UNIT-I **(18hrs)**

Introduction - How to run R - R Sessions and Functions - Basic Math – Variables - Data Types – Vectors – Conclusion - Advanced Data Structures - Data Frames – Lists – Matrices – Arrays - Classes.

UNITII **(18hrs)**

R Programming Structures - Control Statements – Loops – Looping Over Non-vector Sets – IfElse - Arithmetic and Boolean Operators and values - Default Values for Argument - Return Values - Deciding Whether to explicitly call return Returning Complex Objects - Functions are Objective - No Pointers in R – Recursion - A Quicksort Implementation Extended - Example: A Binary Search Tree.

UNITIII **(18hrs)**

Doing Math and Simulation in R - Math Function - Extended Example Calculating Probability Cumulative Sums and Products Minima and Maxima Calculus - Functions Fir Statistical Distribution – Sorting - Linear Algebra Operation on Vectors and Matrices - Extended Example: Vector cross Product Extended Example: Finding Stationary Distribution of Markov Chains - Set Operation - Input /Output - Accessing the Keyboard and Monitor - Reading and writer Files.

UNITIV **(18hrs)**

Graphics - Creating Graphs - The Workhorse of R Base Graphics - the plot() Function – Customizing Graphs - Saving Graphs to Files.

UNITV **(18hrs)**

Probability Distributions - Normal Distribution Binomial Distribution Poisson Distributions other Distribution - Basic Statistics - Correlation and Covariance – Ttests – ANOVA - Linear Models - Simple Linear Regression - Multiple Regression Generalized Linear Models - Logistic Regression – Poisson Regression other Generalized Linear Models Survival Analysis, Nonlinear Models, Splines Decision Random Forests,

1. PRESCRIBED BOOKS

i. The Art of R Programming, Norman Matloff, Cengage Learning
ii. R for Everyone, Lander, Pearson
iii. Siegel, S. (1956), Nonparametric Statistics for the Behavioral Sciences, McGrawHill International, Auckland.
iv. R Cookbook, PaulTeetor, Oreilly.

Introduction: How to run R, R Sessions and Functions, Basic Math, Variables, Data Types, Vectors, Conclusion, Advanced Data Structures:Data Frames, Lists, Matrices, Arrays, Classes.

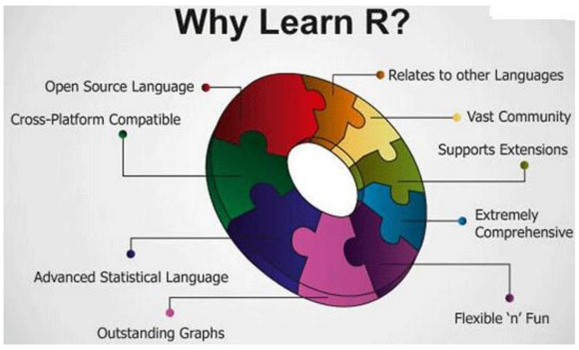
Introduction: R is a scripting language (are often interpreted rather than compiled) for statistical data manipulation and analysis. It was inspired by, and is mostly compatible with, the statistical language S developed by AT&T.R is designed by Ross ihanka and Robert Gentleman, developed by R core team.

Five reasons to learn and use R:

- R is open source and completely free. R community members regularly contribute packages to increase R's functionality.
- R is as good as commercially available statistical packages like SPSS, SAS, and Minitab.
- R has extensive statistical and graphing capabilities. R provides hundreds of built-in statistical functions as well as its own built-in programming language.
- R is used in teaching and performing computational statistics. It is the language of choice for many academics who teach computational statistics.
- Getting help from the R user community is easy. There are readily available online tutorials, data sets, and discussion forums about R.

R uses:

- R combines aspects of functional and object-oriented programming.
- R can use in interactive mode
- It is an interpreted language rather than a compiled one.
- Finding and fixing mistakes is typically much easier in R than in many other languages.



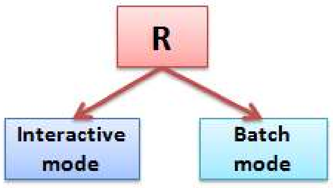
R Features

- Programming language for graphics and statistical computations
- Available freely under the GNU public license
- Used in data mining and statistical analysis
- Included time series analysis, linear and nonlinear modeling among others
- Very active community and package contributions
- Very little programming language knowledge necessary
- Can be downloaded from http://www.r-project.org/ opensource

Free tools of R:-

- ✓ RStudio
- ✓ StatET
- ✓ ESS (Emacs Speaks Statistics)
- ✓ R Commander
- ✓ JGR (Java GUI for R)

How to Run R:- R operates in two modes: interactive and batch mode.



1.Interactive Mode:- Interactive sessions prompt the user for input as data or commands. Typically, in an interactive session there is a software running on a computer environment and accepts input from human. This is the simplest way to work on any system – you simply log on and run whatever commands you need to, whether on the command line or in a graphical environment and you log out when you've finished.

Example:

1) *source("sample.R")* - source causes R to accept its input from the named file or URL or connection or expressions directly.

2) *mean(abs(rnorm(100)))-*

[1] 0.7194236 - Code generates the 100 random variates, finds their absolute values, and then finds the mean of the absolute values.

2.Batch mode:- Batch processing is the execution of a series of programs or only one task on a computer environment without manual intervention. All data and commands are preselected through scripts or command-line parameters and therefore run to completion without human contact. This is termed as "batch processing" or "batch mode" because the input data are collected into batches of files and are processed in batches by the program. In many cases batch jobs are submitted to a job scheduler and run on the first available compute node(s).

Example:- As an example, consider graph-making code into a file named z.R with the following contents:

```
pdf("xh.pdf")          # set graphical output file
hist(rnorm(100))       # generate 100 N(0,1) variates and plot their histogram
dev.off()              # close the graphical output file
```

Here's a step-by-step breakdown of the preceding code:

- ✓ # indicates comments in R scripting language.
- ✓ pdf() function to inform R that we want the graph we create to be saved in the PDF file *xh.pdf*.
- ✓ rnorm() (for *random normal*) to generate 100 *N*(0,1) random variates.
- ✓ hist() on those variates to draw a histogram of these values.
- ✓ dev.off() to close the graphical "device" which is the file *xh.pdf* . This is the mechanism that actually causes the file to be written to disk.

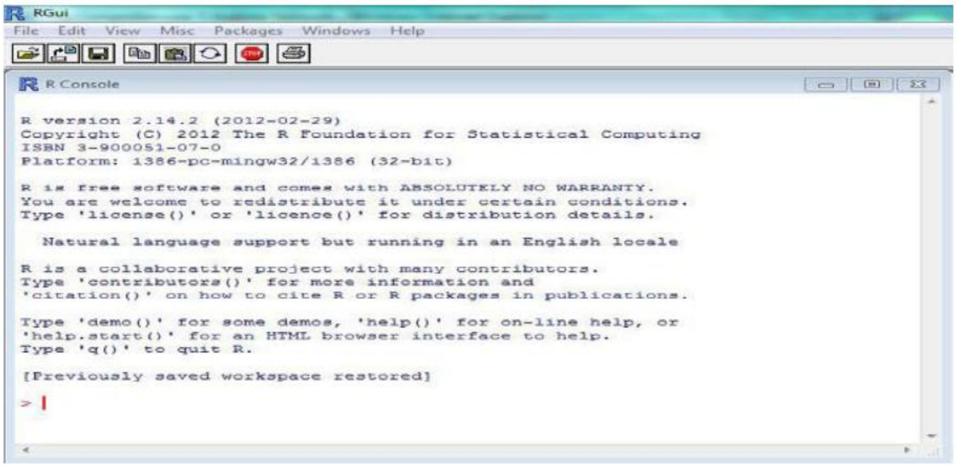
We could run this code automatically, without entering R's interactive mode, by invoking R with an operating system shell command (such as at the $ prompt commonly used in Linux systems):

```
$ R CMD BATCH z.R
```

1.2 How to Run R

What is CRAN?

