arrays are the R data objects which can store data in more than two dimensions.For example − If we create an array of dimension (2, 3, 4) then it creates 4 rectangular matrices each with 2 rows and 3 columns.**While matrices areconfined to two dimensions, arrays can be of any number of dimensions.**An array is created using the **array()** function. It takes vectors as input and uses the values in the **dim** parameter to create an array. In the below example we create 2 arrays of which are 3x3 matrices each.

Examples 1: Here we create two arrays with two elements which are 3x3 matrices each

```
>v1 <- c(5,9,3)
>v2 <- c(10,11,12,13,14,15)
>result<- array(c(v1,v2),dim = c(3,3,2))
>result
```

Output –

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

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

Examples 2.

```
>a <- array(c('green','yellow'),dim = c(3,3,2))
>print(a)
o/p:   , , 1
```

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

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

5. **Factors:** Factors are the data objects which are used to categorize the data and store it as levels. They can store both strings and integers. They are useful in data analysis for statistical modeling.

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

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

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

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

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

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

Ex2:

```
>data <- c("East","West","East","North","North","East","West","West“,"East“)
```

```
>factor_data<- factor(data)
>factor_data
```

Output :

```
[1] East  West  East  North North East  West  West  East
Levels: East North West
```

6. **Data Frames:** A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.Data frames are tabular data objects. Unlike a matrix in data frame each column can contain different modes of data. The first column can be numeric while the second column can be character and third column can be logical. It is a list of vectors of equal length.Data Frames are created using the data.frame() function.

```
# Create the data frame.
BMI <-        data.frame(
gender = c("Male", "Male","Female"),
height = c(152, 171.5, 165),
weight = c(81,93, 78),
  Age = c(42,38,26)
)
print(BMI)
```

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

```
gender height weight Age
1   Male  152.0     81  42
2   Male  171.5     93  38
3 Female  165.0     78  26
```

Ex2:

```
>std_id = c (1:5)
>std_name = c("Rick","Dan","Michelle","Ryan","Gary")
>marks = c(623.3,515.2,611.0,729.0,843.25)
>std.data<- data.frame(std_id, std_name, marks)
>std.data
```

Output :

	std_id	std_name	marks
1	1	Rick	623.30
2	2	Dan	515.20
3	3	Michelle	611.00
4	4	Ryan	729.00
5	5	Gary	843.25

By this, we come to the end of different data types in R. Next, let us move forward in R Tutorial blog and understand another key concept – flow control statements.

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

Variable Name	Validity	Reason
var_name2.	valid	Has letters, numbers, dot and underscore
var_name%	Invalid	Has the character '%'. Only dot(.) and underscore allowed.
2var_name	invalid	Starts with a number
.var_name , var.name	valid	Can start with a dot(.) but the dot(.)should not be followed by a number.
.2var_name	invalid	The starting dot is followed by a number making it invalid.
_var_name	invalid	Starts with _ which is not valid

Variable Assignment

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

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

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

# Assignment using rightward operator.
c(TRUE,1) -> var.3

print(var.1)
cat ("var.1 is ", var.1 ,"\n")
cat ("var.2 is ", var.2 ,"\n")
cat ("var.3 is ", var.3 ,"\n")
```

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

```
[1] 0 1 2 3
var.1 is  0 1 2 3
var.2 is  learn R
var.3 is  1 1
```

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

Data Type of a Variable

In R, a variable itself is not declared of any data type, rather it gets the data type of the R - object assigned to it. So R is called a dynamically typed language, which means that we can change a variable's data type of the same variable again and again when using it in a program.

```
var_x<- "Hello"
cat("The class of var_x is ",class(var_x),"\n")
class(var_x)
var_x<- 34.5
cat("  Now the class of var_x is ",class(var_x),"\n")
class(var_x)
var_x<- 27L
cat("   Next the class of var_x becomes ",class(var_x),"\n")
class(var_x)
```

Finding Variables

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

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

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

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

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

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

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

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

Deleting Variables

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

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

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

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

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

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

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

```
character(0)
```

Operators:

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

Types of Operators

We have the following types of operators in R programming −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Miscellaneous Operators

Arithmetic Operators:

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

Operator	Description	Example
+	Adds two vectors	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v+t) it produces the following result − [1] 10.0 8.5 10.0
−	Subtracts second vector from the first	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v-t) it produces the following result −

		[1] -6.0 2.5 2.0
*	Multiplies both vectors	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v*t) it produces the following result − [1] 16.0 16.5 24.0
/	Divide the first vector with the second	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v/t) When we execute the above code, it produces the following result − [1] 0.250000 1.833333 1.500000
%%	Give the remainder of the first vector with the second	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v%%t) it produces the following result − [1] 2.0 2.5 2.0
%/%	The result of division of first vector with second (quotient)	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v%/%t)

		it produces the following result − [1] 0 1 1
^	The first vector raised to the exponent of second vector	v <- c(2,5.5,6) t <- c(8, 3, 4) print(v^t) it produces the following result − [1] 256.000 166.375 1296.000

Attach() and detach() functions

The database is attached to the R search path. This means that the database is searched by R when evaluating a variable, so objects in the database can be accessed by simply giving their names.

attach() function makes the data available to the R Search Path.

Syntax:

attach(what, pos = 2L, name = deparse(substitute(what), backtick=FALSE), warn.conflicts = TRUE)

Arguments

what

'database'. This can be a data.frame or a list or a R data file created with save or NULL or an environment. See also 'Details'.

pos

integer specifying position in search() where to attach.

name

name to use for the attached database. Names starting with package: are reserved for library.

warn.conflicts

logical. If TRUE, warnings are printed about conflicts from attaching the database, unless that database contains an object .conflicts.OK. A conflict is a function masking a function, or a non-function masking a non-function.

Details

When evaluating a variable or function name R searches for that name in the databases listed by search. The first name of the appropriate type is used.

By attaching a data frame (or list) to the search path it is possible to refer to the variables in the data frame by their names alone, rather than as components of the data frame (e.g., in the example below, height rather than women$height).

By default the database is attached in position 2 in the search path, immediately after the user's workspace and before all previously attached packages and previously attached databases. This can be altered to attach later in the search path with the pos option, but you cannot attach at pos = 1.

attach(x)
x: dataframe, matrix, list

Following file has been used for ANOVA analysis:
Subtype,Gender,Expression
A,m,-0.54
A,m,-0.8
A,m,-1.03
A,m,-0.41
A,m,-1.31
A,f,-0.66
A,m,-0.43
A,m,1.01
A,f,-1.15
A,m,0.14
A,m,1.42
A,f,-0.3
A,m,-0.16
A,m,0.15
A,m,-0.62
A,m,-0.42
A,f,-0.4
A,m,-0.35
A,m,-0.42

Let first read in the data from the file:

>x <- read.csv("anova.csv",header=T,sep=",")

There are 3 variables, "Expression", "Gender" and "Subtype". We can display the variables by:
>x$Gender
[1] m mmmm f m m f m m f m mmm f m mmmmm f m mm f m mmm f m mmm

```
 [38] m mmmmmmmm f m f m mmmm f m m f m m f m mmm f m mmmmmmm
 [75] m m f m mmmm f m mmmmmmmm f m m f m m f m f m m f m m f m m f m
[112] m f m m f m mm f m mm f m f m f fffff m f m f ff m f fff m f m f
[149] m f f m f ffff m f m f f m f f m f f m f ff m f ff m f ff m f f m f
[186] f f m f f m f m m f m f m f f m f ffff m f f m f ff m mm f m mm f f
[223] f ffff m mm f m f f m f ff m f ff m f fff m f m f fff m f ff m
[260] f f m f fffff m f f m f fffff m f f
Levels: f m
```

We can't use the variable "Gender" in R Search Path:

```
>gender
Error: object 'Gender' not found
```

After attach the object "x", "Gender" can be used globally:

```
>attach(x)
>Gender
 [1] m mmmm f m m f m m f m mmm f m mmmmm f m mm f m mmm f m mmm
 [38] m mmmmmmmm f m f m mmmm f m m f m m f m mmm f m mmmmmmm
 [75] m m f m mmmm f m mmmmmmmm f m m f m m f m f m m f m m f m m f m
[112] m f m m f m mm f m mm f m f m f fffff m f m f ff m f fff m f m f
[149] m f f m f ffff m f m f f m f f m f f m f ff m f ff m f ff m f f m f
[186] f f m f f m f m m f m f m f f m f ffff m f f m f ff m mm f m mm f f
[223] f ffff m mm f m f f m f ff m f ff m f fff m f m f fff m f ff m
[260] f f m f fffff m f f m f fffff m f f
Levels: f m
```

detach() function reverses the process:

```
>detach(x)
>Gender
Error: object 'Gender' not found
```

UNIT-II

Importing data into R

One of the most important features we need to be able to do in R is import existing data, whether it be .txt files, .csv files, or even .xls (Excel files). If we can't import data into R, then we can't do anything. It is often necessary to import sample textbook data into R before you start working on your homework.

Reading Tabular Data Files:

If you have a **.txt or a tab-delimited text file**, or a file with table like structure then you can easily import it by using the basic R function **read.table().** It's good to know that the read.table() function is the most important and commonly used function to import simple data files into R. It is easy and flexible. Reads a file in table format and creates a data frame from it.

Syntax: read.table(file, header = FALSE, sep = "", quote = "\"", dec = ".", row.names, col.names, ……)

Here

- file: You have to specify the file name, or Full path along with file name. You can also use the URL of the external (online) txt files. For example, sampleFile.txt or "C:/Users/Suresh/Documents/R Programs/sampleFile.txt"
- header: If the text file contains Columns names as the First Row then please specify TRUE otherwise, FALSE
- sep: It is a short form of separator. You have to specify the character that is separating the fields. ", "means data is separated by comma. The default separator is "white space", that is one or more spaces, tabs, carriage return etc.
- quote: the set of quoting characters. To disable quoting altogether, use quote = "".If your character values (ex: Last-Name, Occupation, Education column etc) are enclosed in quotes then you have to specify the quote type. For double quotes we use: quote = "\"".
- dec: the character used in the file for decimal points.
- row.names: A Character vector that contains the row names for the returned data frame

Example 1: A data table can resides in a text file. The cells inside the table are separated by blank characters. Here is an example of a table with 4 rows and 3 columns.

```
100   a1   b1
200   a2   b2
300   a3   b3
400   a4   b4
```

Now copy and paste the table above in a file named "mydata.txt" with a text editor. Then load the data into the workspace with the function read.table.

```
> mydata = read.table("mydata.txt")  # read text file
> mydata                             # print data frame
  V1 V2 V3
1 100 a1 b1
2 200 a2 b2
3 300 a3 b3
4 400 a4 b4
>mydata1=read.table("rain.txt
```

Example 2:

```
rain<- read.table("C:/Users/SUNITHA/Desktop/rain.txt",header=TRUE,sep=",")
o/p: rain
  month rain_mm flow_cmm
1     1     128    15000
```

```
2    2     98    12000
3    3     92    11000
4    4     77     9800
5    5     68     7600
```

Reading CSV Files:

The csv file is a text file in which the values in the columns are separated by a comma. While R can read excel .xls and .xlsx files these filetypes often cause problems. **Comma separated files (.csv)** are much easier to work with. It's best to save these files as csv before reading them into R. If you need to read in a csv with R the best way to do is by using **read.csv()** function.

Syntax: read.csv (file, header = TRUE, sep = ",", quote = "\"", dec = ".", ..…)

Here

- file: You have to specify the file name, or Full path along with file name. You can also use the URL of the external (online) txt files. For example, sampleFile.txt or "C:/Users/Suresh/Documents/R Programs/sampleFile.txt"
- header: If the text file contains Columns names as the First Row then please specify TRUE otherwise, FALSE
- sep: It is a short form of separator. You have to specify the character that is separating the fields. ", "means data is separated by comma. The default separator is "white space", that is one or more spaces, tabs, carriage return etc.
- quote: the set of quoting characters. To disable quoting altogether, use quote = "".If your character values (ex: Last-Name, Occupation, Education column etc) are enclosed in quotes then you have to specify the quote type. For double quotes we use: quote = "\"".
- dec: the character used in the file for decimal points.

Example1: Let's consider the following data present in the file named **input.csv**.

You can create this file using windows notepad by copying and pasting this data. Save the file as **input.csv** using the save As All files(*.*) option in notepad.

```
id,name,salary,start_date,dept
1,Rick,623.3,2012-01-01,IT
2,Dan,515.2,2013-09-23,Operations
3,Michelle,611,2014-11-15,IT
4,Ryan,729,2014-05-11,HR
5,Gary,843.25,2015-03-27,Finance
6,Nina,578,2013-05-21,IT
7,Simon,632.8,2013-07-30,Operations
8,Guru,722.5,2014-06-17,Finance
```

Following is a simple example of **read.csv()** function to read a CSV file available in your current working directory −

```
data <- read.csv("input.csv")
print(data)
O/P:  id,  name,  salary,  start_date,    dept
1     1   Rick    623.30   2012-01-01     IT
2     2   Dan     515.20   2013-09-23     Operations
```

```
3     3    Michelle 611.00    2014-11-15      IT
4     4    Ryan     729.00    2014-05-11      HR
5    NA    Gary     843.25    2015-03-27      Finance
6     6    Nina     578.00    2013-05-21      IT
```

- We can also analyze the imported csv file for additional information.

```
> data = read.csv("input.csv",header=TRUE,sep=",")
>print(is.data.frame(data))      #o/p TRUE
>print(ncol(data))               #o/p 5
>print(nrow(data))               #o/p 6
```

- It's also possible to choose a file interactively using the function **file.choose**(), which is easy to select the file while reading.

```
# Read a csv file
my_data <- read.csv(file.choose())
```

Here you need to enter file name which you wanted to open by browsing.

Importing Data from Excel Files:

Microsoft Excel is the most widely used spreadsheet program which stores data in the .xls or .xlsx format. R can read directly from these files using some excel specific packages. Few such packages are - XLConnect, xlsx, gdata etc. We will be using xlsx package.

- **Importing and loading "xlsx" package:**

You can use the following command in the R console to install the "xlsx" package. It may ask to install some additional packages on which this package is dependent.

```
install.packages("xlsx")
```

- **Load the imported package into R workspace**

```
# Load the library into R workspace.

library("xlsx")
```

- **To check if you already installed the package or not, type in the following:**

any(grepl("<name of your package>", installed.packages())) o/p True - if installed already

- **Create an excel file and save it.**
  Open Microsoft excel. Create an excel file with name input.xlsx in worksheet named as sheet1.
- **Reading the Excel File**
  The input.xlsx is read by using the **read.xlsx()** function as shown below. The result is stored as a data frame in the R environment.

```
# Read the first worksheet in the file input.xlsx.

data <- read.xlsx("input.xlsx", sheetIndex = 1)
```

```
print(data)
```

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

```
    id,  name,    salary,  start_date,  dept
1    1   Rick     623.30   2012-01-01   IT
2    2   Dan      515.20   2013-09-23   Operations
3    3   Michelle 611.00   2014-11-15   IT
4    4   Ryan     729.00   2014-05-11   HR
5   NA   Gary     843.25   2015-03-27   Finance
6    6   Nina     578.00   2013-05-21   IT
```

- We can read an excel file by selecting sheet index or sheet name.

```
data <- read.xlsx("input.xlsx", sheetIndex = 1)

or

data <- read.xlsx("input.xlsx", sheetName = “sheet1”)
```

Loading and storing data clipboard:
R has a function writeClipboard that does what the name implies. However, the argument to writeClipboard may need to be cast to a character type. For example the code

writeClipboard( ):

```
> x <- "hello world"
> writeClipboard(x)
```

copies the string “hello world” to the clipboard as expected. However the code

```
> x <- 3.14
> writeClipboard(x)
```

Produces the error message. The solution is to call writeClipboard(as.character(x)), casting the object x to a character string.

readClipboard() :
The companion function for writeClipboard is readClipboard.
The command

```
x <- readClipboard()
```

will assign the contents of the clipboard to the vector x. Each line becomes an element of x. The elements will be character strings, even if the clipboard contained a column of numbers before the readClipboard command was executed. If you select a block of numbers from Excel, each row becomes a single string containing tabs where there were originally cell boundaries.

- You can also load directly from the clipboard:

```
# First copy the data to the clipboard
data <- read.table('clipboard', header=TRUE)
```

```
# Or:
# data <- read.csv('clipboard')
```

Ex: display a file by using display file option in file menu, and select any data from that and press ctrl+c. (data is loaded into clipboard) then execute the following command to load the clipboard:

```
data <- read.csv('clipboard')
> data
  Subtype Gender Expression
1       A      m      -0.54
2       A      f      -0.80
p3      B      f      -1.03
4       C      m      -0.41
```

- It is possible to write delimited data to terminal (stdout()), so that it can be copied and pasted elsewhere. Or it can be written directly to the clipboard.

```
write.csv(data, stdout(), row.names=FALSE)
"Subtype","Gender","Expression"
"A","m",-0.54
"A","f",-0.8
"B","f",-1.03
"C","m",-0.41
> write.csv(data, 'clipboard', row.names=FALSE)
```

Saving an R data file:

As you work with your data in R you will eventually want to save it to disk. This will allow you to work with the data later and still retain the original dataset. It can also allow you to share your dataset with other analysts. One of the simplest ways to save your data is by saving it into an RData file with the function **save( )**. The function **save**() can be used to save one or more R objects to a specified file (in **.RData** or **.rda** file formats). The function can be read back from the file using the function **load**().

Let us create an example dataset. The following R script creates an R data frame [explained in another topic of this learning infrastructure] for you to practice saving.

```
x <- c(1:10) # create a numeric vector
y <- c(11:20) # create a numeric vector.
z <- c(21:30) # create a numeric vector
m <- cbind(x, y, z) # create a matrix
d <- as.data.frame(m) # create a data frame

# create a text vector
t <- c("red", "blue", "red", "white", "blue", "white", "red","blue", "white", "white")
df <- cbind(d, t) # add the text vector to the data frame
```

Your R session now has a data frame object named **df** that you can use for the exercises below. You can save the data frame **df** [from the above example] using this command:

save(df, file = "df.RData")

While the **save( )** command can have several arguments, this example uses only two. The first argument is the name of your R data object, **df** in this example. The second argument assigns a name to the RData file, **df.RData** in this example. You can use any text as your file name as long as it does not contain any embedded spaces. While you do not have to use the **.RData** extension, this is a recommended practice because the **.RData** extension will help RStudio to identify your R datasets. Notice that the file name is enclosed in quotation marks.

```
# Saving an object in RData format
 save(data1, file = "data.RData")

# Save multiple objects
save(data1, data2, file = "data.RData")
```

It's also possible to specify the file name for saving your work space:

```
save.image(file = "my_work_space.RData")
```

To restore your workspace, type this:

```
load("my_work_space.RData")


# save your command history
savehistory(file="myfile") # default is ".Rhistory"

# recall your command history
loadhistory(file="myfile") # default is ".Rhistory“
```

Loading R data:

Loading Rdata Files in a Convenient Way. These functions **loads** an **Rdata** object saved as a data frame or a matrix in the current **R** environment. The function **load**.**Rdata** saves the loaded object in the global environment while **load**.Rdata2 **loads** the object only specified environments.