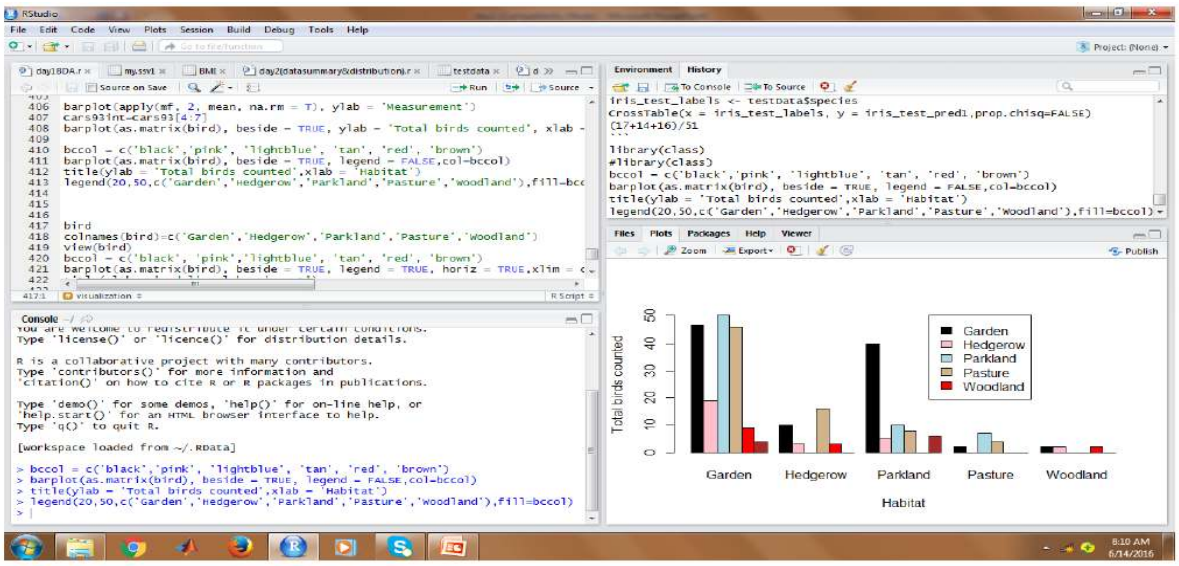
CRAN abbreviates Comprehensive R Archive Network will provide binary files and follow the installation instructions and accepting all defaults. Download from http://cran.r-project.org/ we can see the R Console window will be in the RGui (graphical user interface). Fig 1 is the sample R GUI.



```
R version 2.14.2 (2012-02-29)
Copyright (C) 2012 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: i386-pc-mingw32/i386 (32-bit)

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

  Natural language support but running in an English locale

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

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

[Previously saved workspace restored]

> |
```

Figure 1. R console

R Studio: R Studio is an Integrated Development Environment (IDE) for R Language with advanced and more user-friendly GUI. R Studio allows the user to run R in a more user-friendly environment. It is open-source (i.e.free) and available at http://www.rstudio.com/.



```
barplot(apply(mf, 2, mean, na.rm = T), ylab = 'Measurement')
cars93int-cars93[4:7]
barplot(as.matrix(bird), beside - TRUE, ylab - 'Total birds counted', xlab -

bccol - c('black','pink', 'lightblue', 'tan', 'red', 'brown')
barplot(as.matrix(bird), beside - TRUE, legend - FALSE,col-bccol)
title(ylab = 'Total birds counted',xlab = 'Habitat')
legend(20,50,c('Garden','Hedgerow','Parkland','Pasture','Woodland'),fill-bcc



bird
colnames(bird)=c('Garden','Hedgerow','Parkland','Pasture','Woodland')
view(bird)
bccol - c('black', 'pink','lightblue', 'tan', 'red', 'brown')
barplot(as.matrix(bird), beside = TRUE, legend = TRUE, horiz = TRUE,xlim = c
```

```
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

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

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

[workspace loaded from ~/.RData]

> bccol = c('black','pink', 'lightblue', 'tan', 'red', 'brown')
> barplot(as.matrix(bird), beside - TRUE, legend - FALSE,col-bccol)
> title(ylab - 'Total birds counted',xlab - 'Habitat')
> legend(20,50,c('Garden','Hedgerow','Parkland','Pasture','Woodland'),fill=bccol)
> |
```

```
iris_test_labels <- testData$species
CrossTable(x = iris_test_labels, y = iris_test_pred1,prop.chisq=FALSE)
(17+14+16)/51

library(class)
#library(class)
bccol - c('black','pink', 'lightblue', 'tan', 'red', 'brown')
barplot(as.matrix(bird), beside - TRUE, legend - FALSE,col-bccol)
title(ylab = 'Total birds counted',xlab = 'Habitat')
legend(20,50,c('Garden','Hedgerow','Parkland','Pasture','Woodland'),fill=bccol)
```

■	Garden
▢	Hedgerow
▢	Parkland
▨	Pasture
🟥	Woodland

The fig shows the GUI of R Studio.The R Studio screen has four windows:

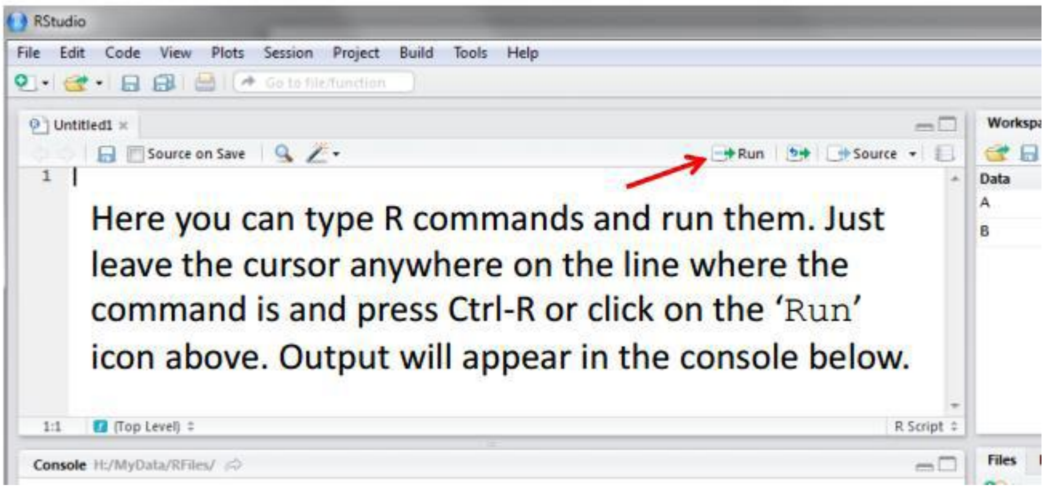
1. Console.
2. Workspace and history.
3. Files, plots, packages and help.
4. The R script(s) and data view.