```
load.Rdata(filename, objname)

load.Rdata2(filename, path=getwd())
```

Arguments
Filename - Rdata file (matrix or data frame)
Objname - Object name. This object will be a global variable in R.
Path - Directory from which the dataset should be loaded

```
# load a data frame in the file "data_s3.Rdata" and save this
# as the object "dat.s3"
load.Rdata( filename="data_s3.Rdata", "dat.s3" )
head(dat.s3)
```

Writing Data to a File

The R base function **write.table()** can be used to export a data frame or a matrix to a file.
A simplified format is as follow:

Syntax:

write.table(x, file, append = FALSE, sep = " ", dec = ".", row.names = TRUE, col.names = TRUE)

where

- **x**: a matrix or a data frame to be written.
- **file**: a character specifying the name of the result file.
- **sep**: the field separator string, e.g., sep = "\t" (for tab-separated value).
- **dec**: the string to be used as decimal separator. Default is "."
- **row.names**: either a logical value indicating whether the row names of x are to be written along with x, or a character vector of row names to be written.
- **col.names**: either a logical value indicating whether the column names of x are to be written along with x, or a character vector of column names to be written. If col.names = NA and row.names = TRUE a blank column name is added, which is the convention used for CSV files to be read by spreadsheets.

Ex1: Write data from R to a txt file: **write.table**(my_data, file = "my_data.txt", sep = "")

Ex2: The R code below exports the built-in R *mtcars* data set to a tab-separated (*sep = "\t"*) file called mtcars.txt in the current working directory:

```
# Loading mtcars data
data("mtcars")
# Writing mtcars data
write.table(mtcars, file = "mtcars.txt", sep = "\t", row.names = TRUE, col.names = NA)
```

If you don't want to write row names, use row.names = FALSE as follow:

```
write.table(mtcars, file = "mtcars.txt", sep = "\t",row.names = FALSE)
```

Writing data to a CSV file:

- **write.csv()** uses "." for the decimal point and a comma (",") for the separator.
- **write.csv2()** uses a comma (",") for the decimal point and a semicolon (";") for the separator.

The syntax is as follow:

```
write.csv(my_data, file = "my_data.csv")
```

Ex3: write data from R to a csv file: **write.csv**(my_data, file = "my_data.csv")
Ex4: write.table(xdata, "c:/mydata.txt", sep="\t")
library(xlsx)
Ex5: write.xlsx(ydata, "c:/mydata.xlsx")

What is Data Manipulation in R?
Data structures provide the way to represent data in data analytics. We can manipulate data in R for analysis and visualization. One of the most important aspects of computing with data Data Manipulation in R and enable its subsequent analysis and visualization.

How to round off numbers in R:
Although R can calculate accurately to up to 16 digits, you don't always want to use that many digits. In this case, you can use a couple functions in R to round numbers. To round a number to two digits after the decimal point, for example, use the round() function as follows:

```
> round(123.456,digits=2)
[1] 123.46
```

You also can use the round() function to round numbers to multiples of 10, 100, and so on. For that, you just add a negative number as the digits argument:

```
> round(-123.456,digits=-2)
[1] -100
```

If you want to specify the number of significant digits to be retained, regardless of the size of the number, you use the signif() function instead:

```
> signif(-123.456,digits=4)
[1] -123.5
```

Both round() and signif() round numbers to the nearest possibility. So, if the first digit that's dropped is smaller than 5, the number is rounded down. If it's bigger than 5, the number is rounded up.

If the first digit that is dropped is exactly 5, R uses a rule that's common in programming languages: Always round to the nearest even number. round(1.5) and round(2.5) both return 2,
Ex:

```
> round(-4.5)
[1] -4
> round(-4.6)
[1] -5
> round(-4.4)
[1] -4
```

Contrary to round(), three other functions always round in the same direction:

- floor(x) rounds to the nearest integer that's smaller than x. So floor(123.45) becomes 123 and floor(-123.45) becomes –124.
- ceiling(x) rounds to the nearest integer that's larger than x. This means ceiling (123.45) becomes 124 and ceiling(123.45) becomes –123.

- trunc(x) rounds to the nearest integer in the direction of 0. So trunc(123.65) becomes 123 and trunc(-123.65) becomes –123.

Merging data in R:

- In R you use the merge() function to combine data frames. This powerful function tries to identify columns or rows that are common between the two different data frames.
- The simplest form of merge() finds the intersection between two different sets of data. In other words, to create a data frame that consists of those states that are cold as well as large, use the default version of merge():

syntax: The merge() function takes quite a large number of arguments. These arguments can look quite intimidating until you realize that they form a smaller number of related arguments:

- **x:** A data frame.
- **y:** A data frame.
- **by, by.x, by.y:** The names of the columns that are common to both x and y. The default is to use the columns with common names between the two data frames.
- **all, all.x, all.y:** Logical values that specify the type of merge. The default value is all=FALSE (meaning that only the matching rows are returned).
- That last group of arguments — all, all.x and all.y — deserves some explanation. These arguments determine the type of merge that will happen.

Ex1: To merge two dataframes (datasets) horizontally, use the merge function. In most cases, you join two dataframes by one or more common key variables (i.e., an inner join).

- # merge two dataframes by id
  total <- merge(dataframea,dataframeb,by="id")
- # merge two dataframes by id and country
  total <- merge(dataframea,dataframeb,by=c("id","country"))

Ex2: Let's create two data frames and merge them:

>exp <- data.frame(samples=c("a","b","c"),values=c(2.43,5.32,-1.23))
>backedup <- data.frame(patients=c("a","b","c"),marked=c("yes","yes","no"))

- Merge two data frames exp and backedup

> xyz <- merge(exp,backedup,by.x="samples",by.y="patients")

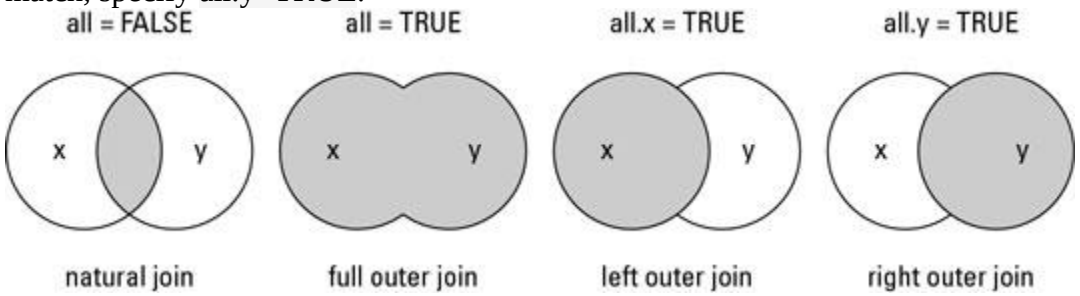
```
>xyz
  samples values marked
1       a   2.43    yes
2       b   5.32    yes
3       c  -1.23     no
```

Different types of merging

The merge() function allows four ways of combining data:

- **Natural join:** To keep only rows that match from the data frames, specify the argument all=FALSE.
- **Full outer join:** To keep all rows from both data frames, specify all=TRUE.

- **Left outer join:** To include all the rows of your data frame x and only those from y that match, specify all.x=TRUE.
- **Right outer join:** To include all the rows of your data frame y and only those from x that match, specify all.y=TRUE.



Ex: We will create two data frame df1 and df2 to illustrate joins in R. We will create two data frame df1 and df2 to illustrate joins in R.

```
> df1 =  data.frame(CustomerId = c(1:6), Product=c(rep("Toaster”,3), rep("Radio",3)))
> df2 = data.frame(CustomerId = c(2,4,6), State = c(rep("Alabama",2), rep("Ohio",1)))
> df1
```

CustomerId	Product
1	Toaster
2	Toaster
3	Toaster
4	Radio
5	Radio
6	Radio

>df2

CustomerId	State
2	Alabama
4	Alabama
6	Ohio

We can merge these data frames by using the merge function and its optional parameters:
Natural join: merge(x=df1, y=df2, by = “CustomerId”,all = FALSE)
Outer join: merge(x = df1, y = df2, by = "CustomerId", all = TRUE)
Left outer: merge(x = df1, y = df2, by = "CustomerId", all.x = TRUE)
Right outer: merge(x = df1, y = df2, by = "CustomerId", all.y = TRUE)

Data aggregation in R:

You have a data set and you need to quickly organize it to perform your data analysis. Where do you start? You could create a table of statistics which summarizes data by aggregating it. We use aggregate function to do this.

aggregate( ) function: Aggregate is a function in base R which can, as the name suggests, aggregate the inputted data.frame d.f by applying a function specified by the FUN parameter to each column of sub-data.frames defined by the by input parameter.

Syntax: aggregate(x, by, FUN, ……)

- The first argument to the function is usually a data.frame.
- The by argument is a list of variables to group by. This must be a list even if there is only one variable, as in the example.
- The FUN argument is the function which is applied to all columns (i.e., variables) in the grouped data. Because we cannot calculate the average of categorical variables such as Name and Shift, they result in empty columns, which I have removed for clarity.

The process involves two stages. First, collate individual cases of raw data together with a grouping variable. Second, perform which calculation you want on each group of cases. These two stages are wrapped into a single function.

To perform aggregation, we need to specify three things in the code:

- The data that we want to aggregate
- The variable to group by within the data
- The calculation to apply to the groups (what you want to find out)

Ex: Load the example data by running the following R code:

data=DownloadXLSX("https://wiki.qresearchsoftware.com/images/1/1b/Aggregation_data.xlsx", want.row.names = FALSE, want.data.frame = TRUE)

	Name	Role	Shift	Salary	Age
1	Ann	Cook	Lunch	1000	19
2	Bob	Server	Lunch	1200	24
3	Charlie	Cook	Lunch	1400	29
4	Dave	Server	Lunch	1500	24
5	Ed	Manager	Lunch	2200	32
6	Fred	Manager	Dinner	2000	41
7	Gary	Cook	Dinner	2000	28
8	Henry	Server	Dinner	1500	30
9	Ian	Cook	Dinner	1600	22
10	Jo	Server	Dinner	1800	25

Perform aggregation with the following R code.

agg = aggregate(data,by = list(data$Role),FUN = mean)

This produces a table of the average salary and age by role, as below.

	Group.1	Salary	Age
1	Cook	1500.0	24.4
2	Manager	2100.0	36.5
3	Server	1500.0	25.8

How to Rename Columns in R

This page will show you how to rename columns in R with examples using either the existing column name or the column number to specify which column name to change.

Ex:

```
> d <- data.frame(alpha=1:3, beta=4:6, gamma=7:9)
> d
  alpha beta gamma
1     1    4     7
2     2    5     8
3     3    6     9
```

We can display the names of columns

```
> names(d)
[1] "alpha" "beta"  "gamma"
```

We can rename a column by using indexing the column name

```
> names(d)[names(d)=="beta"] = "two"
```

We can rename more than one column at a time by using rename function

```
> library(plyr)
> rename(d, c("beta"="two", "gamma"="three"))
> d
  alpha two gamma
1     1   4     7
2     2   5     8
3     3   6     9
```

We can also rename a column by indexing with number.

```
> names(d)[3] = "three"
> d
  alpha two three
1     1   4     7
2     2   5     8
3     3   6     9
```

HOW TO SORT AND ORDER DATA IN R

One very common task in data analysis and reporting is sorting information, which you can do easily in R. we use **sort( ), order( ) and arrange( )** functions in R to sort the data.

sort() function sorts a vector.

Syntax: sort(x, decreasing = FALSE, na.last = NA, ...)

x: vector
decreasing: decrease or not
na.last: if TRUE, NAs are put at last position, FALSE at first, if NA, remove them (default)

...

Sort Vectors:

```
>x <- c(1,2.3,2,3,4,8,12,43,-4,-1,NA)
```

How to sort a vector in ascending order

```
>sort(x)
 [1] -4.0 -1.0  1.0  2.0  2.3  3.0  4.0  8.0 12.0 43.0
```

How to sort a vector in decreasing order

```
>sort(x,decreasing=TRUE)
 [1] 43.0 12.0  8.0  4.0  3.0  2.3  2.0  1.0 -1.0 -4.0
```

put NA values at last position

```
>sort(x,decreasing=TRUE, na.last=TRUE)
 [1] 43.0 12.0  8.0  4.0  3.0  2.3  2.0  1.0 -1.0 -4.0   NA
```

put NA values at first

```
>sort(x,decreasing=TRUE, na.last=FALSE)
 [1]   NA 43.0 12.0  8.0  4.0  3.0  2.3  2.0  1.0 -1.0 -4.0
```

order( ) function: order() function sorts a vector, matrix or data frame.
Syntax: order(x, decreasing = FALSE, na.last = NA, ...)
Where:
x: vector
decreasing: decrease or not
na.last: if TRUE, NAs are put at last position, FALSE at first, if NA, remove them (default)

Ex1: Sort Vectors:

```
>x <- c(1,2.3,2,3,4,8,12,43,-4,-1,NA)
>order(x)
 [1] -4.0 -1.0  1.0  2.0  2.3  3.0  4.0  8.0 12.0 43.0
>order(x,decreasing=TRUE)
 [1] 43.0 12.0  8.0  4.0  3.0  2.3  2.0  1.0 -1.0 -4.0
>order(x,decreasing=TRUE, na.last=TRUE)
 [1] 43.0 12.0  8.0  4.0  3.0  2.3  2.0  1.0 -1.0 -4.0   NA
>order(x,decreasing=TRUE, na.last=FALSE)
 [1]   NA 43.0 12.0  8.0  4.0  3.0  2.3  2.0  1.0 -1.0 -4.0
```

Ex2: Order data frame:

```
>BOD     #R built-in dataset, Biochemical Oxygen Demand
  Time demand
1    1    8.3
2    2   10.3
```

```
3    3   19.0
4    4   16.0
5    5   15.6
6    7   19.8
```

Sort by "demand" column:

```
>BOD[with(BOD,order(demand)),]
  Time demand
1    1    8.3
2    2   10.3
5    5   15.6
4    4   16.0
3    3   19.0
6    7   19.8
```

arrange() function:

We learned how to sort the values with the function sort(). The library dplyr has its sorting function called arrange( ). The arrange() verb can reorder one or many rows, either ascending (default) or descending.

We can reorder the data of a data table, by the value of one or more columns (i.e., variables).

- Sort a data frame rows in ascending order (from low to high) using the R function arrange() [dplyr package]
- Sort rows in descending order (from high to low) using arrange() in combination with the function desc() [dplyrpackage]

Ex:

```
> arrange(A): Ascending sort of variable A
> arrange(A, B): Ascending sort of variable A and B
> arrange(desc(A), B): Descending sort of variable A and ascending sort of B
> arrange(mtcars, cyl, disp)     # mtcars is data set, cyl and disp are columns in that dataset.
> arrange(mtcars, desc(disp))   # descending sort on disp column
```

Data Manipulation in R

Data structures provide the way to represent data in data analytics. We can manipulate data in R for analysis and visualization. One of the most important aspects of computing with data Data Manipulation in R and enable its subsequent analysis and visualization.

Creating Subsets of Data in R

As we know, data size is increasing exponentially and doing an analysis of complete data is very time-consuming. So the data is divided into small sized samples and analysis of samples is done. The process of creating samples is called sub-setting. Different methods of sub-setting in R are:

a. $

The dollar sign operator selects a single element of data. When you use this operator with a data frame, the result is always a vector.

b. [[
Similar to $ in R, the double square brackets operator also returns a single element, but it offers the flexibility of referring to the elements by position rather than by name. It can be used for data frames and lists.
c. [
The single square bracket operator in R returns multiple elements of data. The index within the square brackets can be a numeric vector, a logical vector, or a character vector.

Selecting Rows/Observations:
R has powerful indexing features for accessing object elements. These features can be used to select and exclude variables and observations. The following code snippets demonstrate ways to keep or delete variables and observations and to take random samples from a dataset.
Selecting (Keeping) Variables

```
# select variables v1, v2, v3
myvars <- c("v1", "v2", "v3")
newdata <- mydata[myvars]

# exclude 3rd and 5th variable
newdata <- mydata[c(-3,-5)]

# delete variables v3 and v5
mydata$v3 <- mydata$v5 <- NULL

# select 1st and 5th thru 10th variables
newdata <- mydata[c(1,5:10)]
```

Selecting Observations

```
# first 5 observations
newdata <- mydata[1:5,]

# based on variable values
newdata <- mydata[ which(mydata$gender=='F' & mydata$age > 65), ]

# or
attach(mydata)
newdata <- mydata[ which(gender=='F' & age > 65),]
detach(mydata)
```

Selection using the Subset Function
The subset( ) function is the easiest way to select variables and observations. In the following example, we select all rows that have a value of age greater than or equal to 20 or age less then 10. We keep the ID and Weight columns.

```
# using subset function
newdata <- subset(mydata, age >= 20 | age < 10, select=c(ID, Weight))
```

In the next example, we select all men over the age of 25 and we keep variables weight through income (weight, income and all columns between them).

```
# using subset function (part 2)
newdata <- subset(mydata, sex=="m" & age > 25, select=weight:income)
```

Commands to Extract Rows and Columns

The following represents different commands which could be used to extract one or more rows with one or more columns. Note that the output is extracted as a data frame. This could be checked using the class command.

```
# All Rows and All Columns
df[,]
# First row and all columns
df[1,]
# First two rows and all columns
df[1:2,]
# First and third row and all columns
df[ c(1,3), ]
# First Row and 2nd and third column
df[1, 2:3]
# First, Second Row and Second and Third Column
df[1:2, 2:3]
# Just First Column with All rows
df[, 1]
# First and Third Column with All rows
df[,c(1,3)]
```

How to **identify** and **remove duplicate** data in R.
You will learn how to use the following R base and **dplyr** functions:
R base functions
duplicated(): for identifying duplicated elements and
unique(): for extracting unique elements,
distinct() [**dplyr** package] to remove duplicate rows in a data frame.

Load the tidyverse packages, which include dplyr:
library(tidyverse)

We'll use the R built-in iris data set, which we start by converting into a tibble data frame (tbl_df) for easier data analysis.

```
my_data <- as_tibble(iris)
my_data
```

```
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <fct>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            1.4         0.2 setosa
## 3          4.7         3.2          1.3         0.2 setosa
```

```
## 4         4.6         3.1          1.5         0.2 setosa
## 5         5          3.6          1.4         0.2 setosa
## 6         5.4         3.9          1.7         0.4 setosa
## # ... with 144 more rows
```

Find and drop duplicate elements

The R function duplicated() returns a logical vector where TRUE specifies which elements of a vector or data frame are duplicates.

Given the following vector:

```
x <- c(1, 1, 4, 5, 4, 6)
```

- To find the position of duplicate elements in x, use this:

```
duplicated(x)
## [1] FALSE  TRUE FALSE FALSE  TRUE FALSE
```

- Extract duplicate elements:

```
x[duplicated(x)]
## [1] 1 4
```

- If you want to remove duplicated elements, use !duplicated(), where ! is a logical negation:

```
x[!duplicated(x)]
## [1] 1 4 5 6
```

- Following this way, you can remove duplicate rows from a data frame based on a column values, as follow:

```
# Remove duplicates based on Sepal.Width columns
my_data[!duplicated(my_data$Sepal.Width), ]
## # A tibble: 23 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <fct>
## 1         5.1         3.5          1.4         0.2 setosa
## 2         4.9         3            1.4         0.2 setosa
## 3         4.7         3.2          1.3         0.2 setosa
## 4         4.6         3.1          1.5         0.2 setosa
## 5         5           3.6          1.4         0.2 setosa
## 6         5.4         3.9          1.7         0.4 setosa
## # ... with 17 more rows
```

! is a logical negation. **!duplicated**() means that we don't want duplicate rows.

Extract unique elements

Given the following vector:

```
x <- c(1, 1, 4, 5, 4, 6)
```

You can extract unique elements as follow:

unique(x)

```
## [1] 1 4 5 6
```

It's also possible to apply **unique**() on a data frame, for removing duplicated rows as follow:

```
unique(my_data)
```

Remove duplicate rows in a data frame

The function distinct() [*dplyr* package] can be used to keep only unique/distinct rows from a data frame. If there are duplicate rows, only the first row is preserved. It's an efficient version of the R base function unique().

In this chapter, we describe key functions for identifying and removing duplicate data:

- Remove duplicate rows based on one or more column values: my_data %>% dplyr::distinct(Sepal.Length)
- R base function to extract unique elements from vectors and data frames: unique(my_data)
- R base function to determine duplicate elements: duplicated(my_data)

Example 1 : Remove Duplicate Rows based on all the variables (Complete Row)

The distinct function is used to eliminate duplicates.
x1 = distinct(mydata)
In this dataset, there is not a single duplicate row so it returned same number of rows as in mydata.

Example 2 : Remove Duplicate Rows based on a variable

The .keep_all function is used to retain all other variables in the output data frame.
x2 = distinct(mydata, Index, .keep_all= TRUE)

Example 3 : Remove Duplicates Rows based on multiple variables

In the example below, we are using two variables - Index, Y2010 to determine uniqueness.
x2 = distinct(mydata, Index, Y2010, .keep_all= TRUE)

select( ) Function

It is used to select only desired variables.
syntax : select(data , ...)
data : Data Frame
.... : Variables by name or by function

Example 1 : Selecting Variables (or Columns)

Suppose you are asked to select only a few variables. The code below selects variables "Index", columns from "State" to "Y2008".
mydata2 = select(mydata, Index, State:Y2008)

Example 2 : Dropping Variables

The minus sign before a variable tells R to drop the variable.
mydata = select(mydata, -Index, -State)
The above code can also be written like :
mydata = select(mydata, -c(Index,State))

Example 3 : Selecting or Dropping Variables starts with 'Y'

The starts_with() function is used to select variables starts with an alphabet.
mydata3 = select(mydata, starts_with("Y"))

Adding a negative sign before starts_with() implies dropping the variables starts with 'Y'
mydata33 = select(mydata, -starts_with("Y"))

The following functions helps you to select variables based on their names.

Helpers	Description
starts_with()	Starts with a prefix
ends_with()	Ends with a prefix
contains()	Contains a literal string
matches()	Matches a regular expression
num_range()	Numerical range like x01, x02, x03.
one_of()	Variables in character vector.
everything()	All variables.

Example 4 : Selecting Variables contain 'I' in their names
mydata4 = select(mydata, contains("I"))

filter( ) Function
It is used to subset data with matching logical conditions.
syntax : filter(data , ...)
data : Data Frame
.... : Logical Condition

Example 1 : Filter Rows
Suppose you need to subset data. You want to filter rows and retain only those values in which Index is equal to A.
mydata7 = filter(mydata, Index == "A")