The R script is where you keep a record of your work.

Create a new R script file:

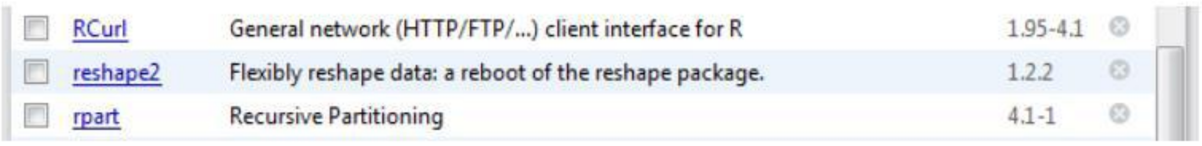
1) File -> New -> R Script,
2) Click on the icon with the "+" sign and select "R Script"
3) Use shortcut as: Ctrl+Shift+N.

Running the R commands on R Script file:



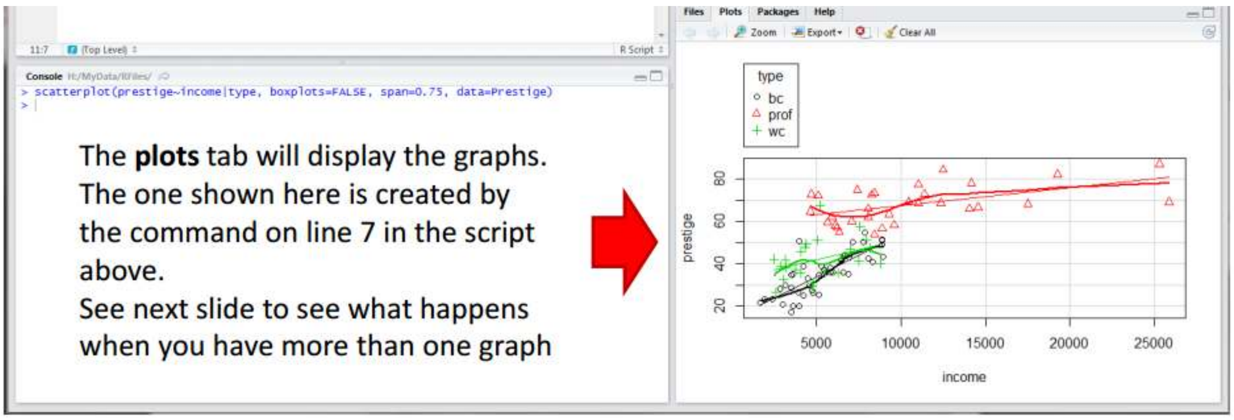
Here you can type R commands and run them. Just leave the cursor anywhere on the line where the command is and press Ctrl-R or click on the 'Run' icon above. Output will appear in the console below.

Installing Packages:



☐ RCurl	General network (HTTP/FTP/...) client interface for R	1.95-4.1	⊗
☐ reshape2	Flexibly reshape data: a reboot of the reshape package.	1.2.2	⊗
☐ rpart	Recursive Partitioning	4.1-1	⊗

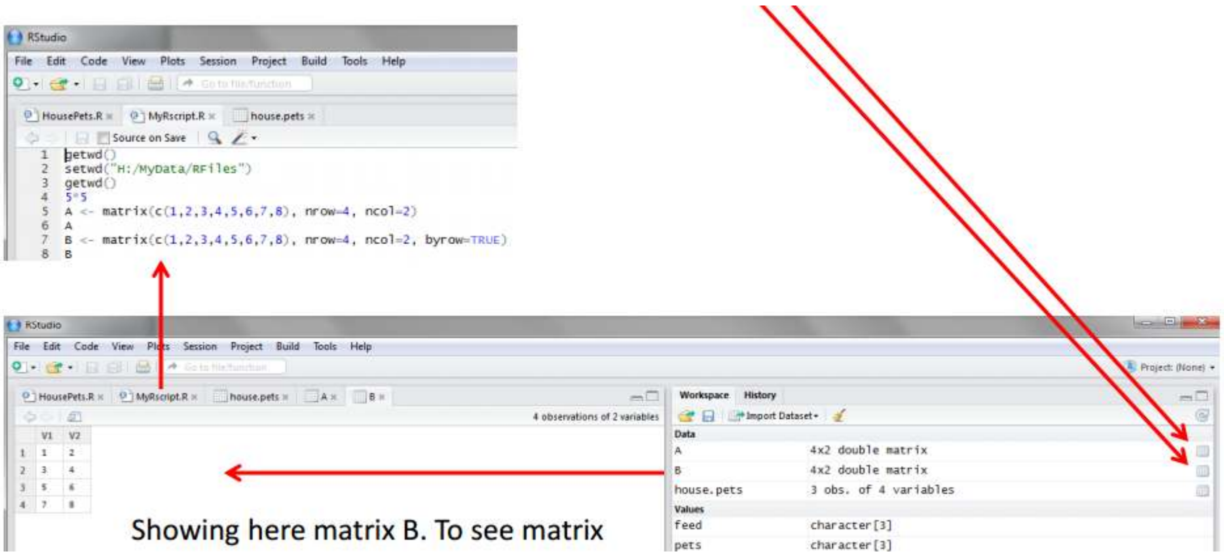
Plots to display:



```
> scatterplot(prestige~income|type, boxplots=FALSE, span=0.75, data=Prestige)
>
```

The **plots** tab will display the graphs. The one shown here is created by the command on line 7 in the script above.
See next slide to see what happens when you have more than one graph

Console: The console is where you can type commands and see output.
Workspace tab: The workspace tab shows all the active objects (see next slide). The workspace tab stores any object, value, function or anything you create during your R session. In the example below, if you click on the dotted squares you can see the data on a screen to the left.



```
getwd()
setwd("H:/MyData/RFiles")
getwd()
5*5
A <- matrix(c(1,2,3,4,5,6,7,8), nrow=4, ncol=2)
A
B <- matrix(c(1,2,3,4,5,6,7,8), nrow=4, ncol=2, byrow=TRUE)
B
```

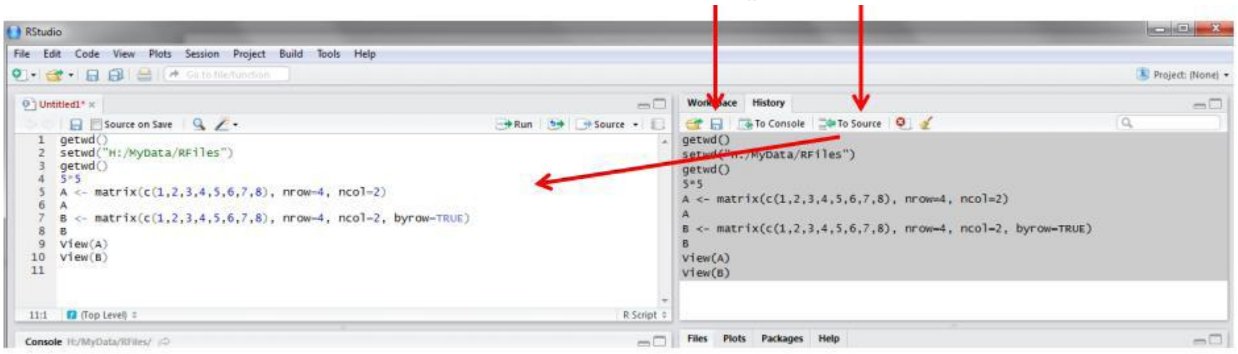
4 observations of 2 variables

	V1	V2
1	1	2
2	3	4
3	5	6
4	7	8

Showing here matrix B. To see matrix

Data	
A	4x2 double matrix
B	4x2 double matrix
house.pets	3 obs. of 4 variables
Values	
feed	character[3]
pets	character[3]

History tab:
The history tab shows a list of commands used so far.The history tab keeps a record of all previous commands. It helps when testing and running processes. Here you can either save the whole list or you can select the commands you want and send them to an R script to keep track of your work. In this example, we select all and click on the "To Source" icon, a window on the left will open with the list of commands. Make sure to save the 'untitled1' file as an *.R script.



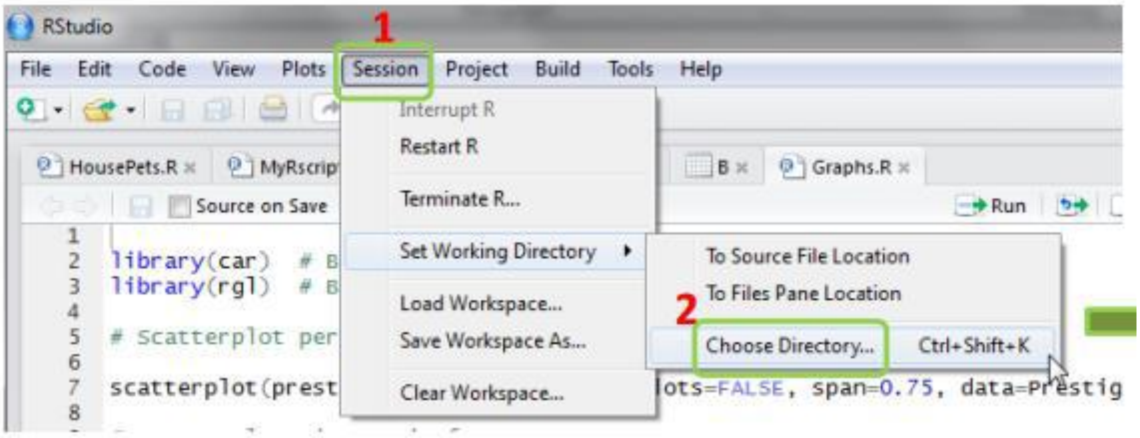
```
getwd()
setwd("H:/MyData/RFiles")
getwd()
5*5
A <- matrix(c(1,2,3,4,5,6,7,8), nrow=4, ncol=2)
A
B <- matrix(c(1,2,3,4,5,6,7,8), nrow=4, ncol=2, byrow=TRUE)
B
View(A)
View(B)
```

Files Tab: The files tab shows all the files and folders in your default workspace as if you were on a PC/Mac window. The plots tab will show all your graphs. The packages tab will list a series of packages or add-ons needed to run certain processes.
Changing the working directory:
To Show the present working directory (wd)

```
>getwd()
C:/mydocuments      #The default working directory is mydocuments
```

To change the working directory

```
>setwd("C:/myfolder/data")
```



```
library(car)  # B
library(rgl)  # B

# Scatterplot per

scatterplot(prest
lots=FALSE, span=0.75, data=Prestig
```

First R program: Using R as calculator:
R commands can run in two ways:
1) Type at console and press enter to see the output. Output will get at console only in R studio.
2) Open new R Script file and write the command, keep the curser on the same line and press Ctrl+enter or click on Run. Then see the output at console along with command.

At console:
R as a calculator, typing commands directly into the R Console. Launch R and type the following code, pressing
< Enter > after each command.
Type an expression on console.

R Sessions:-

- R is a case-sensitive, interpreted language. You can enter commands one at a time at the command prompt (>) or run a set of commands from a source file.
- There are a wide variety of data types, including vectors, matrices, data frames (similar to datasets), and lists (collections of objects).
- The standard assignment operator in R is <-. = can also used, but this is discouraged, as it does not work in some special situations.
- There are no fixed types associated with variables.
- The variables can be printed without any print statement by giving name of the variable.

```
> y <- 5
> y        # print out y
  [1] 5

>print y   # print out y
  [1] 5
```

- Comments (#) are especially valuable for documenting program code, but they are useful in interactive sessions

Note:-

- Prompt for new input is '>'
- '+' is a line continuation character in R.

Functions:- A function is a group of instructions that takes inputs, uses them to compute other values, and returns a result.

Function	Purpose	Example
getwd()	List the current working directory.	> getwd() [1] "C:/Users/SAIRAM/Documents"
setwd("mydirectory")	Change the current working directory to mydirectory.	> setwd("mydirectory")
help(options)	List the objects in the current workspace.	> help(sum)
q( )	Quit R. You'll be prompted to save the workspace.	> q()
length (object)	Number of elements/components.	> x <- c(2, 5, 6, 9) > length(x) [1] 4
dim (object)	Dimensions of an object. Object can be matrix, array or dataframe.	> a <-matrix(1:6,nrow=2,ncol=3) > a [,1] [,2] [,3] [1,] 1 3 5 [2,] 2 4 6 > dim(a) [1] 2 3
str (object)	Structure of an object.	> a <-matrix(1:6,nrow=2,ncol=3) > str(a) int [1:2, 1:3] 1 2 3 4 5 6
class(object)	Class or type of an object.	> y <- 2.4 > class(y) [1] "numeric"
mode(object)	How an object is stored.	> z<-6 > storage.mode(z) [1] "double" > storage.mode(z)<- "character" > storage.mode(z) [1] "character" > z [1] "6"
names(object)	Names of components in an object.	> names(z) NULL > names(z) <- "Rosen" > names(z) [1] "Rosen" > z Rosen "6"
c(object, object,...)	Combines objects into a vector.	> x <- c(1:4,8,13) > x [1] 1 2 3 4 8 13
cbind(object, object, ...)	Combines objects as columns.	> x<-matrix(1:6,nrow=2,ncol=3) > x [,1] [,2] [,3] [1,] 1 3 5