```
 Index   State  Y2002  Y2003  Y2004  Y2005  Y2006  Y2007  Y2008  Y2009
1     A  Alabama 1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173
2     A   Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541
3     A  Arizona 1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330
4     A Arkansas 1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980

Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
1 1237582 1440756 1186741 1852841 1558906 1916661
2 1629616 1230866 1512804 1985302 1580394 1979143
3 1300521 1130709 1907284 1363279 1525866 1647724
4 1669295 1928238 1216675 1591896 1360959 1329341
```

Example 2 : Multiple Selection Criteria
The %in% operator can be used to select multiple items. In the following program, we are telling R to select rows against 'A' and 'C' in column 'Index'.
mydata7 = filter(mydata6, Index %in% c("A", "C"))

UNIT-III

Data Manipulation in R

Data structures provide the way to represent data in data analytics. We can manipulate data in R for analysis and visualization. One of the most important aspects of computing with data Data Manipulation in R and enable its subsequent analysis and visualization.

Creating Subsets of Data in R

As we know, data size is increasing exponentially and doing an analysis of complete data is very time-consuming. So the data is divided into small sized samples and analysis of samples is done. The process of creating samples is called sub-setting. Different methods of sub-setting in R are:

a. $

The dollar sign operator selects a single element of data. When you use this operator with a data frame, the result is always a vector.

b. [[

Similar to $ in R, the double square brackets operator also returns a single element, but it offers the flexibility of referring to the elements by position rather than by name. It can be used for data frames and lists.

c. [

The single square bracket operator in R returns multiple elements of data. The index within the square brackets can be a numeric vector, a logical vector, or a character vector.

Selecting Rows/Observations:

R has powerful indexing features for accessing object elements. These features can be used to select and exclude variables and observations. The following code snippets demonstrate ways to keep or delete variables and observations and to take random samples from a dataset.

Selecting (Keeping) Variables

```
# select variables v1, v2, v3
myvars <- c("v1", "v2", "v3")
newdata <- mydata[myvars]

# exclude 3rd and 5th variable
newdata <- mydata[c(-3,-5)]

# delete variables v3 and v5
mydata$v3 <- mydata$v5 <- NULL

# select 1st and 5th thru 10th variables
newdata <- mydata[c(1,5:10)]
```

Selecting Observations

```
# first 5 observations
newdata <- mydata[1:5,]

# based on variable values
newdata <- mydata[ which(mydata$gender=='F' & mydata$age > 65), ]

# or
attach(mydata)
newdata <- mydata[ which(gender=='F' & age > 65),]
detach(mydata)
```

Selection using the Subset Function

The subset( ) function is the easiest way to select variables and observations. In the following example, we select all rows that have a value of age greater than or equal to 20 or age less then 10. We keep the ID and Weight columns.

```
# using subset function
newdata <- subset(mydata, age >= 20 | age < 10, select=c(ID, Weight))
```

In the next example, we select all men over the age of 25 and we keep variables weight through income (weight, income and all columns between them).

```
# using subset function (part 2)
newdata <- subset(mydata, sex=="m" & age > 25, select=weight:income)
```

Commands to Extract Rows and Columns

The following represents different commands which could be used to extract one or more rows with one or more columns. Note that the output is extracted as a data frame. This could be checked using the class command.

```
# All Rows and All Columns
df[,]
# First row and all columns
df[1,]
# First two rows and all columns
df[1:2,]
```

```
# First and third row and all columns
df[ c(1,3), ]
# First Row and 2nd and third column
df[1, 2:3]
# First, Second Row and Second and Third Column
df[1:2, 2:3]
# Just First Column with All rows
df[, 1]
# First and Third Column with All rows
df[,c(1,3)]
```

How to **identify** and **remove duplicate** data in R.
You will learn how to use the following R base and **dplyr** functions:
R base functions
duplicated(): for identifying duplicated elements and
unique(): for extracting unique elements,
distinct() [**dplyr** package] to remove duplicate rows in a data frame.

Load the tidyverse packages, which include dplyr:
library(tidyverse)

We'll use the R built-in iris data set, which we start by converting into a tibble data frame (tbl_df) for easier data analysis.
my_data <- as_tibble(iris)
my_data

```
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <fct>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            1.4         0.2 setosa
## 3          4.7         3.2          1.3         0.2 setosa
## 4          4.6         3.1          1.5         0.2 setosa
## 5          5           3.6          1.4         0.2 setosa
## 6          5.4         3.9          1.7         0.4 setosa
## # ... with 144 more rows
```

Find and drop duplicate elements
The R function duplicated() returns a logical vector where TRUE specifies which elements of a vector or data frame are duplicates.
Given the following vector:
x <- c(1, 1, 4, 5, 4, 6)

- To find the position of duplicate elements in x, use this:

```
duplicated(x)
## [1] FALSE  TRUE FALSE FALSE  TRUE FALSE
```

- Extract duplicate elements:
  x[duplicated(x)]
  ## [1] 1 4
- If you want to remove duplicated elements, use !duplicated(), where ! is a logical negation:
  x[!duplicated(x)]
  ## [1] 1 4 5 6
- Following this way, you can remove duplicate rows from a data frame based on a column values, as follow:
  # Remove duplicates based on Sepal.Width columns
  my_data[!duplicated(my_data$Sepal.Width),]
  ## # A tibble: 23 x 5
  ## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
  ## <dbl> <dbl> <dbl> <dbl> <fct>
  ## 1 5.1 3.5 1.4 0.2 setosa
  ## 2 4.9 3 1.4 0.2 setosa
  ## 3 4.7 3.2 1.3 0.2 setosa
  ## 4 4.6 3.1 1.5 0.2 setosa
  ## 5 5 3.6 1.4 0.2 setosa
  ## 6 5.4 3.9 1.7 0.4 setosa
  ## # ... with 17 more rows
  ! is a logical negation. **!duplicated**() means that we don't want duplicate rows.

  Extract unique elements
  Given the following vector:
  x <- c(1, 1, 4, 5, 4, 6)
  You can extract unique elements as follow:
  unique(x)
  ## [1] 1 4 5 6
  It's also possible to apply **unique**() on a data frame, for removing duplicated rows as follow:
  unique(my_data)

  Remove duplicate rows in a data frame
  The function distinct() [*dplyr* package] can be used to keep only unique/distinct rows from a data frame. If there are duplicate rows, only the first row is preserved. It's an efficient version of the R base function unique().

  In this chapter, we describe key functions for identifying and removing duplicate data:
- Remove duplicate rows based on one or more column values: my_data %>% dplyr::distinct(Sepal.Length)
- R base function to extract unique elements from vectors and data frames: unique(my_data)
- R base function to determine duplicate elements: duplicated(my_data)

  Example 1 : Remove Duplicate Rows based on all the variables (Complete Row)
  The distinct function is used to eliminate duplicates.
  x1 = distinct(mydata)

In this dataset, there is not a single duplicate row so it returned same number of rows as in mydata.

Example 2 : Remove Duplicate Rows based on a variable
The .keep_all function is used to retain all other variables in the output data frame.
x2 = distinct(mydata, Index, .keep_all= TRUE)

Example 3 : Remove Duplicates Rows based on multiple variables
In the example below, we are using two variables - Index, Y2010 to determine uniqueness.
x2 = distinct(mydata, Index, Y2010, .keep_all= TRUE)

select( ) Function
It is used to select only desired variables.
syntax : select(data , ...)
data : Data Frame
.... : Variables by name or by function

Example 1 : Selecting Variables (or Columns)
Suppose you are asked to select only a few variables. The code below selects variables "Index", columns from "State" to "Y2008".
mydata2 = select(mydata, Index, State:Y2008)

Example 2 : Dropping Variables
The minus sign before a variable tells R to drop the variable.
mydata = select(mydata, -Index, -State)
The above code can also be written like :
mydata = select(mydata, -c(Index,State))

Example 3 : Selecting or Dropping Variables starts with 'Y'
The starts_with() function is used to select variables starts with an alphabet.
mydata3 = select(mydata, starts_with("Y"))
Adding a negative sign before starts_with() implies dropping the variables starts with 'Y'
mydata33 = select(mydata, -starts_with("Y"))
The following functions helps you to select variables based on their names.

Helpers	Description
starts_with()	Starts with a prefix
ends_with()	Ends with a prefix
contains()	Contains a literal string
matches()	Matches a regular expression
num_range()	Numerical range like x01, x02, x03.
one_of()	Variables in character vector.
everything()	All variables.

Example 4 : Selecting Variables contain 'I' in their names
mydata4 = select(mydata, contains("I"))

filter( ) Function
It is used to subset data with matching logical conditions.
syntax : filter(data , ...)
data : Data Frame
.... : Logical Condition

Example 1 : Filter Rows
Suppose you need to subset data. You want to filter rows and retain only those values in which Index is equal to A.
mydata7 = filter(mydata, Index == "A")

	Index	State	Y2002	Y2003	Y2004	Y2005	Y2006	Y2007	Y2008	Y2009
1	A	Alabama	1296530	1317711	1118631	1492583	1107408	1440134	1945229	1944173
2	A	Alaska	1170302	1960378	1818085	1447852	1861639	1465841	1551826	1436541
3	A	Arizona	1742027	1968140	1377583	1782199	1102568	1109382	1752886	1554330
4	A	Arkansas	1485531	1994927	1119299	1947979	1669191	1801213	1188104	1628980

	Y2010	Y2011	Y2012	Y2013	Y2014	Y2015
1	1237582	1440756	1186741	1852841	1558906	1916661
2	1629616	1230866	1512804	1985302	1580394	1979143
3	1300521	1130709	1907284	1363279	1525866	1647724
4	1669295	1928238	1216675	1591896	1360959	1329341

Example 2 : Multiple Selection Criteria
The %in% operator can be used to select multiple items. In the following program, we are telling R to select rows against 'A' and 'C' in column 'Index'.
mydata7 = filter(mydata6, Index %in% c("A", "C"))

Graphics in R:

Graphical facilities are an important and extremely versatile component of the R environment. It is possible to use the facilities to display a wide variety of statistical graphs and also to build entirely new types of graph.
R is capable of creating high quality graphics. Graphs are typically created using a series of high-level and low-level plotting commands. High-level functions create new plots and low-level functions add information to an existing plot. Customize graphs (line style, symbols, color, etc) by specifying graphical parameters. Specify graphic options using the par() function.

Once the device driver is running, R plotting commands can be used to produce a variety of graphical displays and to create entirely new kinds of display. Plotting commands are divided into two basic groups.

- **High-level plotting commands:** High Level plotting functions create a new plot on the graphics device, possibly with axes, labels, titles and so on. High-level plotting functions are designed to generate a complete plot of the data passed as arguments to the function. Where appropriate, axes, labels and titles are automatically generated (unless you request otherwise.) High-level plotting commands always start a new plot, erasing the current plot if necessary.

plot()	Scatter plot
hist()	Histogram
boxplot()	Boxplot
qqplot(), qqnorm(), qqline()	Quantile plots
interaction.plot()	Interaction plot
sunflower plot()	Sunflower scatter plot
pairs()	Scatter plot matrix
symbols()	Draw symbols on a plot
dotchart(), barplot(), pie()	Dot chart, bar chart, pie chart
curve()	Draw a curve from a given function
image()	Create a grid of colored rectangles with colors based on the values of a third variable

- **Low-level plotting commands: Low-level** plotting functions add more information to an existing plot, such as extra points, lines and labels. Sometimes the high-level plotting functions don't produce exactly the kind of plot you desire. In this case, low-level plotting commands can be used to add extra information (such as points, lines or text) to the current plot.

points()	Add points to a figure
lines()	Add lines to a figure
text()	Insert text in the plot region
mtext()	Insert text in the figure and outer margins
title()	Add figure title or outer title
legend()	Insert legend
axis(), axis.Date()	Customize axes
abline()	Add horizontal and vertical lines or a single line
box()	Draw a box around the current plot
polygon()	Draw a polygon
rect()	Draw a rectangle
arrows()	Draw arrows
segments()	Draw line segments

Bar Chart:

A bar chart represents data in rectangular bars with length of the bar proportional to the value of the variable. R uses the function **barplot()** to create bar charts. R can draw both vertical and Horizontal bars in the bar chart. In bar chart each of the bars can be given different colors.

Syntax

The basic syntax to create a bar-chart in R is −

```
barplot(H,xlab,ylab,main, names.arg,col)
```

Following is the description of the parameters used −

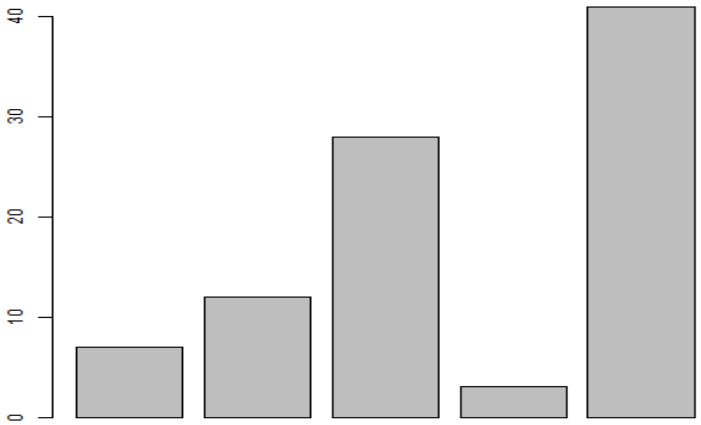
- **H** is a vector or matrix containing numeric values used in bar chart.
- **xlab** is the label for x axis.
- **ylab** is the label for y axis.
- **main** is the title of the bar chart.
- **names.arg** is a vector of names appearing under each bar.
- **col** is used to give colors to the bars in the graph.

Example

A simple bar chart is created using just the input vector and the name of each bar.The below script will create and save the bar chart in the current R working directory.

```
# Create the data for the chart
H <- c(7,12,28,3,41)
# Give the chart file a name
png(file = "barchart.png")
# Plot the bar chart
barplot(H)
# Save the file
dev.off()
```

When we execute above code, it produces following result −



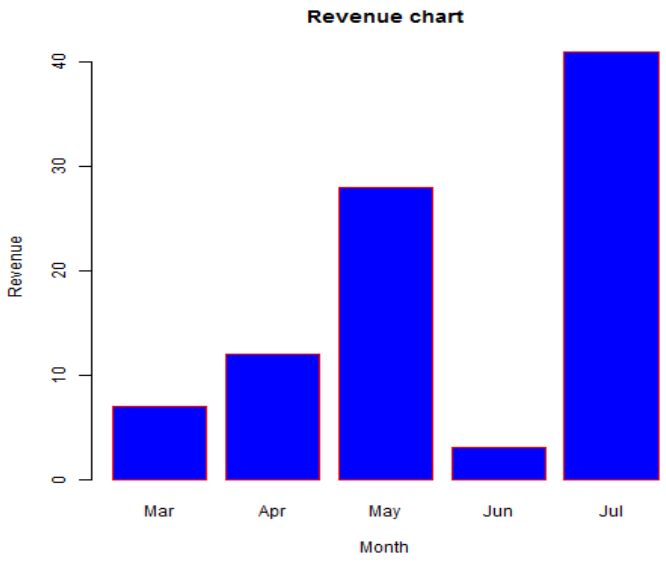
Bar Chart Labels, Title and Colors

The features of the bar chart can be expanded by adding more parameters. The **main** parameter is used to add **title**. The **col** parameter is used to add colors to the bars. The **args.name** is a vector having same number of values as the input vector to describe the meaning of each bar.

Example

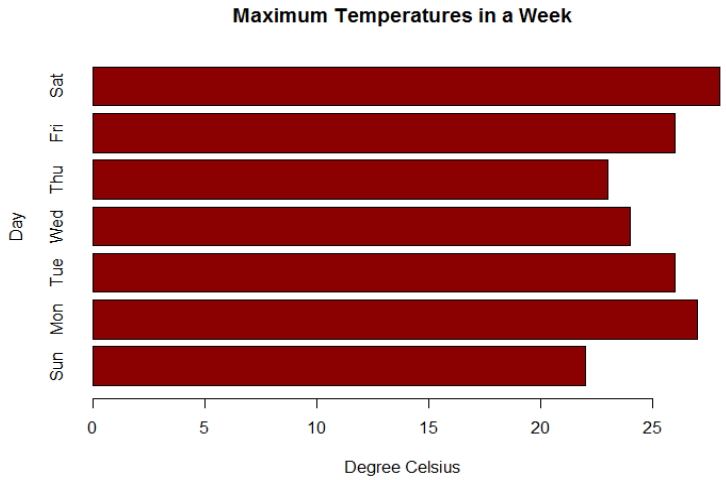
The below script will create and save the bar chart in the current R working directory.

```
# Create the data for the chart
H <- c(7,12,28,3,41)
M <- c("Mar","Apr","May","Jun","Jul")

# Give the chart file a name
png(file = "barchart_months_revenue.png")

# Plot the bar chart
barplot(H,names.arg=M,xlab="Month",ylab="Revenue",col="blue",
main="Revenue chart",border="red")

# Save the file
dev.off()
```



We can also plot bars horizontally by providing the argument horiz = TRUE.

```
# barchart with added parameters
barplot(max.temp, main = "Maximum Temperatures in a Week", xlab = "Degree Celsius", ylab =
"Day",
names.arg = c("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"), col = "darkred", horiz =
TRUE)
```



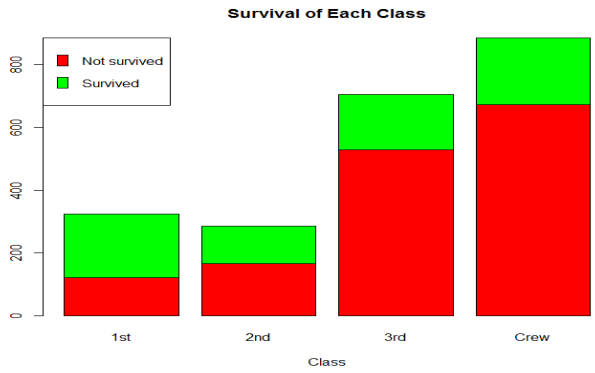
How to plot barplot with matrix?

As mentioned before, barplot() function can take in vector as well as matrix. If the input is matrix, a stacked bar is plotted. Each column of the matrix will be represented by a stacked bar. Let us consider the following matrix which is derived from our Titanic dataset.

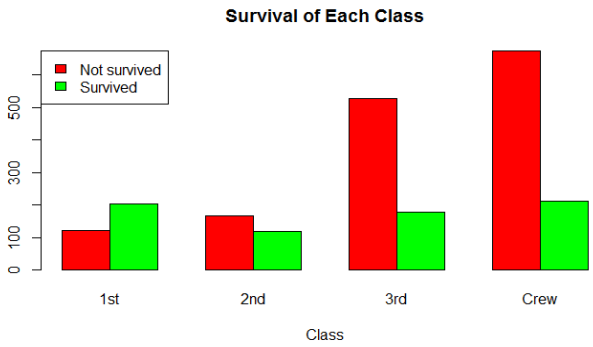
```
> titanic.data
          Class
Survival 1st 2nd 3rd Crew
No  122 167 528  673
Yes 203 118 178  212
```

This data is plotted as follows.