		[2,] 2 4 6 > cbind(x,7) [,1] [,2] [,3] [,4] [1,] 1 3 5 7 [2,] 2 4 6 7
rbind(object, object, ...)	Combines objects as rows.	> x<-matrix(1:4,nrow=2,ncol=2) > x [,1] [,2] [1,] 1 3 [2,] 2 4 > rbind(x,5) [,1] [,2] [1,] 1 3 [2,] 2 4 [3,] 5 5
object	Prints the object.	> y <- 12 > y [1] 12
head(object)	Lists the first part of the object i.e, the first six rows of the data	> x <- c(1:100) > head(x) [1] 1 2 3 4 5 6
tail(object)	Lists the last part of the object i.e, the last six rows of the data	> x <- c(1:100) > tail(x) [1] 95 96 97 98 99 100
ls()	Lists current objects.	> ls() [1] "a" "b" "cells" "cnames" "d"
rm(object, object, ...)	Deletes one or more objects. The statement rm(list = ls()) will remove most objects from the working environment.	> rm(a) > a Error: object 'a' not found > rm(list = ls()) > ls() character(0)
newobject <- edit(object)	Edits object and saves as newobject.	> y <- edit(x) > y [1] 5
fix(object)	Edits in place.	> fix(x) > x [1] 5

Mathematical functions:-

Function	Purpose	Example
abs(x)	Absolute value	> abs(-4) [1] 4
sqrt(x)	Square root sqrt(25) returns 5. This is the same as 25^(0.5).	> sqrt(16) [1] 4 > 16^0.5 [1] 4
ceiling(x)	Smallest integer not less than x	> ceiling(3.15452) [1] 4
floor(x)	Largest integer not greater than x	> trunc(5.99) [1] 5
trunc(x)	Integer formed by truncating values in x toward 0.	> floor(3.65452) [1] 3
round(x, digits=n)	Round x to the specified number of decimal places	> round(3.475, digits=2) [1] 3.48
signif(x, digits=n)	Round x to the specified number of significant digits	> signif(3.475, digits=3) [1] 3.48

cos(x) , sin(x) , tan(x)	Cosine, sine, and tangent	> cos(2) [1] -0.4161468

log(x,base=n)
log(x)
log10(x)
	Logarithm of x to the base n
	For convenience
	log(x) is the natural logarithm.
	log10(x) is the common logarithm.
		> log(100)
		[1] 4.60517
		> log(100,base=2)
		[1] 6.643856
		> log10(100)
		[1] 2

exp(x)	Exponential function	> exp(2.3026) [1] 10.00015

Statistical function:-

Function	Purpose	Example
range(x)	Range returns a vector containing the minimum and maximum of all the given arguments.	> x <- c(1,2,3,4) > range(x) [1] 1 4
sum(x)	Sum of all the values present in its arguments.	> x <- c(1,2,3,4) > sum(x) [1] 10
diff(x, lag=n)	Lagged differences, with lag indicating which lag to use. The default lag is 1.	> x<-c(1,6,3,4) > diff(x) [1] 5 -3 1
min(x)	Returns the minima of the input values.	> min(c(1,2,3,4)) [1] 1
max(x)	Returns the maxima of the input values.	> max(c(1,2,3,4)) [1] 4
mean(x)	Generic function for the (trimmed) arithmetic mean.	> x<-c(1,6,3,4) > mean(x) [1] 3.5
sd(x)	Computes the standard deviation of the values in x	> x<-c(1,6,3,4) > sd(x) [1] 2.081666
median(x)	Computes the median of the values in x	> x<-c(1,6,3,4) > median(x) [1] 3.5
sort (x)	Computes the elements in ascending order	> x <-c(1,56,3,25) > sort(x) [1] 1 3 25 56

Basic Math:- R is a powerful tool for all manner calculations, data manipulation and scientific computations. R can certainly be used to do basic math.

Examples:-

```
> 1+1
[1] 2

> 1+2+3
[1] 6

> 3*7*2
[1] 42

> 4/2
[1] 2

> 4/3
[1] 1.333333
```

R follows the basic order of operations: Parenthesis, Exponents, Multiplication, Division, Addition and Subtraction (PEMDAS). This means the operations inside parenthesis take priority over other operations. Next on the priority list is exponentiation.After that multiplication and division are performed, followed by addition and subtraction.

Example:-

```
> 4 * 6 + 5
[1] 29

> (4 * 6) + 5
[1] 29
```

```
> 4 * (6 + 5)
[1] 44
```

Variables:- Variables are integral part of any programming language. R does not require variable types to be declared. A variable can take on any available datatype.It can hold any R object such as a function, the result of an analysis or a plot. A single variable, at one point hold a number, then later hold a character and then later a number again.

Variable Assignment:- There a number of ways to assign a value to a variable, it does not depend on the type of value being assigned. There is no need to declare your variable first:

Example:-

```
> x <- 6      # assignment operator: a less-than character (<) and a hyphen (-) with no space
> x
[1] 6

> y = 3       # assignment operator = is used.
> y
[1] 3

> z <<- 9     # assignment to a global variable rather than a local variable.
> z
[1] 9

> 5 -> fun     #A rightward assignment operator (->) can be used anywhere
> fun
[1] 5

> a <- b <- 7   # Multiple values can be assigned simultaneously.
> a
[1] 7
> b
[1] 7

> assign("k",12)  # assign function can be used.
> k
[1] 12
```

Removing Variables:- rm() function is used to remove variables. This frees up memory so that R can store more objects,althought it does not necessarily free up memory for the operating system.

- There is no “undo”; once the variable is removed.
- Variable names are case sensitive.

```
> x <- 2*pi
> x
[1] 6.283185
> rm(x)                  # x variable is removed
> x
Error: object 'x' not found

> rm(x,a,y)              # removing multiple variables
> a
Error: object 'a' not found
> x
Error: object 'x' not found
> y
Error: object 'y' not found

> marks <-89             # Case sensitive
> mArks
Error: object 'mArks' not found
```

Modifying existing variable: Rename the existing variable by using rename() function.

For examples, *mydata<- rename(mydata, c(oldname="newname"))*

Variable (Object) Names: Certain variable names are reserved for particular purposes. Some reserved symbols are: c q t C D F I T

```
### meaning of c q t C D F I T
?     ## to see help document
?c    ## c means Combine Values into a Vector or List
?q    ## q means Terminate an R Session
?t    ## t means Matrix Transpose
?C    ## C means sets contrast for a factor
?D    ##  D means Symbolic and Algorithmic Derivatives of Simple Expressions
?F    ## F means logical vector Character strings
>F     ##[1] FALSE
?I    ##Inhibit Interpretation/Conversion of Objects
```

c("T", "TRUE", "True", "true") are true, c("F", "FALSE", "False", "false") as false, and all others as NA.

Data Types:- There are numerous data types in R that store various kinds of data. The four main types of data are numeric, character, Date/POSIXct (time-based) and logical (TRUE /FALSE).
The type of data contained in a variable is checked with the ***class*** function.

```
> x <- 8
> class(x)
[1] "numeric"
```

Numeric Data:- The most commonly used numeric data is ***numeric***. This is similar to float or double in other languages. It handles integers and decimals, both positive and negative, and also zero.

```
> i <- 5L          # To set an integer to a variable, append the value with an 'L'.
> i
[1] 5

> is.integer(i)     # Testing whether a variable is integer or not
[1] TRUE

> is.numeric(i)
[1] TRUE
```

R promotes integers to numeric when needed.

- Multiplying an integer to numeric results in decimal number.
- Dividing an integer with numeric results in decimal number.
- Dividing an integer with integer results in decimal number.

```
> class(4L)
[1] "integer"

> class(2.8)
[1] "numeric"

> x <- (4L*2.8)
> x
[1] 11.2
```

```
> class(x)
[1] "numeric"

> k <- (5L/2L)
> k
[1] 2.5
> class(k)
[1] "numeric"
```

Character Data:- The character datatype is used to store character and widely used in statistical analysis. x contains the word "data" encapsulated in quotes, while y has the word "data" without quotes and a second line has levels.

```
> x <- "data"
> x
[1] "data"

> y <- factor("data")
> y
[1] data
Levels: data
```

- Characters are case sensitive, “data” is different from “DaTa”.
- To find the length of the character nchar function can be used. nchar function cannot be used with factor data.

```
> x <- "Vishnu"
> nchar(x)
[1] 6
> nchar(3)
[1] 1
> nchar(567)
[1] 3
> nchar("hello")
[1] 5
```

Dates:- R has numerous different types of dates. The most useful are Date and POSIXct. Date stores just a date while POSIXct stores a date and time.

```
> date1 <- as.Date("2017-06-23")
> date1
[1] "2017-06-23"

> class(date1)
[1] "Date"
> as.numeric(date1)
[1] 17340
 > date2 <- as.POSIXct("2017-06-23 17:42")
> date2
[1] "2017-06-23 17:42:00 IST"

> class(date2)
[1] "POSIXct" "POSIXt"

> as.numeric(date2)
[1] 1498219920
```

Logical:- Logical are a way of representing data that can be either TRUE or FALSE. Numerically, TRUE is the same as 1 and FALSE is the same as 0. So TRUE*5 equals 5 while FALSE*5 equals 0.

```
> TRUE
[1] TRUE

> T
[1] TRUE

> TRUE*5
[1] 5

> FALSE*5
[1] 0

> k <- TRUE
> class(k)
[1] "logical"
```

Mode vs Class:

- 'mode' is a mutually exclusive classification of objects according to their basic structure. The 'atomic' modes are numeric, complex, character and logical. Recursive objects have modes such as 'list' or 'function' or a few others. An object has one and only one mode.

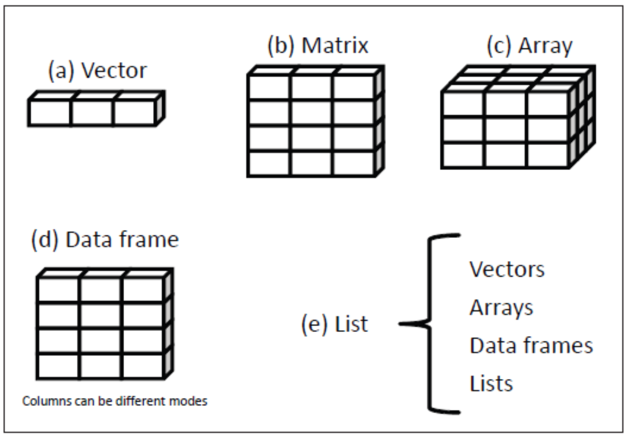
- 'class' is a property assigned to an object that determines how generic functions operate with it. It is not a mutually exclusive classification. If an object has no specific class assigned to it, such as a simple numeric vector, it's class is usually the same as its mode, by convention.Changing the mode of an object is often called 'coercion'. The mode of an object can change without necessarily changing the class.

e.g.

```
> x <- 1:16
> mode(x)
[1] "numeric"
> dim(x) <- c(4,4)
> mode(x)
[1] "numeric"
> class(x)
[1] "matrix"
```

```
> is.numeric(x)
[1] TRUE
> mode(x) <- "character"
> mode(x)
[1] "character"
> class(x)
[1] "matrix"
```

Advanced Data Structures:- R has a wide variety of objects for holding data, including scalars, vectors, matrices,arrays, data frames, and lists. They differ in terms of the type of data they can hold, how they're created, their structural complexity, and the notation used to identify and access individual elements.



Vector:- Vectors must be homogeneous i.e, the type of data in a given vector must all be the same. Vectors are one-dimensional arrays that can hold numeric data, character data, or logical data. The combine function c() is used to form the vector. Here are examples of each type of vector:

```
> a <- c(1, 2, 5, 3, 6, -2, 4)
> a
[1]  1  2  5  3  6 -2  4

> b <- c("one", "two", "three")
> b
[1] "one"   "two"   "three"

> c <- c(TRUE, TRUE, TRUE, FALSE, TRUE, FALSE)
> c
[1]  TRUE  TRUE  TRUE FALSE  TRUE FALSE
```

Here, a is numeric vector, b is a character vector, and c is a logical vector. Note that the data in a vector must only be one type or mode (numeric, character, or logical). You can't mix modes in the same vector.

Following are some other possibilities to create vectors

```
> x <- 1:10
> x
```

```
[1] 1 2 3 4 5 6 7 8 9 10
> y <- seq(10)      #Create a sequence
> y
[1] 1 2 3 4 5 6 7 8 9 10
> z <- rep(1,10)    #Create a repetitive pattern
> z
[1] 1 1 1 1 1 1 1 1 1 1
```

NOTE :- Scalars are one-element vectors. Examples include f <- 3, g <- "US" and h <- TRUE. They're used to hold constants.

- Elements of a vector are refered using a numeric vector of positions within brackets. For example, a[c(2, 4)] refers to the 2nd and 4th element of vector a. Here are additional examples:

```
> a <- c(1, 2, 5, 3, 6, -2, 4)
> a[3]
[1] 5
> a[c(1, 3, 5)]
[1] 1 5 6
> a[2:6]
[1] 2 5 3 6 -2
```

- The colon operator used in the last statement is used to generate a sequence of numbers. For example, *a <- c(2:6)* is equivalent to *a <- c(2, 3, 4, 5, 6).*
- If the arguments to c(...) are themselves vectors, it flattens them and combines them into one single vector:

```
> v1 <- c(1,2,3)
> v2 <- c(4,5,6)
> c(v1,v2)
[1] 1 2 3 4 5 6
```

- Vectors cannot contain a mix of data types, such as numbers and strings. If you create a vector from mixed elements, R will try to accommodate you by converting one of them:

```
> v1 <- c(1,2,3)
> v3 <- c("A","B","C")
> c(v1,v3)
[1] "1" "2" "3" "A" "B" "C"
```

  Here, the user tried to create a vector from both numbers and strings. R converted all the numbers to strings before creating the vector, thereby making the data elements compatible.
- Technically speaking, two data elements can coexist in a vector only if they have the same mode. The modes of 3.1415 and "foo" are numeric and character, respectively:

```
> mode(3.1415)
[1] "numeric"
> mode("foo")
[1] "character"
```

  Those modes are incompatible. To make a vector from them, R converts 3.1415 to character mode so it will be compatible with "foo":

```
> c(3.1415, "foo")
[1] "3.1415" "foo"
> mode(c(3.1415, "foo"))
[1] "character"
```

- Length of the vector can be obtained by length() function.

```
> x <- c(1,2,4)
> length(x)
[1] 3
```