```
barplot(titanic.data, main = "Survival of Each Class", xlab = "Class",col = c("red","green") )
legend("topleft", c("Not survived","Survived"),     fill = c("red","green") )
```



Instead of a stacked bar we can have different bars for each element in a column juxtaposed to each other by specifying the parameter beside = TRUE as shown below.



Pie Chart:

A pie-chart is a representation of values as slices of a circle with different colors. The slices are labeled and the numbers corresponding to each slice is also represented in the chart.In R the pie chart is created using the **pie()** function which takes positive numbers as a vector input. The additional parameters are used to control labels, color, title etc.

Syntax:

```
pie(x, labels, radius, main, col, clockwise)
```

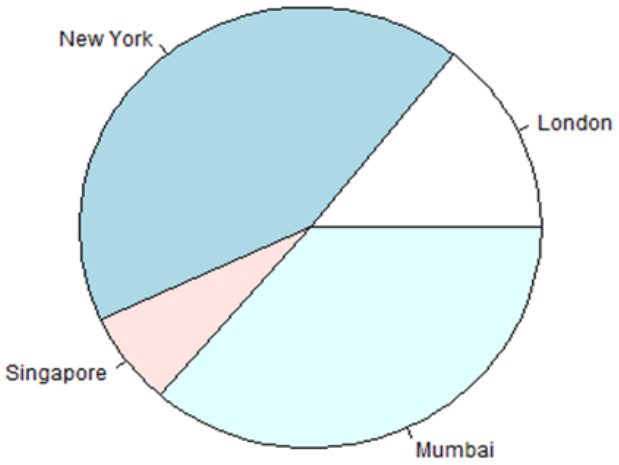
Following is the description of the parameters used −

- **x** is a vector containing the numeric values used in the pie chart.
- **labels** is used to give description to the slices.
- **radius** indicates the radius of the circle of the pie chart.(value between −1 and +1).
- **main** indicates the title of the chart.
- **col** indicates the color palette.
- **clockwise** is a logical value indicating if the slices are drawn clockwise or anti clockwise.

Example:

A very simple pie-chart is created using just the input vector and labels. The below script will create and save the pie chart in the current R working directory

```
# Create data for the graph.
x <- c(21, 62, 10, 53)
labels<- c("London", "New York", "Singapore", "Mumbai")

# Give the chart file a name.
png(file = "city.jpg")

# Plot the chart.
pie(x,labels)

# Save the file.
dev.off()
```



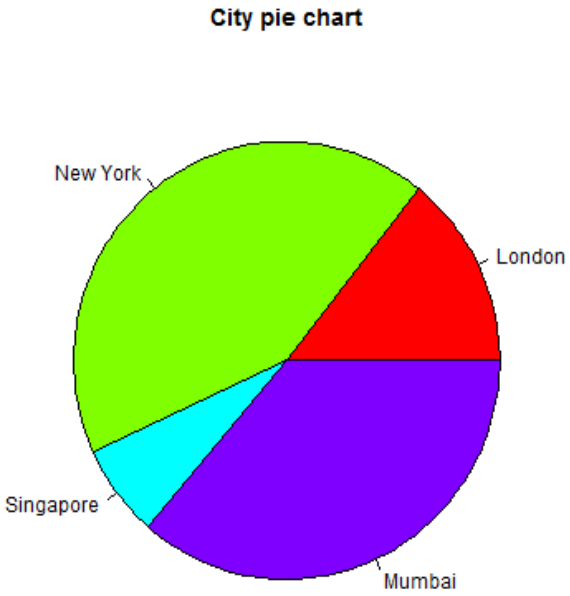
Pie Chart Title and Colors

We can expand the features of the chart by adding more parameters to the function. We will use parameter **main** to add a title to the chart and another parameter is **col** which will make use of rainbow color pallet while drawing the chart. The length of the pallet should be same as the number of values we have for the chart. Hence we use length(x).

Example

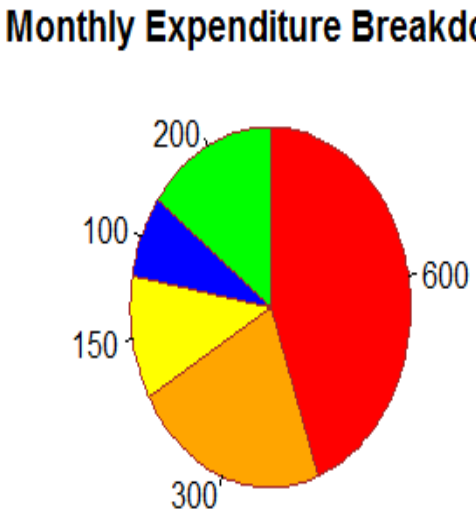
The below script will create and save the pie chart in the current R working directory.

```
# Create data for the graph.
x <- c(21, 62, 10, 53)
labels<- c("London", "New York", "Singapore", "Mumbai")

# Give the chart file a name.
png(file = "city_title_colours.jpg")

# Plot the chart with title and rainbow color pallet.
pie(x, labels, main = "City pie chart", col = rainbow(length(x)))

# Save the file.
dev.off()
```



Example 2: Pie chart with additional parameters

```
pie(expenditure, labels=as.character(expenditure), main="Monthly Expenditure Breakdown",
col=c("red","orange","yellow","blue","green"), border="brown", clockwise=TRUE )
```



Box Plots:

Boxplots are a measure of how well distributed is the data in a data set. It divides the data set into three quartiles. This graph represents the minimum, maximum, median, first quartile and third quartile in the data set. It is also useful in comparing the distribution of data across data sets by drawing boxplots for each of them. The boxplot() function takes in any number of numeric vectors, drawing a boxplot for each vector. You can also pass in a list (or data frame)

with numeric vectors as its components. Boxplots are created in R by using the **boxplot()** function.

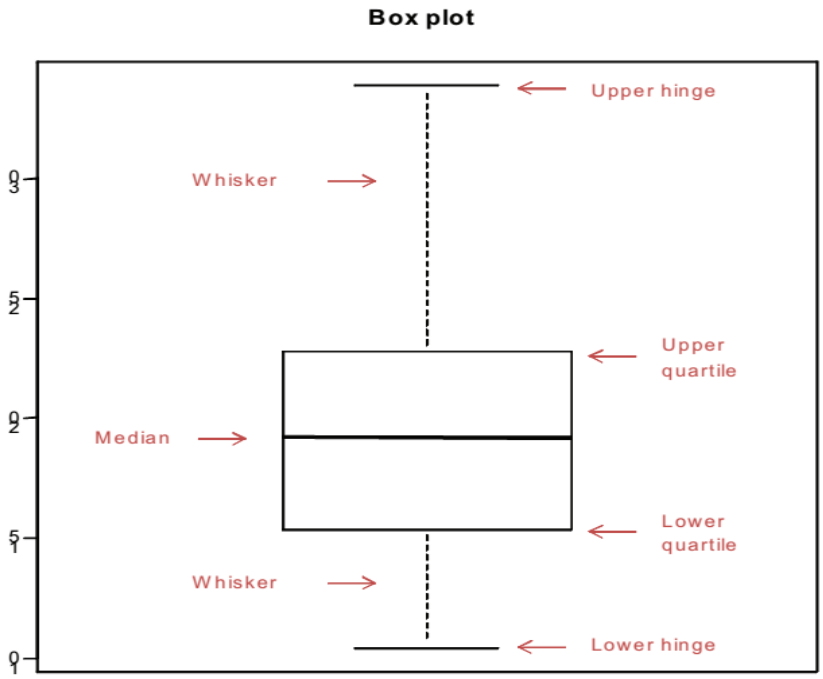
Syntax

The basic syntax to create a boxplot in R is −

```
boxplot(x, data, notch, varwidth, names, main)
```

Following is the description of the parameters used −

- **x** is a vector or a formula.
- **data** is the data frame.
- **notch** is a logical value. Set as TRUE to draw a notch.
- **varwidth** is a logical value. Set as true to draw width of the box proportionate to the sample size.
- **names** are the group labels which will be printed under each boxplot.
- **main** is used to give a title to the graph.



Ex1:

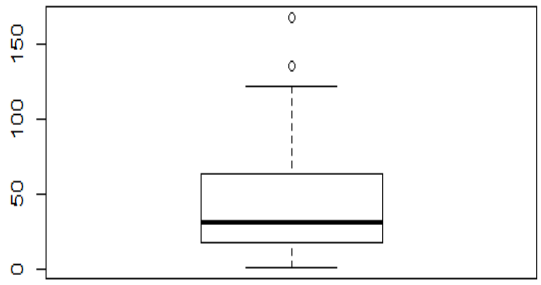
Let us use the built-in dataset airquality which has "Daily air quality measurements in New York, May to September 1973."-R documentation.

> str(airquality)

'data.frame': 153 obs. of 6 variables:
$ Ozone : int 41 36 12 18 NA 28 23 19 8 NA ...
$ Solar.R: int 190 118 149 313 NA NA 299 99 19 194 ...
$ Wind : num 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
$ Temp : int 67 72 74 62 56 66 65 59 61 69 ...
$ Month : int 5 5 5 5 5 5 5 5 5 5 ...
$ Day : int 1 2 3 4 5 6 7 8 9 10 ...

Let us make a boxplot for the ozone readings.

>boxplot(airquality$Ozone)

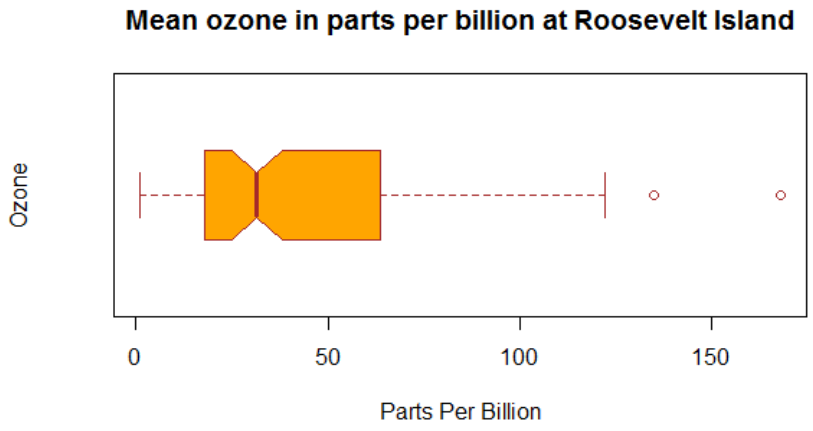


We can see that data above the median is more dispersed. We can also notice two outliers at the higher extreme.

Ex2:

We can pass in additional parameters to control the way our plot looks. Some of the frequently used ones are, main-to give the title, xlab and ylab-to provide labels for the axes, col to define color etc. Additionally, with the argument horizontal = TRUE we can plot it horizontally and with notch = TRUE we can add a notch to the box.

```
boxplot(airquality$Ozone,
main = "Mean ozone in parts per billion at Roosevelt Island",
xlab = "Parts Per Billion",
ylab = "Ozone",
col = "orange",
border = "brown",
horizontal = TRUE,
notch = TRUE
)
```



Multiple Boxplots

We can draw multiple boxplots in a single plot, by passing in a list, data frame or multiple vectors.

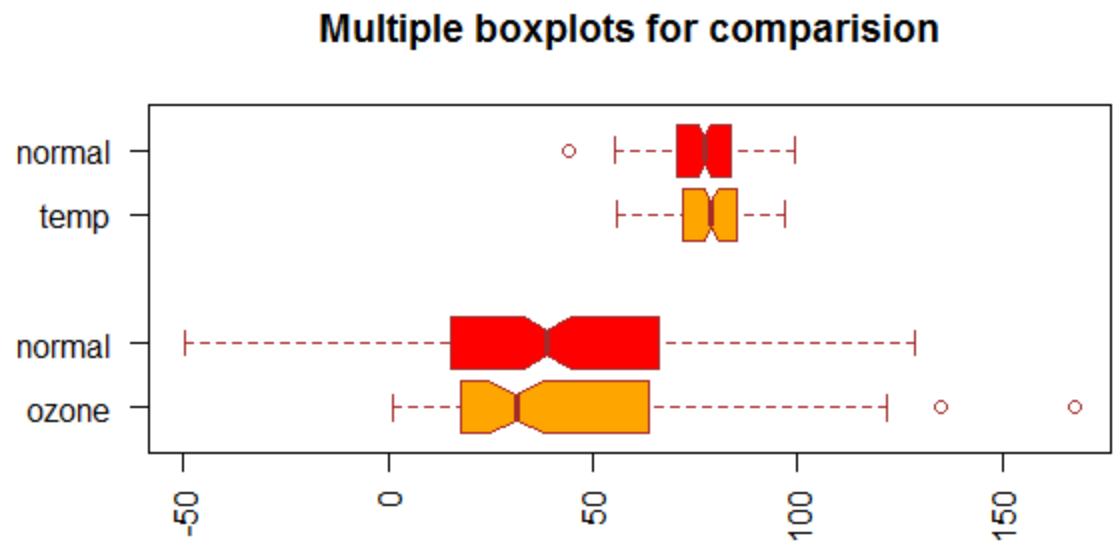
Let us consider the Ozone and Temp field of airquality dataset. Let us also generate normal distribution with the same mean and standard deviation and plot them side by side for comparison.

```
# prepare the data
>ozone <- airquality$Ozone
>temp <- airquality$Temp
# gererate normal distribution with same mean and sd
>ozone_norm <- rnorm(200,mean=mean(ozone, na.rm=TRUE), sd=sd(ozone, na.rm=TRUE))
>temp_norm <- rnorm(200,mean=mean(temp, na.rm=TRUE), sd=sd(temp, na.rm=TRUE))
```

- rnorm generates a random value from the normal distribution. runif generates a random value from the uniform.

Now we us make 4 boxplots with this data. We use the arguments at and names to denote the place and label.

```
>boxplot(ozone, ozone_norm, temp, temp_norm,
main = "Multiple boxplots for comparision",
at = c(1,2,4,5),
names = c("ozone", "normal", "temp", "normal"),
las = 2,
col = c("orange","red"),
border = "brown",
horizontal = TRUE,
notch = TRUE
)
```



Scatter Plot:

The Scatter Plot in R Programming is very useful to visualize the relationship between two sets of data. The data is displayed as collection of points that shows the linear relation between those two data sets. For example, if we want to visualize the Age against Weight then we can use this Scatter Plot.

Scatterplots show many points plotted in the Cartesian plane. Each point represents the values of two variables. One variable is chosen in the horizontal axis and another in the vertical axis. The simple scatterplot is created using the **plot()** function.

Syntax

The basic syntax for creating scatterplot in R is −

```
plot(x, y, type,main, xlab, ylab, xlim, ylim, axes)
```

Following is the description of the parameters used −

- **x** is the data set whose values are the horizontal coordinates.
- **y** is the data set whose values are the vertical coordinates.
- type: Please specify, what type of plot you want to draw.
  - To draw Points, use type = “p”
  - To draw Lines use type = “l”
  - Use type = “h” for Histograms
  - Use type = “s” for stair steps
  - To draw over-plotted use type = “o”
- **main** is the tile of the graph.
- **xlab** is the label in the horizontal axis.
- **ylab** is the label in the vertical axis.
- **xlim** is the limits of the values of x used for plotting.
- **ylim** is the limits of the values of y used for plotting.

- **axes** indicate whether both axes should be drawn on the plot.

Example

We use the data set **"mtcars"** available in the R environment to create a basic scatterplot. Let's use the columns "wt" and "mpg" in mtcars.

```
input<- mtcars[,c('wt','mpg')]

print(head(input))
```

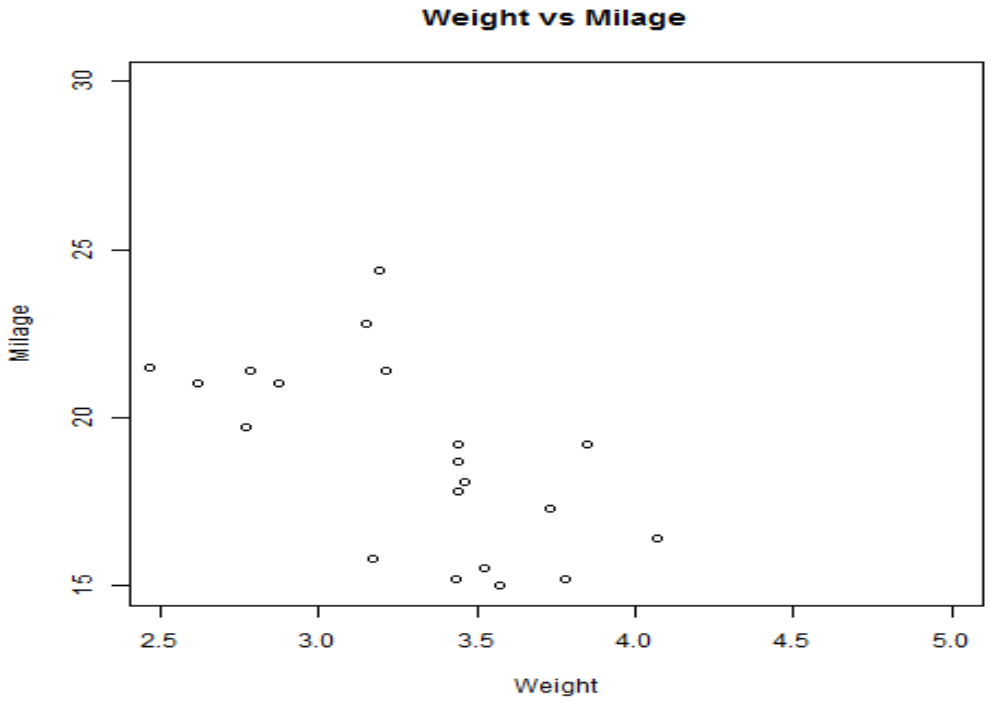
When we execute the above code, it produces the following result −

```
wt      mpg
Mazda RX4          2.620   21.0
Mazda RX4 Wag      2.875   21.0
Datsun 710         2.320   22.8
Hornet 4 Drive     3.215   21.4
Hornet Sportabout  3.440   18.7
Valiant            3.460   18.1
```

Creating the Scatterplot

The below script will create a scatterplot graph for the relation between wt(weight) and mpg(miles per gallon).