Vector Arithmetic:

```
> x <- c(1:10)
> x
[1] 1 2 3 4 5 6 7 8 9 10
> y <- 10
> x + y
```

```
[1] 11 12 13 14 15 16 17 18 19 20
> 2 + 3 * x         #Note the order of operations
[1] 5 8 11 14 17 20 23 26 29 32
> (2 + 3) * x       #See the difference
[1] 5 10 15 20 25 30 35 40 45 50
> sqrt(x)           #Square roots
[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
[9] 3.000000 3.162278
> x %% 4            #This is the integer divide (modulo) operation
[1] 1 2 3 0 1 2 3 0 1 2
> y <- 3 + 2i       #R does complex numbers
> x * y
[1] 3+ 2i 6+ 4i 9+ 6i 12+ 8i 15 + 10i 18 + 12i 21 + 14i 24 + 16i 27 + 18i 30 + 20i
```

Matrices: A matrix is a two-dimensional array where each element has the same mode (numeric,character, or logical). Matrices are created with the matrix function . The general format is

```
myymatrix <- matrix(vector, nrow=number_of_rows, ncol=number_of_columns,
byrow=logical_value, dimnames=list(char_vector_rownames, char_vector_colnames))
```

where *vector* contains the elements for the matrix, nrow and ncol specify the row and column dimensions, and dimnames contains optional row and column labels stored in character vectors. The option byrow indicates whether the matrix should be filled in by row (byrow=TRUE) or by column (byrow=FALSE). The default is by column. The following listing demonstrates the matrix function.Matrix can be created in three ways

- matrix(): A vector input to the matrix function.
- Using rbind() and cbind() functions.
- Using dim() to the existing vector

Creating a matrix using matrix():

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

```
# Create a matrix.
   > y <- matrix(1:20, nrow=5, ncol=4)
   > y
      [,1] [,2] [,3] [,4]
   [1,]    1    6   11   16
   [2,]    2    7   12   17
   [3,]    3    8   13   18
   [4,]    4    9   14   19
   [5,]    5   10   15   20
```

```
# Create a matrix.
   > cells <- c(1,26,24,68)
   > rnames <- c("R1", "R2")
   > cnames <- c("C1", "C2")
   > mymatrix <- matrix(cells, nrow=2, ncol=2, byrow=TRUE,dimnames=list(rnames, cnames))
   > mymatrix
      C1 C2
   R1  1 26
   R2 24 68
   > mymatrix <- matrix(cells, nrow=2, ncol=2,dimnames=list(rnames, cnames))
   > mymatrix
      C1 C2
   R1  1  24
```

```
R2 26  68
```

Creating a matrix using rbind() or cbind(): First create two vectors and then create a matrix using rbind() .It binds the two vectors data into two rows of matrix.

Example:

```
create two vectors as xr1,xr2
> xr1 <- c( 6, 2, 10)
> xr2 <- c(1, 3, -2)
> x <- rbind (xr1, xr2) ## binds the vectors into rows of a matrix (2X3)
> x
    [,1] [,2] [,3]
xr1    6    2   10
xr2    1    3   -2

> y <- cbind(xr1, xr2) ## binds the same vectors into columns of a matrix(3X2)
> y
     xr1 xr2
[1,]  6  1
[2,]  2  3
[3,]  10  -2
```

Create a matrix using dim(): Create a vector and add the dimensions using the dim ( ) function.It's especially useful if you have your data already in a vector.

Example: A vector with the numbers 1 through 12, like this:

```
> x <- 1:12
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

You can easily convert that vector to an array exactly like "x" simply by assigning the dimensions, like this:

```
> dim(x)<-c(4,3)
> x
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Using matrix subscripts :- You can identify rows, columns, or elements of a matrix by using subscripts and brackets. X[i,] refers to the ith row of matrix X, X[,j] refers to jth column, and X[i, j] refers to the ijth element, respectively

```
> x <- matrix(1:10, nrow=2) # create a matrix with 2 rows
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> x[2,]           # Displays the second row elements
[1]  2  4  6  8 10
> x[,2]           # Displays the second column elements
[1] 3 4
> x[1,4]          # Displays 1st row 4th col element
[1] 7
> x[1, c(4,5)]    # Displays 1st row, 4th and 5th elements
[1] 7 9
>z[,2:3]          # Displays 2nd and 3rd columns values
```

Matrix operations:

```
> A <- matrix(c( 6, 1, 0, -3),2, 2, byrow = TRUE)
> B <- matrix(c( 4, 2, 0, 1),2, 2, byrow = TRUE)
> A
     [,1] [,2]
[1,]    6    1
[2,]    0   -3
> B
     [,1] [,2]
[1,]    4    2
[2,]    0    1

> A+B          #Addition of a matrices
     [,1]  [,2]
[1,]   10    3
[2,]    0   -2
> A – B             #Subtraction of a matrices
     [,1] [,2]
[1,]    2   -1
[2,]    0   -4
> A * B    # this is component-by-component multiplication, not matrix multiplication
     [,1] [,2]
[1,]   24    2
[2,]    0   -3
> t(A)      #Transpose of a matrix
     [,1] [,2]
[1,]    6    0
[2,]    1   -3
> A %*% B
     [,1] [,2]
[1,]   24   13
[2,]    0   -3
```

Apply functions on matrices:- apply(), which instructs R to call a user-specified function on each of the rows or each of the columns of a matrix.

Using the apply() Function

This is the general form of apply for matrices: ***apply(m,dimcode,f,fargs)*** where the arguments are :

- m is the matrix.
- dimcode is the dimension, equal to 1 if the function applies to rows or 2 for columns.
- f is the function to be applied.
- fargs is an optional set of arguments to be supplied to f.

For example, here we apply the R function mean() to each column of a matrix A:

```
> A
     [,1] [,2]
[1,]    6    1
[2,]    0   -3
> apply(A,1,mean)
[1]  3.5 -1.5
```

Arrays:- An array is essentially a multidimensional vector. It must all be of the same type and individual elements are accessed in a similar fashion using brackets. The first element is the row index, the second is the column index and the remaining elements are for outer dimensions.

The array function takes a dim attribute which creates the required number of dimension. In the below example we create an array with two elements which are 2x3 matrices each. Creating an array:

```
> m< - array(1:12, dim=c(2,3,2))
> m
, , 1

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

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

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

In the above example, "my.array" is the name of the array we have given. There are 12 units in this array mentioned as "1:12" and are divided in three dimensions "(2, 3, 2)".

Alternative: with existing vector and using dim() *: my.vector<- 1:24*
To convert my.vector vector to an array exactly like my.array simply by assigning the dimensions, like this: *> dim(my.vector) <- c(3,4,2)*

Accessing elements of the array :-

- m[1, ,]   # Display every first row of the matrices
```
     [,1] [,2]
[1,]    1    7
[2,]    3    9
[3,]    5   11
```
- m[1, ,1] # Display first row of matrix 1
```
[1] 1 3 5
```
- m[, , 1] # Display first matrix
```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Data Frames:- Data frames are tabular data objects. Has both rows and columns analogous to excel spreadsheet. 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. It displays data along with header information.
A data frame is created with the data.frame() function : *mydata <- data.frame(col1, col2, col3,…)*
where *col1, col2, col3,* … are column vectors of any type (such as character, numeric,or logical). Names for each column can be provided with the names function.

```
# Create the data frame.
> BMI <- data.frame(gender = c("M", "M","F"),
                height = c(152, 171.5, 165),
                weight = c(81,93, 78),
                Age =c(42,38,26))
>print(BMI)
  gender  height  weight Age
1   M      152.0   81     42
2   M      171.5   93     38
3  F       165.0   78     26
```

```
> stu.ht <- c( 66, 62, 63, 70, 74)
> stu.gpa < - c( 3.80, 3.78, 3.88, 3.72, 3.69)
> student.data < - data.frame(stu.ht, stud.gpa)
> student.data
   student.ht   student.gpa
1       66         3.80
2       62         3.78
3       63         3.88
4       70         3.72
5       74         3.69
```