```
# Get the input values.
input<- mtcars[,c('wt','mpg')]
# Give the chart file a name.
png(file = "scatterplot.png")
# Plot the chart for cars with weight between 2.5 to 5 and mileage between 15 and 30.
plot(x = input$wt,y = input$mpg,
xlab = "Weight",
ylab = "Milage",
xlim = c(2.5,5),
ylim = c(15,30),
main = "Weight vs Milage" )
# Save the file.
dev.off()
```




```
# How to create a Scatter Plot in R Example
faithful

# Finding the Correlation
cor(faithful$eruptions, faithful$waiting)

# Drawing Scatter Plot
plot(faithful$eruptions, faithful$waiting)
```

Following statement will find the correlation between the eruptions, and waiting
cor(faithful$eruptions, faithful$waiting)

Change Colors of Scatter plot in R

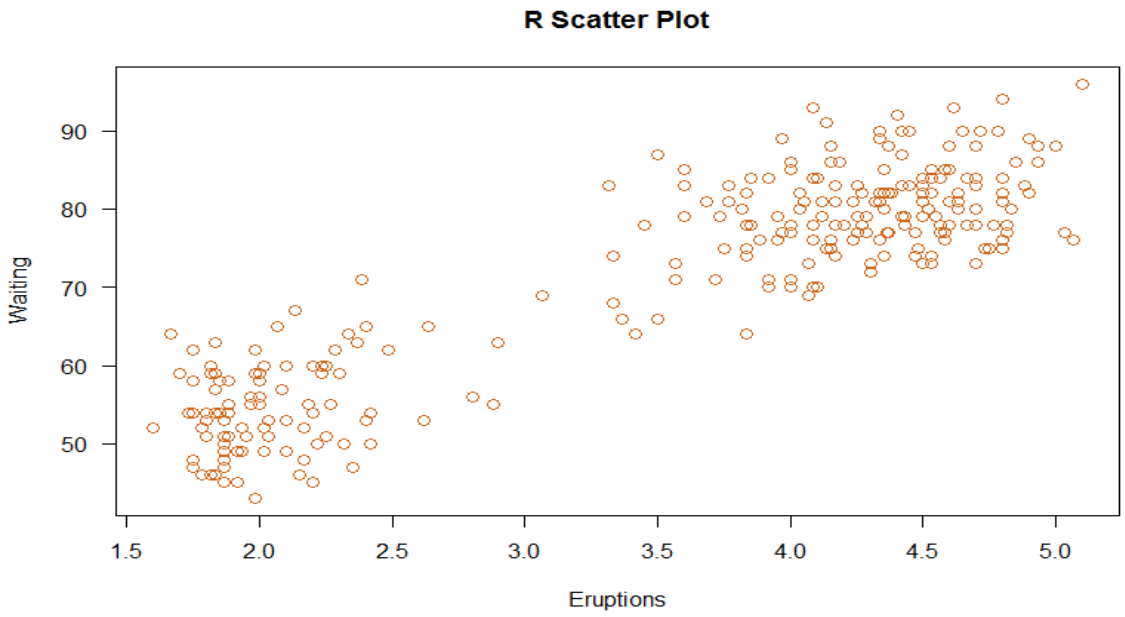
In this example we will show you, how to change the scatter plot color using **col** argument, and size of the character that represents the point using **cex** (character expansion) argument.

- **col:** Please specify the color you want to use for your Scatter plot.
- **cex:** Please specify the size of the point(s)

R CODE

```
# R Scatter Plot - Changing Color, Dot Size Example
# Drawing Scatter Plot
plot(faithful$eruptions, faithful$waiting,
     col = "chocolate",
     cex = 1.2,
     main = "R Scatter Plot",
```

```
    xlab = "Eruptions",
    ylab = "Waiting",
    las = 1)
```



Change Shapes and Axis limits of Scatter Plot in R

In this example we will show you, How to change the shape using ***pch*** argument.

- **xlim:** This argument can help you to specify the limits for the X-Axis
- **ylim:** This argument may help you to specify the Y-Axis limits

R CODE

```
# R Scatter Plot - Changing X, Y Limitations, Dot Sape Example
faithful

# Drawing Scatter Plot
plot(faithful$eruptions, faithful$waiting,
     col = "chocolate",
     pch = 8,
     main = "R Scatter Plot",
     xlab = "Eruptions",
     ylab = "Waiting",
     las = 1,
     xlim = c(1.5, 5.5),
     ylim = c(40, 100))
```

Low-Level Graphics:

The plot( ) function in R:

The most used plotting function in R programming is the plot() function. It is a generic function, meaning, it has many methods which are called according to the type of object passed to plot(). In the simplest case, we can pass in a vector and we will get a scatter plot of magnitude vs index. But generally, we pass in two vectors and a scatter plot of these points are plotted.

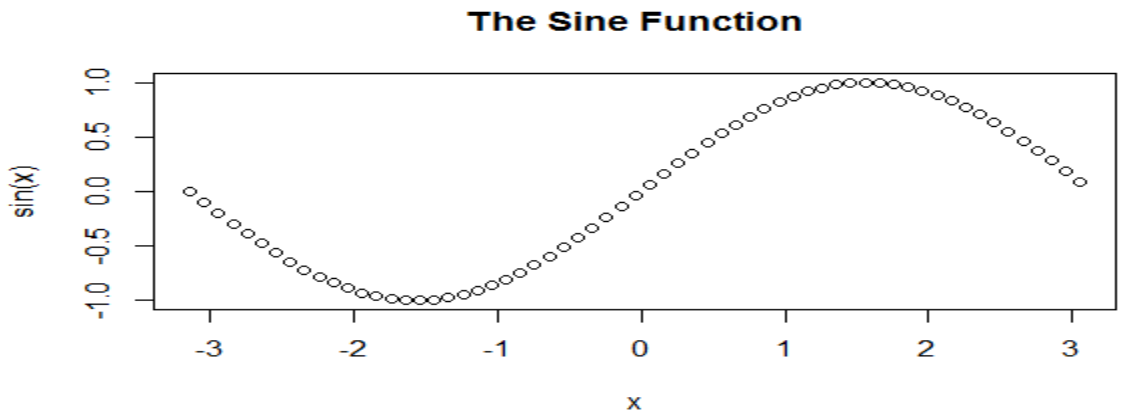
For example, the command plot(c(1,2),c(3,5)) would plot the points (1,3) and (2,5).

Adding shapes to Graphs:

- **Adding Titles and Labeling Axes**

We can add a title to our plot with the parameter **main.** similarly, **xlab** and **ylab** can be used to label the x-axis and y-axis respectively.

```
plot(x, sin(x),
main="The Sine Function",
ylab="sin(x)")
```



- **Changing Color and Plot Type**

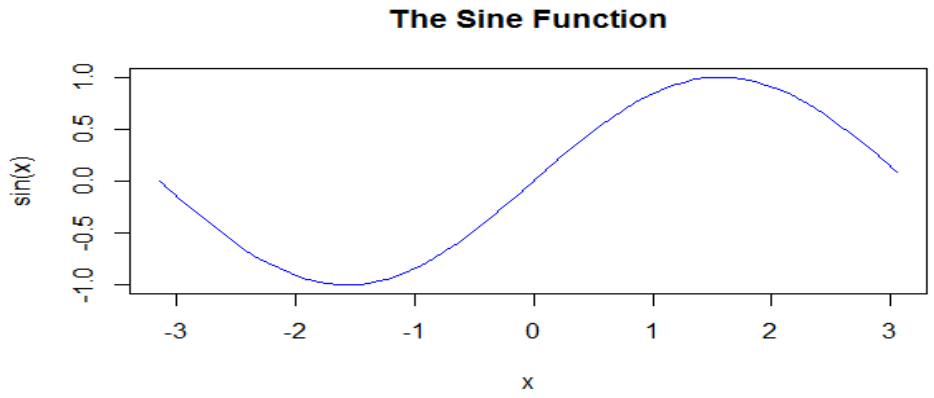
We can see above that the plot is of circular points and black in color. This is the default color. We can change the plot type with the argument type. It accepts the following strings and has the given effect.

```
"p" - points
"l" - lines
"b" - both points and lines
"c" - empty points joined by lines
"o" - over plotted points and lines
"s" and "S" - stair steps
"h" - histogram-like vertical lines
```

"n" - does not produce any points or lines

Ex:

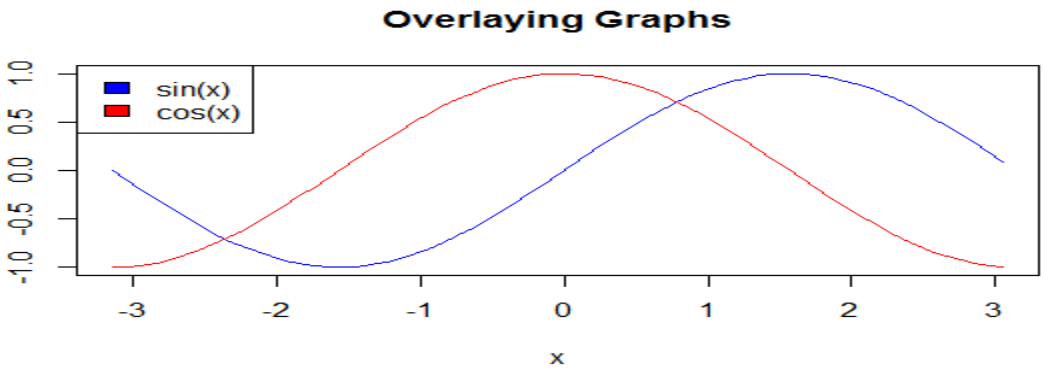
```
plot(x, sin(x),
main="The Sine Function",
ylab="sin(x)",
type="l",
col="blue")
```



- **Overlaying Plots Using legend () and lines () functions:**

Calling plot() multiple times will have the effect of plotting the current graph on the same window replacing the previous one. However, sometimes we wish to overlay the plots in order to compare the results. This is made possible with the functions lines() and points() to add lines and points respectively, to the existing plot.

```
plot(x, sin(x),
main="Overlaying Graphs",
ylab="",
type="l",
col="blue")
lines(x,cos(x), col="red")
legend("topleft",
c("sin(x)","cos(x)"),
fill=c("blue","red")
)
```



➢ **Adding Other Shapes to a Plot**

Using the following functions, we can add the extra graphical objects in plots:

- **rect –** For plotting rectangles – rect(xleft, ybottom, xright, ytop)

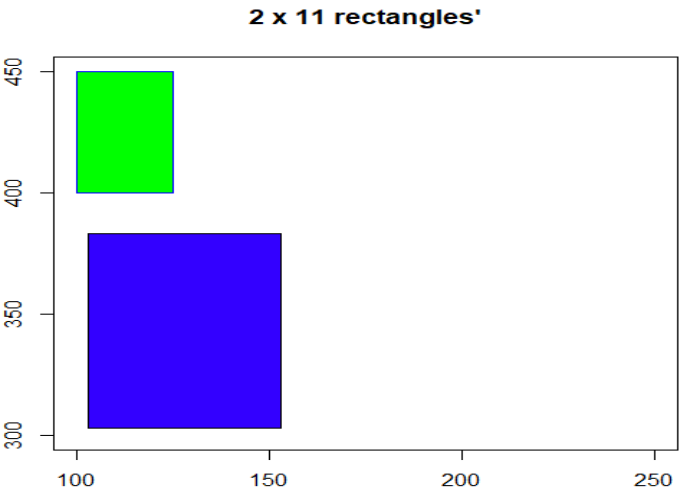
Ex: > plot(c(100, 250), c(300, 450), type = "n", xlab = "", ylab = "",

+ main = "2 x 11 rectangles'")

> rect(100+i, 300+i, 150+i, 380+i, col = rainbow(11, start = 0.7, end = 0.1))

> rect(100, 400, 125, 450, col = "green", border = "blue")

o/p:



Using the locater function, we can obtain the coordinates of the corners of the rectangle. But the rect function does not accept locator as its argument.

- **arrows –** For plotting arrows and headed bars – The syntax for the arrows function is to draw a line from the point (xO, yO) to the point (x1, y1) with the arrowhead, by default, at the "second" end (x1, y1).

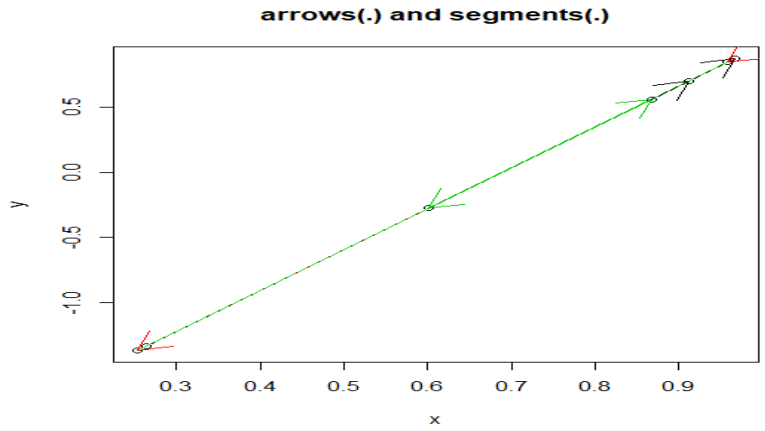
```
arrows(xO, yO, xl, yl)
```

Adding code=3 produces a horizontal double-headed arrow from (1,9) to (5,9), for example:

```
arrows(1,9,5,9,code=3)
```

Ex:

```
> plot(x,y, main = "arrows and segments")
> ## draw arrows from point to point:
> s <- seq(length(x)-1)          # one shorter than data
> arrows(x[s], y[s], x[s+1], y[s+1], col = 1:3)
```



- **polygon –** For plotting more complicated filled shapes, including objects with curved sides.

To draw a polygon in R, save the coordinates of six points in a vector called locations by using the following commands:

```
locations<-locator(6)
```

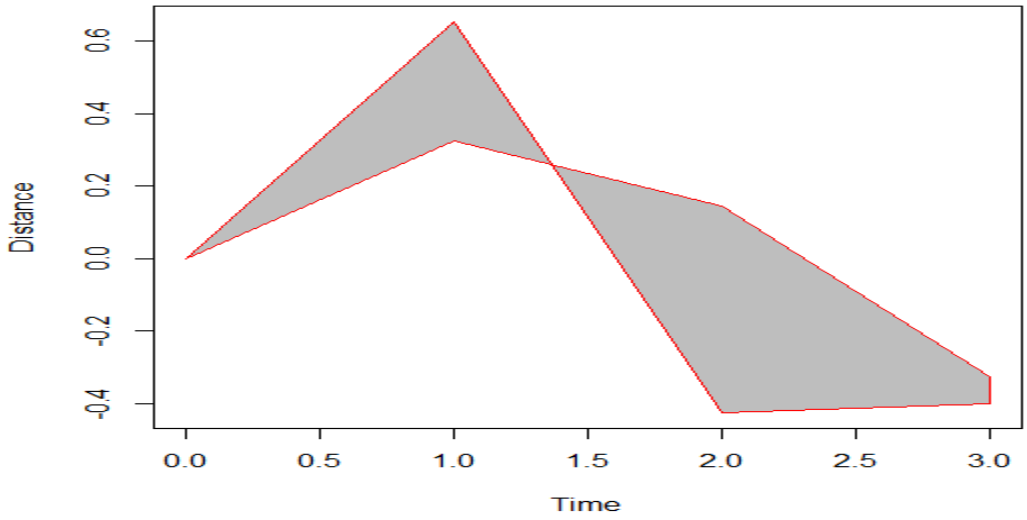
Now you can draw a lavender-colored polygon by using the following command:

```
polygon(locations,col=.lavender.)
```

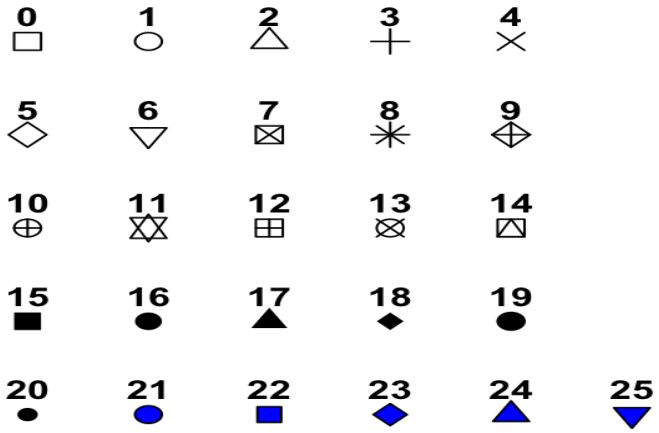
```
Ex: > xx <- c(0:n, n:0)
```

```
> yy <- c(c(0,cumsum(rnorm(n))), rev(c(0,cumsum(rnorm(n)))))
```

```
> plot   (xx, yy, type="n", xlab="Time", ylab="Distance")
> polygon(xx, yy, col="gray", border = "red")
```

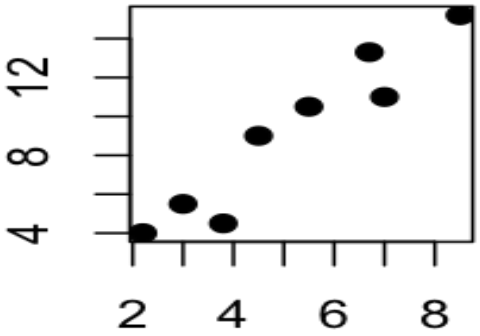


- Plot symbols in R: Different plotting symbols are available in R. The graphical argument used to specify point shapes is pch. By default **pch=1** . The different points symbols commonly used in R are shown in the figure below



Example:

```
x<-c(2.2, 3, 3.8, 4.5, 7, 8.5, 6.7, 5.5)
y<-c(4, 5.5, 4.5, 9, 11, 15.2, 13.3, 10.5)

# Plot points plot(x, y)
# Change plotting symbol # Use solid
circle
plot(x, y, pch = 19)
```



Saving Graphs to Files:
If you want to publish your results, you have to save your plot to a file in R and then import this graphics file into another document. Much of the time however, you may simply want to use R graphics in an interactive way to explore your data.
To save a plot to an image file, you have to do three things in sequence:

1. **Open a graphics device.**
   - The default graphics device in R is your computer screen. To save a plot to an image file, you need to tell R to open a new type of device — in this case, a graphics file of a specific type, such as PNG, PDF, or JPG.
   - The R function to create a PNG device is png(). Similarly, you create a PDF device with pdf() and a JPG device with jpg().
   - The first step in deciding how to save plots is to decide on the output format that you want to use. The following table lists some of the available formats, along with guidance as to when they may be useful.

Format	Driver	Notes
JPG	jpeg	Can be used anywhere, but doesn't resize
PNG	png	Can be used anywhere, but doesn't resize
WMF	win.metafile	Windows only; best choice with Word; easily resizable
PDF	pdf	Best choice with pdflatex; easily resizable
Postscript	postscript	Best choice with latex and Open Office; easily resizable

2. **Create the plot.**

Methods to Save Graphs to Files in R
Below, are the methods to Save Graphs to Files in R

i. A General Method

Here's a general method that will work on any computer with R, regardless of operating system or the way that we are connecting.
For Example:
If we have to save a plot as a JPG file, so we will use the jpeg driver. If we want to save a jpg file called "rplot.jpg" containing a plot of x and y, we would type the following commands:

- Save as Jpeg image

```
>jpeg('rplot.jpg')
> plot(x,y)
> dev.off()
```

- **Save as png image**

```
>png(file="C:/Datamentor/R-tutorial/saving_plot2.png", width=600, height=350)
>hist(Temperature, col="gold")
>dev.off()
```

ii. Another Approach

In R, the dev.copy command is used to copy the contents of the graph window to a file without having to re-enter the commands.
For Example:
To create a png file called myplot.png from a graph that is displayed by R, type
> dev.copy(png,'myplot.png')
> dev.off()

3. **Close the graphics device**.
   You do this with the dev.off() function.
   Put this in action by saving a plot of faithful to the home folder on your computer. First set your working directory to your home folder (or to any other folder you prefer
   Now you can check your file system to see whether the file faithful.png exists. (It should!) The result is a graphics file of type PNG that you can insert into a presentation, document, or website.

To save a plot as jpeg image we would perform the following steps. Please note that we need to call the function dev.off() after all the plotting, to save the file and return control to the screen.

```
Ex:   jpeg(file="saving_plot1.jpeg")
      hist(Temperature, col="darkgreen")
      dev.off()
```

Descriptive Statistics:

Statistical analysis in R is performed by using many in-built functions. Most of these functions are part of the R base package. These functions take R vector as an input along with the arguments and give the result.
The functions we are discussing in this chapter are mean, median and mode.

Mean

It is calculated by taking the sum of the values and dividing with the number of values in a data series.
The function **mean()** is used to calculate this in R.
Syntax:

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

Following is the description of the parameters used −

- **x** is the input vector.
- **trim** is used to drop some observations from both end of the sorted vector.
- **na.rm** is used to remove the missing values from the input vector.

Ex1 : x <- c(12,7,3,4.2,18,2,54,-21,8,-5)
# Find Mean.
result.mean <- mean(x)
print(result.mean)
When we execute the above code, it produces the following result −
[1] 5.55
Ex2: x <- c(12,7,3,4.2,18,2,54,-21,8,-5,NA)
# Find mean.
result.mean <- mean(x)
print(result.mean)
# Find mean dropping NA values.
result.mean <- mean(x,na.rm = TRUE)
print(result.mean)
When we execute the above code, it produces the following result −
[1] NA
[1] 8.22

Median:
The middle most value in a data series is called the median. The median()function is used in R to calculate this value.
Syntax:
median(x, na.rm = FALSE)
Following is the description of the parameters used −
x is the input vector.
na.rm is used to remove the missing values from the input vector.
# Create the vector.
x <- c(12,7,3,4.2,18,2,54,-21,8,-5)

# Find the median.
median.result <- median(x)
print(median.result)
When we execute the above code, it produces the following result −
[1] 5.6

Variance: How far a set of data values are spread out from their mean. Calculating variance in R is simplicity itself. You use the var() function. The **variance** is a numerical measure of how the data values is dispersed around the mean. In particular, the **sample variance** is defined as:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Ex:
x <- c(1,2,3,4,5,1,2,3,1,2,4,5,2,3,1,1,2,3,5,6) # our data set
variance.result = var(x) # calculate variance
> print (variance.result)
[1] 2.484211

Standard Deviation: A measure that is used to quantify the amount of variation or dispersion of a set of data values. Standard deviations are calculated in the same way as means. The standard deviation of a single variable can be computed with the sd(VAR) command, where VAR is the name of the variable whose standard deviation you wish to retrieve.

Ex:

```
x <- c(1,2,3,4,5,1,2,3,1,2,4,5,2,3,1,1,2,3,5,6)          # our data set
> sd.result = sqrt(var(x)) # calculate standard deviation
> print (sd.result)
[1] 1.576138
```

Minimum and Maximum

Keeping with the pattern, a minimum can be computed on a single variable using the min(VAR) command. The maximum, via max(VAR), operates identically. However, in contrast to the mean and standard deviation functions, min(DATAVAR) or max(DATAVAR) will retrieve the minimum or maximum value from the entire dataset, not from each individual variable. Therefore, it is recommended that minimums and maximums be calculated on individual variables, rather than entire datasets, in order to produce more useful information. The sample code below demonstrates the use of the min and max functions.

Ex:

```
x <- c(1,2,3,4,5,1,2,3,1,2,4,5,2,3,1,1,2,3,5,6)          # our data set
> y=min(x)
> y
[1] 1
> z=max(x)
> z
[1] 6
```

Correlation and lines of Regression:

Regression analysis is a very widely used statistical tool to establish a relationship model between two variables. One of these variable is called predictor variable whose value is gathered through experiments. The other variable is called response variable whose value is derived from the predictor variable.

Linear Regression

Linear regression is one of the most commonly used predictive modelling techniques. The aim of linear regression is to find a mathematical equation for a continuous response variable Y as a function of one or more X variable(s). So that you can use this regression model to predict the Y when only the X is known.

Mathematically a linear relationship represents a straight line when plotted as a graph. A non-linear relationship where the exponent of any variable is not equal to 1 creates a curve.

The general mathematical equation for a linear regression is −

```
y = ax + b
```

Following is the description of the parameters used −

- **y** is the response variable.
- **x** is the predictor variable.
- **a** and **b** are constants which are called the coefficients.

Steps to Establish a Regression

A simple example of regression is predicting weight of a person when his height is known. To do this we need to have the relationship between height and weight of a person.

The steps to create the relationship is −

- Carry out the experiment of gathering a sample of observed values of height and corresponding weight.
- Create a relationship model using the **lm()** functions in R.
- Find the coefficients from the model created and create the mathematical equation using these
- Get a summary of the relationship model to know the average error in prediction. Also called **residuals**.
- To predict the weight of new persons, use the **predict()** function in R.

Input Data

Below is the sample data representing the observations −

```
# Values of height
151, 174, 138, 186, 128, 136, 179, 163, 152, 131

# Values of weight.
63, 81, 56, 91, 47, 57, 76, 72, 62, 48
```

lm() Function

This function creates the relationship model between the predictor and the response variable.

Syntax

The basic syntax for **lm()** function in linear regression is −

```
lm(formula,data)
```

Following is the description of the parameters used −

- **formula** is a symbol presenting the relation between x and y.
- **data** is the vector on which the formula will be applied.

Create Relationship Model & get the Coefficients

```
x <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)

y <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)

# Apply the lm() function.

relation <- lm(y~x)

print(relation)
```

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

```
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)          x
   -38.4551       0.6746
```

Nonlinear Regression:

Regression is nonlinear when at least one of its parameters appears nonlinearly. It commonly sorts and analyzes data of various industries like retail and banking sectors. It also helps to draw conclusions and predict future trends on the basis of user's activities on the net.

In non-linear regression the analyst specify a function with a set of parameters to fit to the data. The most basic way to estimate such parameters is to use a non-linear least squares approach (function nls in R) which basically approximate the non-linear function using a linear one and iteratively try to find the best parameter values (wiki). A nice feature of non-linear regression in an applied context is that the estimated parameters have a clear interpretation (Vmax in a Michaelis-Menten model is the maximum rate) which would be harder to get using linear models on transformed data

```
#simulate some data
set.seed(20160227)

x<-seq(0,50,1)
y<-((runif(1,10,20)*x)/(runif(1,0,10)+x))+rnorm(51,0,1)

#for simple models nls find good starting values for the parameters even if it throw a warning
m<-nls(y~a*x/(b+x))

#get some estimation of goodness of fit
cor(y,predict(m))
```

[1] 0.9496598

Multiple regression:

Multiple regression is an extension of linear regression into relationship between more than two variables. In simple linear relation we have one predictor and one response variable, but in multiple regression we have more than one predictor variable and one response variable.

The general mathematical equation for multiple regression is −

```
y = a + b1x1 + b2x2 +...bnxn
```

Following is the description of the parameters used −

- **y** is the response variable.
- **a, b1, b2...bn** are the coefficients.
- **x1, x2, ...xn** are the predictor variables.

We create the regression model using the **lm()** function in R. The model determines the value of the coefficients using the input data. Next we can predict the value of the response variable for a given set of predictor variables using these coefficients.

lm() Function

This function creates the relationship model between the predictor and the response variable.

Syntax

The basic syntax for **lm()** function in multiple regression is −

```
lm(y ~ x1+x2+x3...,data)
```

Following is the description of the parameters used −

- **formula** is a symbol presenting the relation between the response variable and predictor variables.
- **data** is the vector on which the formula will be applied.

Example

Input Data

Consider the data set "mtcars" available in the R environment. It gives a comparison between different car models in terms of mileage per gallon (mpg), cylinder displacement("disp"), horse power("hp"), weight of the car("wt") and some more parameters.

The goal of the model is to establish the relationship between "mpg" as a response variable with "disp","hp" and "wt" as predictor variables. We create a subset of these variables from the mtcars data set for this purpose.

Live Demo

```
input <- mtcars[,c("mpg","disp","hp","wt")]

print(head(input))
```

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

```
                   mpg   disp   hp    wt
Mazda RX4         21.0  160    110   2.620
Mazda RX4 Wag     21.0  160    110   2.875
Datsun 710        22.8  108     93   2.320
Hornet 4 Drive    21.4  258    110   3.215
Hornet Sportabout 18.7  360    175   3.440
Valiant           18.1  225    105   3.460
```

Logistic Regression

The Logistic Regression is a regression model in which the response variable (dependent variable) has categorical values such as True/False or 0/1. It actually measures the probability of a binary response as the value of response variable based on the mathematical equation relating it with the predictor variables.

The general mathematical equation for logistic regression is −

```
y = 1/(1+e^-(a+b1x1+b2x2+b3x3+...))
```

Following is the description of the parameters used −

- **y** is the response variable.
- **x** is the predictor variable.
- **a** and **b** are the coefficients which are numeric constants.

The function used to create the regression model is the **glm()** function.

Syntax

The basic syntax for **glm()** function in logistic regression is −

```
glm(formula,data,family)
```

Following is the description of the parameters used −

- **formula** is the symbol presenting the relationship between the variables.
- **data** is the data set giving the values of these variables.
- **family** is R object to specify the details of the model. It's value is binomial for logistic regression.

Example

The in-built data set "mtcars" describes different models of a car with their various engine specifications. In "mtcars" data set, the transmission mode (automatic or manual) is described by the column am which is a binary value (0 or 1). We can create a logistic regression model between the columns "am" and 3 other columns - hp, wt and cyl.

```
# Select some columns form mtcars.

input <- mtcars[,c("am","cyl","hp","wt")]



print(head(input))
```

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

```
                  am   cyl  hp    wt
Mazda RX4          1   6    110   2.620
Mazda RX4 Wag      1   6    110   2.875
Datsun 710         1   4     93   2.320
Hornet 4 Drive     0   6    110   3.215
Hornet Sportabout  0   8    175   3.440
Valiant            0   6    105   3.460
```

Create Regression Model

We use the **glm()** function to create the regression model and get its summary for analysis.

Live Demo

```
input <- mtcars[,c("am","cyl","hp","wt")]



am.data = glm(formula = am ~ cyl + hp + wt, data = input, family = binomial)



print(summary(am.data))
```

Time Series Analysis

Time series is a series of data points in which each data point is associated with a timestamp. A simple example is the price of a stock in the stock market at different points of time on a given day. Another example is the amount of rainfall in a region at different months of the year. R language uses many functions to create, manipulate and plot the time series data. The data for the time series is stored in an R object called **time-series object**. It is also a R data object like a vector or data frame.

The time series object is created by using the **ts()** function.

Syntax

The basic syntax for **ts()** function in time series analysis is −

```
timeseries.object.name <-  ts(data, start, end, frequency)
```

Following is the description of the parameters used −

- **data** is a vector or matrix containing the values used in the time series.
- **start** specifies the start time for the first observation in time series.
- **end** specifies the end time for the last observation in time series.
- **frequency** specifies the number of observations per unit time.

Except the parameter "data" all other parameters are optional.

Example

Consider the annual rainfall details at a place starting from January 2012. We create an R time series object for a period of 12 months and plot it.

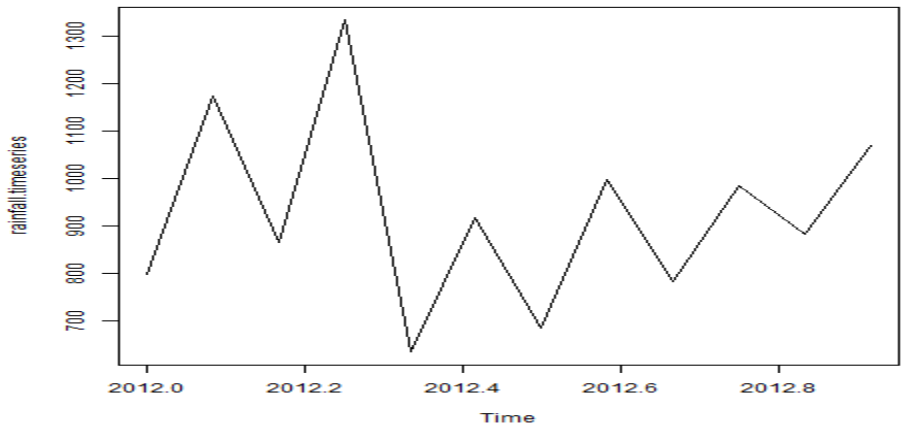
```
# Get the data points in form of a R vector.

rainfall <- c(799,1174.8,865.1,1334.6,635.4,918.5,685.5,998.6,784.2,985,882.8,1071)



# Convert it to a time series object.

rainfall.timeseries <- ts(rainfall,start = c(2012,1),frequency = 12)

# Print the timeseries data.

print(rainfall.timeseries)

# Give the chart file a name.

png(file = "rainfall.png")

# Plot a graph of the time series.

plot(rainfall.timeseries)

# Save the file.

dev.off()
```

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

```
Jan   Feb   Mar   Apr   May   Jun   Jul   Aug   Sep
```

```
2012  799.0  1174.8  865.1  1334.6  635.4  918.5  685.5  998.6  784.2
      Oct    Nov    Dec
2012  985.0  882.8 1071.0
```

The Time series chart −



Different Time Intervals

The value of the **frequency** parameter in the ts() function decides the time intervals at which the data points are measured. A value of 12 indicates that the time series is for 12 months. Other values and its meaning is as below −

- **frequency = 12** pegs the data points for every month of a year.
- **frequency = 4** pegs the data points for every quarter of a year.
- **frequency = 6** pegs the data points for every 10 minutes of an hour.
- **frequency = 24*6** pegs the data points for every 10 minutes of a day.

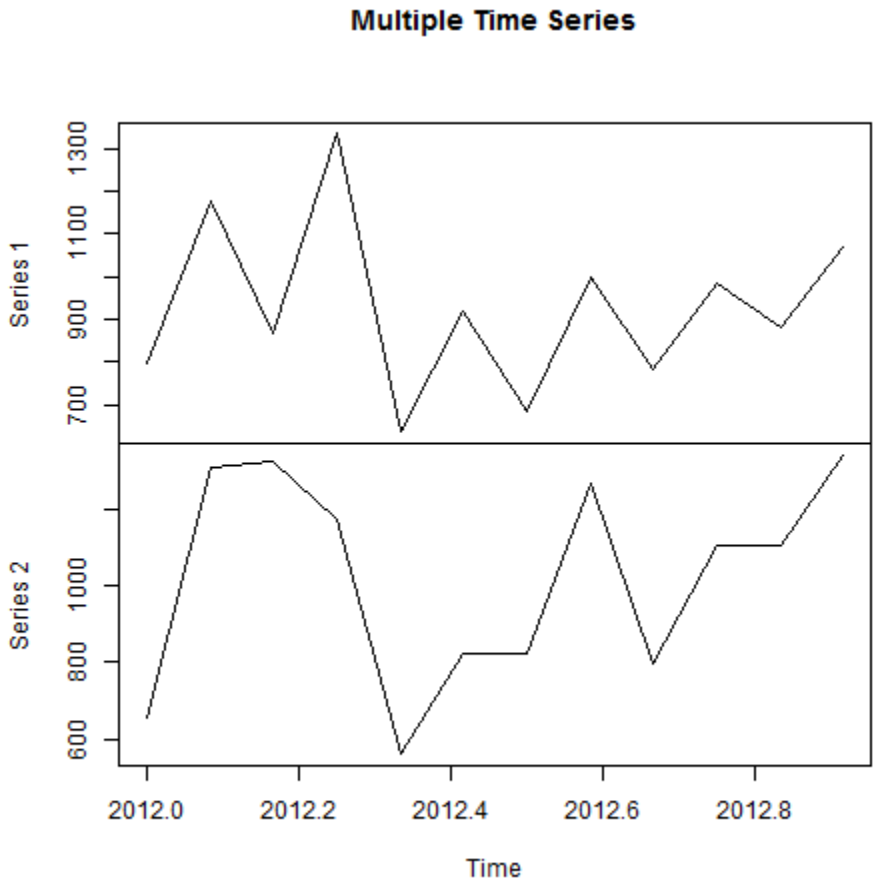
Multiple Time Series

We can plot multiple time series in one chart by combining both the series into a matrix.

```
# Get the data points in form of a R vector.

rainfall1 <- c(799,1174.8,865.1,1334.6,635.4,918.5,685.5,998.6,784.2,985,882.8,1071)

rainfall2 <-

        c(655,1306.9,1323.4,1172.2,562.2,824,822.4,1265.5,799.6,1105.6,1106.7,1337.8)

# Convert them to a matrix.

combined.rainfall <-  matrix(c(rainfall1,rainfall2),nrow = 12)

# Convert it to a time series object.

rainfall.timeseries <- ts(combined.rainfall,start = c(2012,1),frequency = 12)

# Print the timeseries data.

print(rainfall.timeseries)
```

```
# Give the chart file a name.
png(file = "rainfall_combined.png")
# Plot a graph of the time series.
plot(rainfall.timeseries, main = "Multiple Time Series")
# Save the file.
dev.off()
```

The Multiple Time series chart −



You can customize many features of your graphs (fonts, colors, axes, titles) through graphic options.
One way is to specify these options in through the **par( )** function. If you set parameter values here, the changes will be in effect for the rest of the session or until you change them again. The format is **par(*optionname*=*value*, *optionname*=*value*, ...)**

```
# Set a graphical parameter using par()

par()             # view current settings
opar <- par()     # make a copy of current settings
par(col.lab="red") # red x and y labels
hist(mtcars$mpg)  # create a plot with these new settings
par(opar)         # restore original settings
```

A second way to specify graphical parameters is by providing the ***optionname***=*value* pairs directly to a high level plotting function. In this case, the options are only in effect for that specific graph.

```
# Set a graphical parameter within the plotting function
hist(mtcars$mpg, col.lab="red")
```

See the help for a specific high level plotting function (e.g. plot, hist, boxplot) to determine which graphical parameters can be set this way.

The remainder of this section describes some of the more important graphical parameters that you can set.

Text and Symbol Size

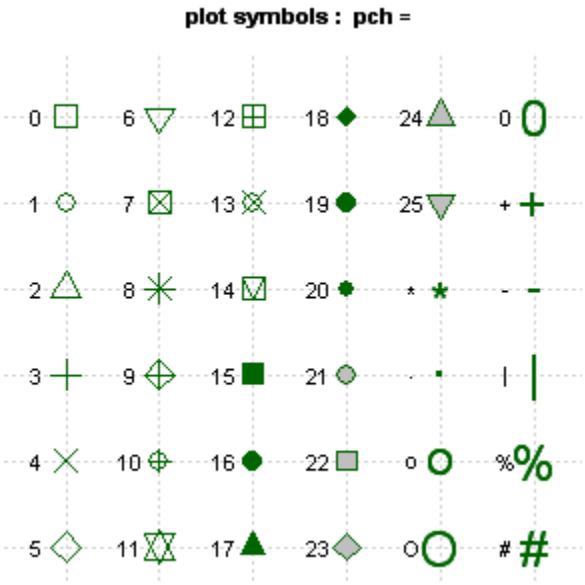
The following options can be used to control text and symbol size in graphs.

option	description
cex	number indicating the amount by which plotting text and symbols should be scaled relative to the default. 1=default, 1.5 is 50% larger, 0.5 is 50% smaller, etc.
cex.axis	magnification of axis annotation relative to cex
cex.lab	magnification of x and y labels relative to cex

cex.main	magnification of titles relative to cex
cex.sub	magnification of subtitles relative to cex

Plotting Symbols

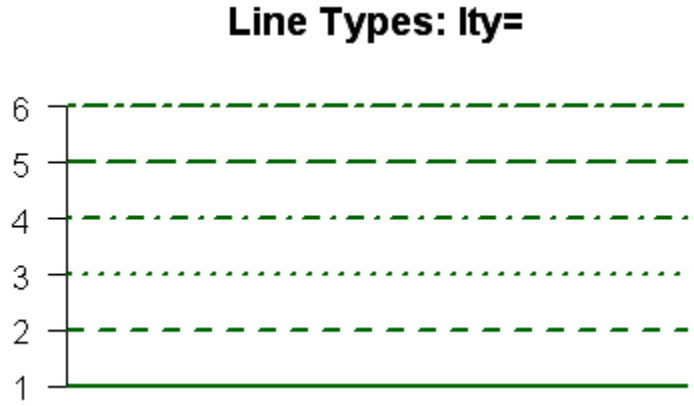
Use the **pch=** option to specify symbols to use when plotting points. For symbols 21 through 25, specify border color (col=) and fill color (bg=).



Lines

You can change lines using the following options. This is particularly useful for reference lines, axes, and fit lines.

option	description
lty	line type. see the chart below.
lwd	line width relative to the default (default=1). 2 is twice as wide.

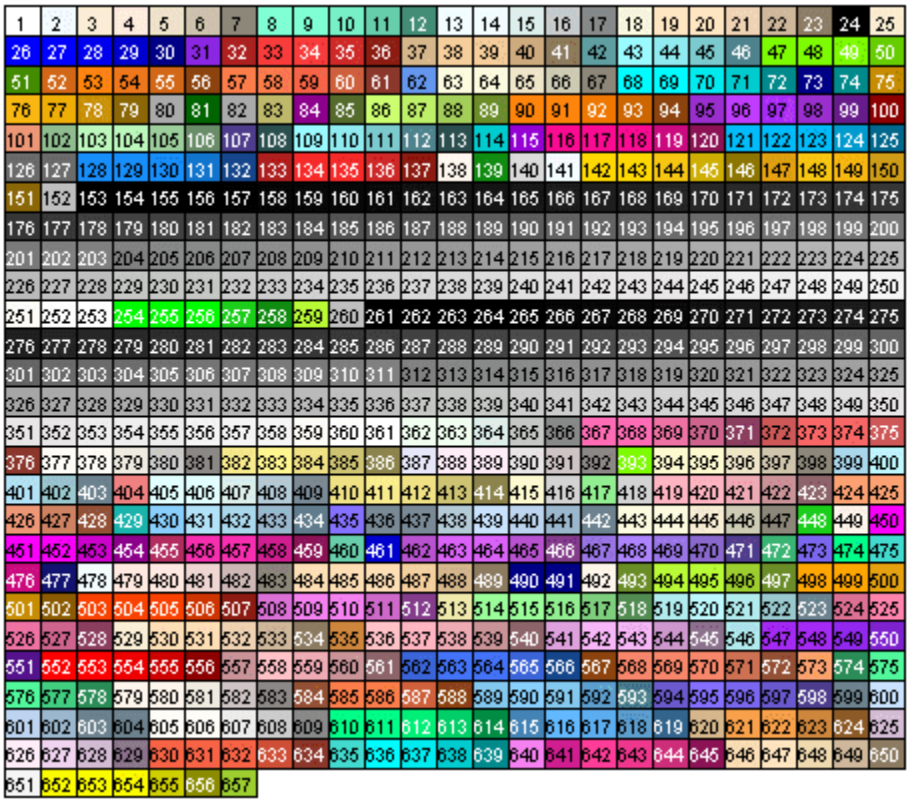


Colors

Options that specify colors include the following.

option	description
col	Default plotting color. Some functions (e.g. lines) accept a vector of values that are recycled.
col.axis	color for axis annotation
col.lab	color for x and y labels
col.main	color for titles
col.sub	color for subtitles
fg	plot foreground color (axes, boxes - also sets col= to same)
bg	plot background color

You can specify colors in R by index, name, hexadecimal, or RGB.
For example **col=1**, **col="white"**, and **col="#FFFFFF"** are equivalent.
The following chart was produced with code developed by Earl F. Glynn. See his Color Chart for all the details you would ever need about using colors in R.



You can also create a vector of *n* contiguous colors using the functions **rainbow(*n*)**, **heat.colors(*n*)**, **terrain.colors(*n*)**, **topo.colors(*n*)**, and **cm.colors(*n*)**. **colors()** returns all available color names.

Fonts

You can easily set font size and style, but font family is a bit more complicated.

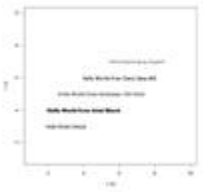
option	description
font	Integer specifying font to use for text. 1=plain, 2=bold, 3=italic, 4=bold italic, 5=symbol
font.axis	font for axis annotation
font.lab	font for x and y labels
font.main	font for titles
font.sub	font for subtitles
ps	font point size (roughly 1/72 inch) text size=ps*cex
family	font family for drawing text. Standard values are "serif", "sans", "mono", "symbol". Mapping is device dependent.

In windows, mono is mapped to "TT Courier New", serif is mapped to"TT Times New Roman", sans is mapped to "TT Arial", mono is mapped to "TT Courier New", and symbol is mapped to "TT Symbol" (TT=True Type). You can add your own mappings.

```
# Type family examples - creating new mappings
plot(1:10,1:10,type="n")
windowsFonts(
  A=windowsFont("Arial Black"),
  B=windowsFont("Bookman Old Style"),
  C=windowsFont("Comic Sans MS"),
  D=windowsFont("Symbol")
)
text(3,3,"Hello World Default")
text(4,4,family="A","Hello World from Arial Black")
text(5,5,family="B","Hello World from Bookman Old Style")
```

```
text(6,6,family="C","Hello World from Comic Sans MS")
```

```
text(7,7,family="D", "Hello World from Symbol")
```

 click to view

Margins and Graph Size

You can control the margin size using the following parameters.

option	description
mar	numerical vector indicating margin size c(bottom, left, top, right) in lines. default = c(5, 4, 4, 2) + 0.1
mai	numerical vector indicating margin size c(bottom, left, top, right) in inches
pin	plot dimensions (width, height) in inches

For complete information on margins, see Earl F. Glynn's margin tutorial.

UNIT-V

Measures of Central Tendency: Mean, Median and Mode

The Mean: A Distribution's Numerical Average

The ***mean*** is simply a numerical average. It can be found by adding up each value assigned to an interval-ratio variable and dividing the sum by the total number of cases.

Example 1: We surveyed 5 people, asking each respondent their age (in years). The ages reported in our survey were: 21, 45, 24, 78, 45. Find the mean.

- (21 + 45 + 24 + 78 + 45) / (5) = 42.6

To find the mean, we added the ages of each respondent, then divided by the total number of people in our study (5). The mean for age is 42.6 years.

Example 2:

```
> x=c(5,2,6,9,3,5,2,2)
> x
[1] 5 2 6 9 3 5 2 2
> mean(x)
[1] 4.25
```

The Median: The centre Value

The ***median*** is the value that lies in the centre of the distribution. When the data are ordered from least to greatest, the median is located in the middle of the list. The median can be found for both numbers *and* ranked categories. It is first necessary to order your values from least to greatest. If there is only one center value (there are an equal number of cases above and below), great, you've found the median! If there are two center values (this will happen when there is an odd number of cases), the median is found by taking the average of the two center values.

Example 1: We surveyed 5 people, asking each respondent their age (in years). The ages reported in our survey were: 21, 45, 24, 78, 45. Find the median.

- We must first rearrange the values for age from least to greatest: 21, 24, 45, 45, 78
- We then identify the value(s) in the centre: 21, 24, **45**, 45, 78
- ***Answer:*** The median is 45

Example 2:

```
> x
[1] 5 2 6 9 3 5 2 2
> median(x)
[1] 4
```

The Mode: The Most Frequently Occurring Value

The **mode** is the value that occurs most frequently. It is found by determining the number or category that appears most often. If no value occurs more than once, there is no mode. If there are two values that occur most often, report both of them--this type of distribution is bimodal.

Example 1: We surveyed 5 people, asking each respondent their age (in years). The ages reported in our survey were: 21, 45, 24, 78, 45. Find the mode.

- We see in the following distribution (21, **45,** 24, 78, **45**) that 45 occurs twice, whereas the other ages occur only once. Therefore, 25 is the mode for age

Measures of Central Tendency:

Central Tendency Measures		
Measure	**Formula**	**Description**
Mean	∑x/n	Balance Point
Median	n+1/2 Position	Middle Value when ordered
Mode	None	Most frequent

MEASURES OF DISPERSION

Methods of Dispersion
Methods of studying dispersion are divided into two types:

(i) Mathematical Methods: We can study the 'degree' and 'extent' of variation by these methods. In this category, commonly used measures of Dispersion are:
(a)Range
(b)Quartile Deviation
(c)Average Deviation
(d) Standard deviation and coefficient of variation.

(ii) Graphic Methods: Where we want to study only the extent of variation, whether it is higher or lesser a Lorenz-curve is used.
(a)Range:
It is the simplest method of studying dispersion. Range is the difference between the smallest value and the largest value of a series. While computing range, we do not take into account frequencies of different groups.
Formula: Absolute Range = L – S
where, L represents largest value in a distribution
S represents smallest value in a distribution
We can understand the computation of range with the help of examples of different series.

The range is the absolute measure of dispersion. It cannot be used to compare two distributions with different units. So the relative measures corresponding to the range known as coefficient of range is defined by

Coefficient of range = L – S / L + S

(i) Raw Data: Marks out of 50 in a subject of 12 students, in a class are given as follows:
12, 18, 20, 12, 16, 14, 30, 32, 28, 12, 12 and 35.
In the example, the maximum or the highest marks obtained by a candidate is '35' and the lowest marks obtained by a candidate are '12'. Therefore, we can calculate range;
L = 35 and S = 12
Absolute Range = L – S = 35 – 12 = 23 marks
Coefficient of Range =L-S/L+S

(ii) Discrete Series

Marks of the Students in Statistics (out of 50) (X)	**No. of students (f)**
Smallest 10	4
12	10
18	16
Largest 20	15
	Total = 45

Absolute Range = 20 – 10 = 10 marks

(iii) Continuous Series

	X	**Frequencies**
	10 – 15	4
S = 10	15 – 20	10
L = 30	20 – 25	26
	25 – 30	8

Absolute Range = L – S = 30 – 10 = 20 marks

Range is a simplest method of studying dispersion. It takes lesser time to compute the 'absolute' and 'relative' range. Range does not take into account all the values of a series, i.e. it considers only the extreme items and middle items are not given any importance. Therefore, Range cannot tell us anything about the character of the distribution. Range cannot be computed in the case of "open ends' distribution i.e., a distribution where the lower limit of the first group and upper limit of the higher group is not given. The concept of range is useful in the field of quality control and to study the variations in the prices of the shares etc.

(b) Quartile Deviations (Q.D.)
The concept of 'Quartile Deviation does take into account only the values of the 'Upper quartile

(Q3) and the 'Lower quartile' (Q1). Quartile Deviation is also called 'inter-quartile range'. It is a better method when we are interested in knowing the range within which certain proportion of the items fall.
Before taking up the quartile deviation, we must know the meaning of quarters and quartiles.

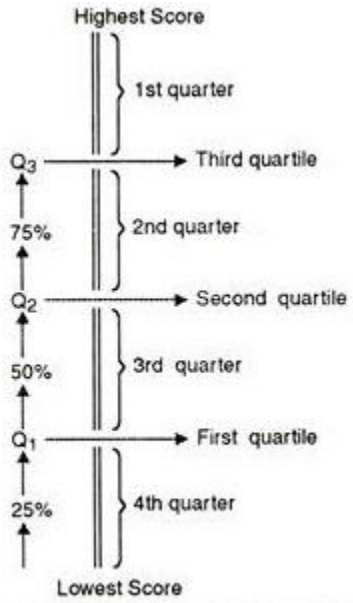


Fig. 4.2 Diagram showing quarters and quartiles.

- The **first quartile,** denoted by $\boldsymbol{Q_1}$, is the median of the *lower half* of the data set. This means that about 25% of the numbers in the data set lie below Q_1 and about 75% lie above Q_1 .
- The **third quartile,** denoted by $\boldsymbol{Q_3}$, is the median of the *upper half* of the data set. This means that about 75% of the numbers in the data set lie below Q_3 and about 25% lie above Q_3 .

For example a test results 20 scores and these scores are arranged in a descending order. Let us divide the distribution of scores into four equal parts. Each part will present a 'quarter'. In each quarter there will be 25% (or 1/4th of N) cases.
As scores are arranged in descending order,

- ✓ The top 5 scores will be in the 1st quarter,
- ✓ The next 5 scores will be in the 2nd quarter,
- ✓ The next 5 scores will be in the 3rd quarter, and
- ✓ And the lowest 5 scores will be in the 4th quarter.

'Quartile Deviation' can be obtained as:
(i) Inter-quartile range = Q3 – Q1
Eample:
Quartile Deviation in case of Raw Data
Suppose the values of X are : 20, 12, 18, 25, 32, 10
In case of quartile-deviation, it is necessary to calculate the values of Q1 and Q3 by arranging the given data in ascending of descending order.
Therefore, the arranged data are (in ascending order):
X = 10, 12, 18, 20, 25, 32

No. of items = 6
Q1 = the value of item = = 1.75th item
= the value of 1st item + 0.75 (value of 2nd item – value of 1st item)
= 10 + 0.75 (12 – 10) = 10 + 0.75(2) = 10 + 1.50 = 11.50
Q3 = the value of item =
= the value of 3(7/4)th item = the value of 5.25th item
= 25 + 0.25 (32 – 25) = 25 + 0.25 (7) = 26.075
Therefore,
(i) Inter-quartile range(IQR) = Q3 – Q1 = 26.75 – 11.50 = 15.25

(ii) **Semi quartile deviation**: The difference Q3−Q1divided by 2 is called semi-inter quartile range or the quartile deviation.
Semi-quartile range =(Q3−Q1)/2
(iii) Coefficient of Quartile Deviation = (Q3−Q1)/(Q3+Q1)

Calculation of Q1 and Q3 (Another method) with example
Steps:
Step 1: Put the numbers in order.
1, 2, 5, 6, 7, 9, 12, 15, 18, 19, 27.
Step 2: Find the median.
1, 2, 5, 6, 7, 9, 12, 15, 18, 19, 27.
Step 3: Place parentheses around the numbers above and below the median.
Not necessary statistically, but it makes Q1 and Q3 easier to spot.
(1, 2, 5, 6, 7), 9, (12, 15, 18, 19, 27).
Step 4: Find Q1 and Q3
Think of Q1 as a median in the lower half of the data and think of Q3 as a median for the upper half of data.
(1, 2, 5, 6, 7), 9, (12, 15, 18, 19, 27). Q1 = 5 and Q3 = 18.
Step 5: Subtract Q1 from Q3 to find the interquartile range.
Q = 18 – 5 = 13.

Calculation of Q1 and A3 (In case of even set of numbers)
Sample question: Find the IQR for the following data set: 3, 5, 7, 8, 9, 11, 15, 16, 20, 21.
Step 1: Put the numbers in order.
3, 5, 7, 8, 9, 11, 15, 16, 20, 21.
Step 2: Make a mark in the center of the data:
3, 5, 7, 8, 9, | 11, 15, 16, 20, 21.
Step 3: Place parentheses around the numbers above and below the mark you made in Step 2–it makes Q1 and Q3 easier to spot.
(3, 5, 7, 8, 9), | (11, 15, 16, 20, 21).
Step 4: Find Q1 and Q3
Q1 is the median (the middle) of the lower half of the data, and Q3 is the median (the middle) of the upper half of the data.
(3, 5, 7, 8, 9), | (11, 15, 16, 20, 21). Q1 = 7 and Q3 = 16.

Step 5: Subtract Q1 from Q3.
16 – 7 = 9.
This is your IQR.

Calculation of Inter-quartile Range, semi-quartile Range and Coefficient of Quartile Deviation in discrete series

Suppose a series consists of the salaries (Rs.) and number of the workers in a factory:

Salaries (Rs.)	**No. of workers**
60	4
100	20
120	21
140	16
160	9

In the problem, we will first compute the values of Q3 and Q1

Salaries (Rs.) (x)	**No. of workers (f)**	**Cumulative frequencies (c.f.)**
60	4	4
100	20	24 – Q1 lies in this cumulative
120	21	45 frequency
140	16	61 – Q3 lies in this cumulative
160	9	70 frequency
	N = Sf = 70	

Calculation of Q1:
Q1 = size of th item
= size of th item = 17.75
17.75 lies in the cumulative frequency 24, which is corresponding to the value Rs. 100
Q1 = Rs. 100

Calculation of Q3 :
Q3 = size of th item
= size of th item = 53.25th item
53.25 lies in the cumulative frequency 61 which is corresponding to Rs. 140
Q3 = Rs. 140

(i) Inter-quartie range = Q3 – Q1 = Rs. 140 – Rs. 100 = Rs. 40
(ii) Semi-quartie range =
(iii) Coefficient of Quartile Deviation =

Calculation of Inter-quartile range, semi-quartile range and Coefficient of Quartile Deviation in case of continuous series

We are given the following data:

Salaries (Rs.)	**No. of Workers**

10 – 20 4
20 – 30 6
30 – 10 10
40 – 50 5

Total = 25
In this example, the values of Q3 and Q1 are obtained as follows:
Salaries (Rs.) No. of workers Cumulative frequencies
(x) (f) (c.f.)
10 – 20 4 4
20 – 30 6 10
30 – 40 10 20
40 – 50 5 25
N = 25
Q1 =
Therefore, . It lies in the cumulative frequency 10, which is corresponding to class 20 – 30.
Therefore, Q1 group is 20 – 30.
where, l1 = 20, f = 6, i = 10, and cfo = 4
Q1 =
Q3 =
Therefore, = 18.75, which lies in the cumulative frequency 20, which is corresponding to class 30 –40, Therefore Q3 group is 30 – 40.
where, l1 = 30, i = 10, cf0 = 10, and f = 10
Q3 = = Rs. 38.75
Therefore :
(i) Inter-quartile range = Q3 – Ql = Rs. 38.75 – Rs. 23.75 = Rs.15.00
(iii) Semi-quartile range =
(iii) Coefficient of Quartile Deviation

(c) Average Deviation

Average deviation is defined as a value which is obtained by taking the average of the deviations of various items from a measure of central tendency Mean or Median or Mode, ignoring negative signs. Generally, the measure of central tendency from which the deviations arc taken is specified in the problem. If nothing is mentioned regarding the measure of central tendency specified than deviations are taken from median because the sum of the deviations (after ignoring negative signs) is minimum.

Computation in case of raw data

Absolute Deviation: The **absolute deviation** is the **absolute difference** between each **value** of the **statistical variable** and the **arithmetic mean**.

$$D_i = |x - x|$$

Average Deviation: The **average deviation** is the **arithmetic mean** of the **absolute deviations**.

The **average deviation** is represented by D_x

$$D_{\bar{x}} = \frac{|x_1 - \bar{x}| + |x_2 - \bar{x}| + \ldots + |x_n - \bar{x}|}{N}$$

$$D_{\bar{x}} = \frac{\sum_{i=1}^{n} |x_i - \bar{x}|}{N}$$

Steps to Compute Average Deviation :
(i) Calculate the value of Mean or Median or Mode
(ii) Take deviations from the given measure of central-tendency and they are shown as d.
(iii) Ignore the negative signs of the deviation that can be shown as \d\ and add them to find S|d|.
(iv) Apply the formula to get Average Deviation about Mean or Median or Mode.

Example: Find the average deviation of the following set of numbers:
3, 8, 8, 8, 8, 9, 9, 9, 9.
Step 1: Find the mean:
(3 + 8 + 8 + 8 + 8 + 9 + 9 + 9 + 9) = 71.9 = 7.89.
$\bar{x}$ = 7.89
Step 2: Find each individual absolute deviation using the formula |x – $\bar{x}$|.
|3 – 7.89| = 4.89
|8 – 7.89| = 0.11
|8 – 7.89| = 0.11
|8 – 7.89| = 0.11
|8 – 7.89| = 0.11
|9 – 7.89| = 1.11
|9 – 7.89| = 1.11
|9 – 7.89| = 1.11
|9 – 7.89| = 1.11
Step 3: Add up all of the values you found in Step 1.
4.89+0.11+0.11+0.11+0.11+1.11+1.11+1.11+1.11= 9.77
Step 4: Divide by the number of items in your data set. There are 9 items, so:
9.77/9 = 1.09.
The average deviation is 1.09.
Example 2:
Question 1: Calculate the average deviation for the given data:
2, 4, 6, 8, 10
Solution:

Given:
n = 5
First lets find the mean by using the formula,
$\bar{x} = \sum xn$
$\bar{x}$ = 2+4+6+8+1052+4+6+8+105
$\bar{x}$ =305
∴Mean = 6
Now, lets calculate the deviation of each value
For xi = 2, $|xi - \bar{x}| = |2-6| = 4$

For xi = 4, $|xi - \bar{x}| = |4-6| = 2$
For xi = 6, $|xi - \bar{x}| = |6-6| = 0$
For xi = 8, $|xi - \bar{x}| = |8-6| = 2$
For xi = 10, $|xi - \bar{x}| = |10-6| = 4$

Average deviation = $\sum(x - \bar{x})/n$

= 12/5

= 2.4

Average deviation in case of discrete and continuous series
Average Deviation about Mean or Median or Mode =
where N = No. of items
|d| = deviations from Mean or Median or Mode after ignoring signs.
Coefficient of A.D. about Mean or Median or Mode =
Example: Suppose we want to calculate coefficient of Average Deviation about Mean from the following discrete series:

X	**Frequency**
10	5
15	10
20	15
25	10
30	5

Solution: First of all, we shall calculate the value of arithmetic Mean,
Calculation of Arithmetic Mean

In case we want to calculate coefficient of Average Deviation about Median from the following data:

Class Interval	**Frequency**
10 – 14	5
15 – 19	10
20 – 24	15
25 – 29	10
30 – 34	5

First of all we shall calculate the value of Median but it is necessary to find the 'real limits' of the given class-intervals. This is possible by subtracting 0.5 from all the lower-limits and adds 0.5 to all the upper limits of the given classes. Hence, the real limits shall be: 9.5 – 14.5, 14.5 – 19.5, 19.5 – 24.5, 24.5 – 29.5 and 29.5 – 34.5.

Advantages of Average Deviations:
1. Average deviation takes into account all the items of a series and hence, it provides sufficiently representative results.
2. It simplifies calculations since all signs of the deviations are taken as positive.
3. Average Deviation may be calculated either by taking deviations from Mean or Median or Mode.

4. Average Deviation is not affected by extreme items.
5. It is easy to calculate and understand.
6. Average deviation is used to make healthy comparisons.

Disadvantages of Average Deviations

1. It is illogical and mathematically unsound to assume all negative signs as positive signs.
2. Because the method is not mathematically sound, the results obtained by this method are not reliable.
3. This method is unsuitable for making comparisons either of the series or structure of the series.

This method is more effective during the reports presented to the general public or to groups who are not familiar with statistical methods.

(d) Standard Deviation:

- The standard deviation, which is shown by Greek letter s (read as sigma) is extremely useful in judging the representativeness of the mean.
- The concept of standard deviation, which was introduced by Karl Pearson, has a practical significance because it is free from all defects, which exists in a range, quartile deviation or average deviation.
- Standard deviation is calculated as the square root of average of squared deviations taken from actual mean.
- It is also called root mean square deviation. The square of standard deviation i.e., s2 is called 'variance'.

Calculation of standard deviation in case of raw data

There are four ways of calculating standard deviation for raw data:
(i) When actual values are considered;
(ii) When deviations are taken from actual mean;
(iii) When deviations are taken from assumed mean; and
(iv) When 'step deviations' are taken from assumed mean.

(i) When the actual values are considered:

σ = where, N = Number of the items,
or σ^2 = X = Given values of the series,
= Arithmetic mean of the series

Steps to calculate σ

(i) Compute simple mean of the given values,
(ii) Square the given values and aggregate them
(iii) Apply the formula to find the value of standard deviation

Example: Suppose the values are given 2, 4, 6, 8, 10. We want to apply the formula
σ =

Solution: We are required to calculate the values of N, SX2. They are calculated as follows:

X	**X^2**
2	4
4	16
6	36
8	64

10 100
N = 5 $\sum X^2$ = 220
σ =
Variance $(\sigma)^2$ =

(ii) When the deviations are taken from actual mean
σ = where, N = no. of items and x = (mean)
Steps to Calculate σ
(i) Compute the deviations of given values from actual mean i.e., (mean) and represent them by x.
(ii) Square these deviations and aggregate them
(iii) Use the formula, **σ** =
Example : We are given values as 2, 4, 6, 8, 10. We want to find out standard deviation.

X	**(X –) = x**	**x^2**
2	2 – 6 = – 4	$(– 4)^2 = 16$
4	4 – 6 = – 2	$(– 2)^2 = 4$
6	6 –6 = 0	= 0
8	8 – 6 = + 2	$(2)^2 = 4$
10	10 – 6 = + 4	$(4)^2 = 16$
N = 5		**$\sum x^2$ = 40**

(iii) When the deviations are taken from assumed mean
σ =
where, N = no. of items,
dx = deviations from assumed mean i.e., (X – A).
A = assumed mean
Steps to Calculate :
(i) We consider any value as assumed mean. The value may be given in the series or may not be given in the series.
(ii) We take deviations from the assumed value i.e., (X – A), to obtain dx for the series and aggregate them to find ∑dx.
(iii) We square these deviations to obtain dx2 and aggregate them to find $\sum dx^2$.
(iv) Apply the formula given above to find standard deviation.
Example : Suppose the values are given as 2, 4, 6, 8 and 10. We can obtain the standard deviation as:

X	**dx = (X – A)**	**dx^2**
2	– 2 = (2 – 4)	4
assumed mean (A) 4	0 = (4 – 4)	0
6	+ 2 = (6 – 4)	4
8	+ 4 = (8 – 4)	16
10	+ 6 = (10 – 4)	36
N = 5	**∑dx = 10**	**$\sum dx^2$ = 60**

Coefficient of Variation or C. V:

Generally, coefficient of variation is used to compare two or more series. If coefficient of variation (C.V.) is more for one series as compared to the other, there will be more variations in that series, lesser stability or consistency in its composition. If coefficient of variation is lesser as compared to other series, it will be more stable or consistent. Moreover that series is always better where coefficient of variation or coefficient of standard deviation is lesser.

Example : Suppose we want to compare two firms where the salaries of the employees are given as follows:

	Firm A	**FirmB**
No. of workers	100	100
Mean salary (Rs.)	100	80
Standard-deviation (Rs.)	40	45

Solution: We can compare these firms either with the help of coefficient of standard deviation or coefficient of variation. If we use coefficient of variation, then we shall apply the formula:

Firm A **Firm B**

C.V. = C.V. =

= 100, σ = 40. = 80, σ = 45

Because the coefficient of variation is lesser for firm A than firm B, therefore, firm A is less variable and more stable.

Calculation of standard deviation in discrete and continuous series

we use the same formula for calculating standard deviation for a discrete series and a continuous series. The only difference is that in a discrete series, values and frequencies are given whereas in a continuous series, class-intervals and frequencies are given. When the mid-points of these class-intervals are obtained, a continuous series takes shape of a discrete series. X denotes values in a discrete series and mid points in a continuous series.

When the deviations are taken from actual mean

We use the same formula for calculating standard deviation for a continuous series

σ =

where N = Number of items

f = Frequencies corresponding to different values or class-intervals.

x = Deviations from actual mean.

X = Values in a discrete series and mid-points in a continuous series.

Step to calculate σ

(i) Compute the arithmetic mean by applying the required formula.

(ii) Take deviations from the arithmetic mean and represent these deviations by x.

(iii) Square the deviations to obtain values of x .

(iv) Multiply the frequencies of different class-intervals with x^2 to find fx^2. Aggregate fx^2 column to obtain $\sum fx^2$.

(v) Apply the formula to obtain the value of standard deviation.

If we want to calculate variance then we can compute σ^2 =

Example : We can understand the procedure by taking an example :

σ = where, N = 45, $\sum fx^2$ = 1500

σ =

When the deviations are taken from assumed mean

In some cases, the value of simple mean may be in fractions, then it becomes time consuming to

take deviations and square them. Alternatively, we can take deviations from the assumed mean.
σ =
where N = Number of the items,
dx = deviations from assumed mean (X – A),
f = frequencies of the different groups,
A = assumed mean and
X = Values or mid points.

Step to calculate σ

(i) Take the assumed mean from the given values or mid points.
(ii) Take deviations from the assumed mean and represent them by dx.
(iii) Square the deviations to get dx^2 .
(iv) Multiply f with dx of different groups to abtain fdx and add them up to get ∑fdx.
(v) Multiply f with dx^2 of different groups to abtain fdx^2 and add them up to get $\sum fdx^2$.
(vi) Apply the formula to get the value of standard deviation.

Steps to calculate σ

(i) Take deviations from the assumed mean of the calculated mid-points and divide all deviations by a common factor (i) and represent these values by dx.
(ii) Square these step deviations dx to obtain dx^2 for different groups.
(iii) Multiply f with dx of different groups to find fdx and add them to obtain fdx .
(iv) Multiply f with dx^2 of different groups to find fdx2 for different groups and add them to obtain ∑fdx2.
(v) Apply the formula to find standard deviation.

R - Linear Regression

- Regression analysis is a very widely used statistical tool to establish a relationship model between two variables. One of these variable is called **predictor variable** whose value is gathered through experiments. The other variable is called **response variable** whose value is derived from the predictor variable.
- In Linear Regression, these two variables are related through an equation, where exponent (power) of both these variables is 1. Mathematically a linear relationship represents a straight line when plotted as a graph. A non-linear relationship where the exponent of any variable is not equal to 1 creates a curve.
- The general mathematical equation for a linear regression is −

```
y = ax + b
```

Following is the description of the parameters used −

- **y** is the response variable.
- **x** is the predictor variable.
- **a** and **b** are constants which are called the coefficients.

Steps to Establish a Regression

A simple example of regression is predicting weight of a person when his height is known. To do this we need to have the relationship between height and weight of a person.

The steps to create the relationship is −

- Carry out the experiment of gathering a sample of observed values of height and corresponding weight.
- Create a relationship model using the **lm()** functions in R.
- Find the coefficients from the model created and create the mathematical equation using these
- Get a summary of the relationship model to know the average error in prediction. Also called **residuals**.
- To predict the weight of new persons, use the **predict()** function in R.

Input Data

Below is the sample data representing the observations −

```
# Values of height
151, 174, 138, 186, 128, 136, 179, 163, 152, 131

# Values of weight.
63, 81, 56, 91, 47, 57, 76, 72, 62, 48
```

lm() Function

This function creates the relationship model between the predictor and the response variable.

Syntax

The basic syntax for **lm()** function in linear regression is −

```
lm(formula,data)
```

Following is the description of the parameters used −

- **formula** is a symbol presenting the relation between x and y.
- **data** is the vector on which the formula will be applied.

Create Relationship Model & get the Coefficients

```
x <- c(151,174,138,186,128,136,179,163,152,131)
y <- c(63,81,56,91,47,57,76,72,62,48)

# Apply the lm() function.
relation <- lm(y~x)

print(relation)
```

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

```
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)          x
   -38.4551       0.6746
```

Get the Summary of the Relationship

```
x <- c(151,174,138,186,128,136,179,163,152,131)
y <- c(63,81,56,91,47,57,76,72,62,48)

# Apply the lm() function.
```

```
relation <- lm(y~x)

print(summary(relation))
```

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

```
Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-6.3002  -1.6629  0.0412  1.8944  3.9775

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -38.45509    8.04901  -4.778  0.00139 **
x             0.67461    0.05191  12.997 1.16e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.253 on 8 degrees of freedom
Multiple R-squared:  0.9548,    Adjusted R-squared:  0.9491
F-statistic: 168.9 on 1 and 8 DF,  p-value: 1.164e-06
```

predict() Function

Syntax

The basic syntax for predict() in linear regression is −

```
predict(object, newdata)
```

Following is the description of the parameters used −

- **Object** is the formula which is already created using the lm() function.
- **newdata** is the vector containing the new value for predictor variable.

Predict the weight of new persons

```
# The predictor vector.
x <- c(151,174,138,186,128,136,179,163,152,131)

# The resposne vector.
y <- c(63,81,56,91,47,57,76,72,62,48)

# Apply the lm() function.
relation <- lm(y~x)

# Find weight of a person with height 170.
a <- data.frame(x =170)
result <-  predict(relation,a)
print(result)
```

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

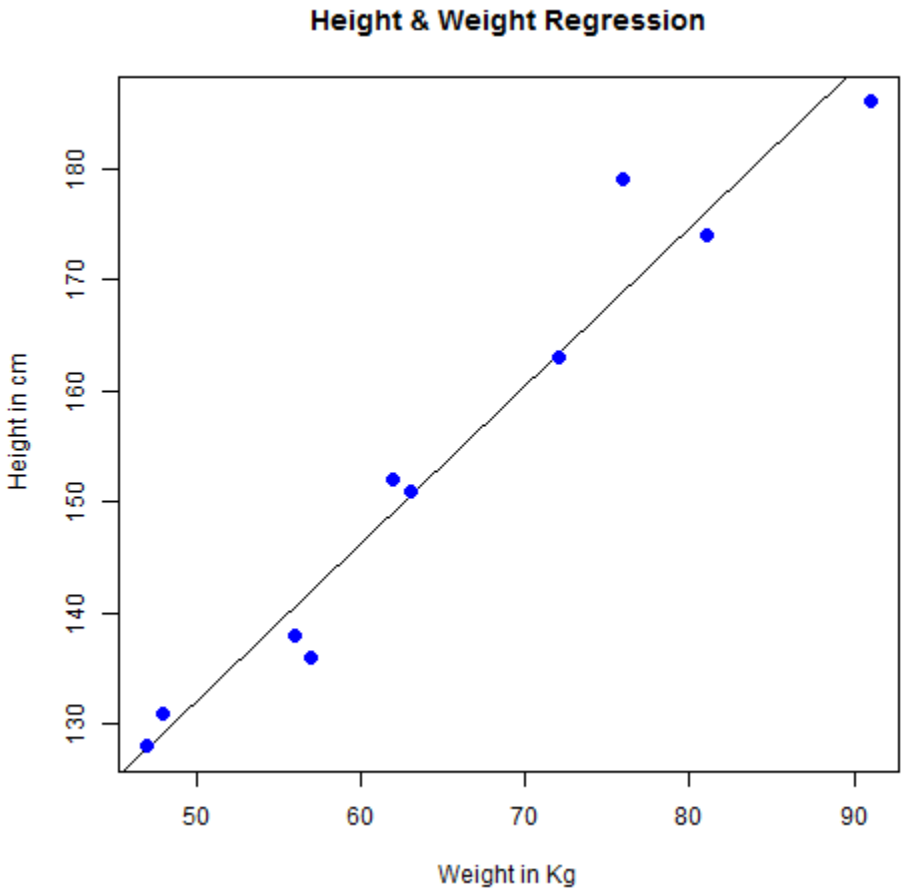
```
       1
```

```
76.22869
```

Visualize the Regression Graphically

```
# Create the predictor and response variable.
x <- c(151,174,138,186,128,136,179,163,152,131)
y <- c(63,81,56,91,47,57,76,72,62,48)
relation <- lm(y~x)

# Give the chart file a name.
png(file ="linearregression.png")

# Plot the chart.
plot(y,x,col ="blue",main ="Height & Weight Regression",
abline(lm(x~y)),cex =1.3,pch =16,xlab ="Weight in Kg",ylab ="Height in cm")

# Save the file.
dev.off()
```

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



R - Nonlinear Least Square

- When modelling real world data for regression analysis, we observe that it is rarely the case that the equation of the model is a linear equation giving a linear graph.
- Most of the time, the equation of the model of real world data involves mathematical functions of higher degree like an exponent of 3 or a sin function. In such a scenario, the plot of the model gives a curve rather than a line.
- The goal of both linear and non-linear regression is to adjust the values of the model's parameters to find the line or curve that comes closest to your data. On finding these values we will be able to estimate the response variable with good accuracy.
- In Least Square regression, we establish a regression model in which the sum of the squares of the vertical distances of different points from the regression curve is minimized.
- We generally start with a defined model and assume some values for the coefficients. We then apply the **nls()** function of R to get the more accurate values along with the confidence intervals.

Syntax

The basic syntax for creating a nonlinear least square test in R is −

```
nls(formula, data, start)
```

Following is the description of the parameters used −

- **formula** is a nonlinear model formula including variables and parameters.
- **data** is a data frame used to evaluate the variables in the formula.
- **start** is a named list or named numeric vector of starting estimates.

Example

We will consider a nonlinear model with assumption of initial values of its coefficients. Next we will see what is the confidence intervals of these assumed values so that we can judge how well these values fir into the model.

So let's consider the below equation for this purpose −

```
a = b1*x^2+b2
```

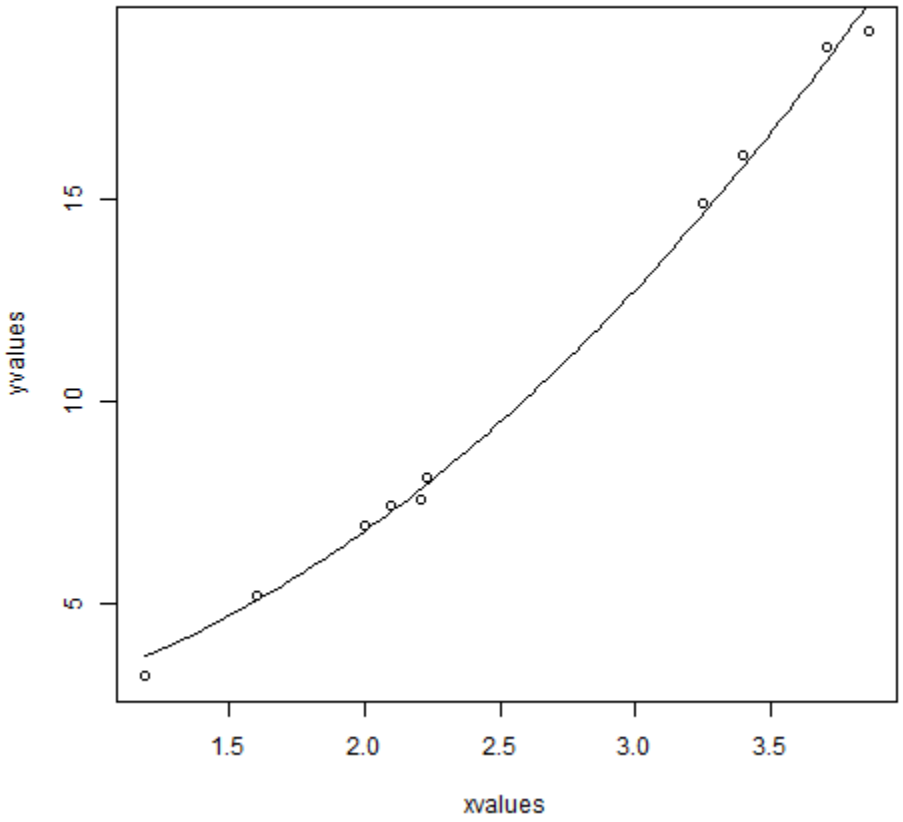
Let's assume the initial coefficients to be 1 and 3 and fit these values into nls() function.

```
xvalues <- c(1.6,2.1,2,2.23,3.71,3.25,3.4,3.86,1.19,2.21)
yvalues <- c(5.19,7.43,6.94,8.11,18.75,14.88,16.06,19.12,3.21,7.58)

# Give the chart file a name.
png(file ="nls.png")

# Plot these values.
plot(xvalues,yvalues)


# Take the assumed values and fit into the model.
model <- nls(yvalues ~ b1*xvalues^2+b2,start = list(b1 =1,b2 =3))

# Plot the chart with new data by fitting it to a prediction from 100 data points.
new.data <- data.frame(xvalues = seq(min(xvalues),max(xvalues),len =100))
lines(new.data$xvalues,predict(model,newdata =new.data))
```

```
# Save the file.
dev.off()

# Get the sum of the squared residuals.
print(sum(resid(model)^2))

# Get the confidence intervals on the chosen values of the coefficients.
print(confint(model))
```

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

```
[1] 1.081935
Waiting for profiling to be done...
      2.5%    97.5%
b1 1.137708 1.253135
b2 1.497364 2.496484
```



We can conclude that the value of b1 is more close to 1 while the value of b2 is more close to 2 and not 3.

Time Series

R has extensive facilities for analysing time series data. This section describes the creation of a time series, seasonal decomposition, modelling with exponential and ARIMA models, and forecasting with the forecast package.

Creating a time series

The ts() function will convert a numeric vector into an R time series object. The format is ts(vector, start=, end=, frequency=) where start and end are the times of the first and last observation and frequency is the number of observations per unit time (1=annual, 4=quartile, 12=monthly, etc.).

```
# save a numeric vector containing 72 monthly observations
# from Jan 2009 to Dec 2014 as a time series object
myts <- ts(myvector, start=c(2009, 1), end=c(2014, 12), frequency=12)

# subset the time series (June 2014 to December 2014)
myts2 <- window(myts, start=c(2014, 6), end=c(2014, 12))

# plot series
plot(myts)
```

Correlations

You can use the cor( ) function to produce correlations and the cov( ) function to produces covariance.
A simplified format is cor(x, use=, method=)
Where

Option	Description
x	Matrix or data frame
use	specifies the handling of missing data. Options are all.obs (assumes no missing data - missing data will produce an error), complete.obs (list wise deletion), and pair-wise. complete.obs (pair-wise deletion).
method	Specifies the type of correlation. Options are Pearson, spearman or Kendall.

```
# Correlations/covariance among numeric variables in
# data frame mtcars. Use list-wise deletion of missing data.
cor(mtcars, use="complete.obs", method="kendall")
cov(mtcars, use="complete.obs")
```

Unfortunately, neither cor( ) or cov( ) produce tests of significance, although you can use the cor.test( ) function to test a single correlation coefficient.

The rcorr( ) function in the Hmisc package produces correlations/covariance and significance levels for Pearson and spearman correlations. However, input must be a matrix and pair-wise deletion is used.

```
# Correlations with significance levels
```

```
library(Hmisc)
rcorr(x, type="pearson") # type can be Pearson or spearman

#mtcars is a data frame
rcorr(as.matrix(mtcars))
```

You can use the format cor(X, Y) or rcorr(X, Y) to generate correlations between the columns of X and the columns of Y. This similar to the VAR and WITH commands in SAS PROC CORR.

```
# Correlation matrix from mtcars
# with mpg, cyl, and disp as rows
# and hp, drat, and wt as columns
x <- mtcars[1:3]
y <- mtcars[4:6]
cor(x, y)
```