Example:To retrieve the cell value from the first row, second column of mtcars.
>mtcars[1,2]

- ***Function can be used with dataframes:-***
    - *nrow(BMI)*   *#Displays the no. of rows in a dataframe*
      [1] 3
    - *ncol(BMI)*   *#Displays the no. of columns in a dataframe*
      `[1] 4`
    - *dim(BMI)*   *#Displays the dimensions of dataframe*
      [1] 3 4
    - *names(BMI)*   *#Displays the names of dataframe*
      [1] "gender" "height" "weight" "Age"
    - *names(BMI)[3]*   *#Displays the name of 3rd column in dataframe*
      [1] "weight"

- *rownames(BMI)* *# Display rownames of data frame*

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

- *rownames(BMI) <- c("One","Two","Three")* *# Assigning rownames to dataframe*
- *rownames(BMI)* *# Display rownames of data frame*

```
[1] "One"  "Two"  "Three"
BMI
          gender  height weight Age
One     Male   152.0    81    42
Two     Male   171.5    93    38
Three Female  165.0    78    26
```

- *rownames(BMI) <- NULL* *#Row names are assigned as NULL*

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

- *head(BMI,2)* *# Displays first row, where n=2 so display 1st two lines of dataframe*

```
  gender height weight Age
1   Male  152.0    81  42
2   Male  171.5    93  38
```

- *tail(BMI,1)* *# Displays last row, where n=1 so display last line of dataframe*

```
   gender height weight Age
3 Female    165     78     26
```

- *class(BMI)*

```
[1] "data.frame"
```

- *BMI$weight* *# Dataframe columns can be accessed with column name with $ symbol*

```
[1] 81 93 78
```

- *To retrieve data in a particular cell:* Enter its row and column coordinates in the single square bracket "[ ]" operator.
  - *BMI[2,3]*

    ```
    [1] 93
    ```

  - *BMI[2,2:3]* *# Displays 2nd row, 2nd and 3rd column values*

    ```
        height   weight
      2  171.5     93
    ```

  - *BMI[2,]* *# Displays 2nd row*

    ```
        gender height weight Age
      2   Male  171.5     93  38
    ```

  - BMI[,c("weight","gender")] *# Displays weight and gender column*

    ```
     weight gender
    1     81   Male
    2     93   Male
    3     78   Female
    ```

  - View(BMI) *# Displays in a table format*

Lists: A list is a R object which can contain many different types of elements inside it like vectors, matrices, data frames, functions and even other lists.

mylist <- list(object1, object2, …) where the objects are any of the structures seen so far.

Example:-

```
>list1 <- list(c(2,5,3),21.3,sin)  # Create a list.
>print(list1)                      # Print the list.
        [[1]]
         [1] 2 5 3
        [[2]]
        [1] 21.3
        [[3]]
        function (x)  .Primitive("sin")
```

Naming the objects in a list:

mylist <- list(name1=object1, name2=object2, …) *or* names(mylist) <- c("name1","name2",….)

Example:-
```
> g <- "My First List"
> h <- c(25, 26, 18, 39)
> j <- matrix(1:10, nrow=5)
> k <- c("one", "two", "three")
> mylist <- list(title=g, ages=h, j, k)
> mylist
$title
[1] "My First List"

$ages
[1] 25 26 18 39

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

[[4]]
[1] "one"   "two"   "three"
```

Accessing lists:- *Double square brackets are used for accessing elements of the list*

```
✓ > names(mylist)          # Displays names of the list
  [1] "title" "ages"  ""     ""
✓ > mylist[["ages"]]       # Displays ages vector
  [1] 25 26 18 39
✓ > mylist[[2]]            # Displays 2 element in the list
  [1] 25 26 18 39
✓ > length(mylist)         # Displays length of the list
  [1] 4
✓ > mylist[[3]][,2]        # Displays 3rd element i.e, matrix 2nd column
  [1]  6  7  8  9 10
✓ > mylist[[5]] <- 3:6     # Adds a new element in the list
  > length(mylist)
  [1] 5
✓ > mylist[[3]] <- NULL    # Removes 3rd element from the list
✓ > class(mylist)          # Displays datatype
  [1] "list"
✓ > x <- list(1:6,'a')     # converting list to vector
  > y <- unlist(x)
  > y
  [1] "1" "2" "3" "4" "5" "6" "a"
```

Factors:- A factor is a statistical datatype that stores categorical variables, which is used in statistical modelling. A categorial variable is one that has two or more categories, but their is no intrinsic ordering to the categories. Eg: male and female. An R factor might be viewed simply as a vector with a bit more information added which consists of a record of the distinct values in that vector, called levels.

```
> x <- c(1,2,4,56,1,4,6)
> factor(x)                  # Displays factors of the object
      [1] 1  2  4  56 1  4  6
      Levels: 1 2 4 6 56
> y <- factor(c('M','F','M'))
> levels(y)                  # Levels of the factor variable
     [1] "F" "M"
> levels(y) <- c('M','F')    # Levels of the factor variable can be set using levels( ) function
> y
      [1] F M F
      Levels: M F
> summary(y)                 # Summary of the factor variable
     M F
     1 2
> str(y)                     # Structure of the factor object
      Factor w/ 2 levels "M","F": 2 1 2
> summary(x)                 # Summary of the object
     Min.  1st Qu.  Median   Mean  3rd Qu.   Max.
     1.00    1.50     4.00   10.57   5.00    56.00
```

Classes:- R possesses a simple generic function mechanism which can be used for an object-oriented style of programming. Method dispatch takes place based on the class of the first argument to the generic function.

Usage

```
class(x)
class(x) <- value
unclass(x)
inherits(x, what, which = FALSE)
oldClass(x)
oldClass(x) <- value
```

Arguments

X	a **R** object
what, value	a character vector naming classes. value can also be NULL.
Which	logical affecting return value: see 'Details'.

Details

Here, we describe the so called "S3" classes (and methods). For "S4" classes (and methods), see 'Formal classes' below.

Many **R** objects have a class attribute, a character vector giving the names of the classes from which the object *inherits*. (Functions oldClass and oldClass<- get and set the attribute, which can also be done directly.)

If the object does not have a class attribute, it has an implicit class, notably "matrix", "array", "function" or "numeric" or the result of typeof(x) (which is similar to mode(x)), but for type "language" and mode "call", where the following extra classes exist for the corresponding function calls: if, while, for, =, <-, (, {, call.

Note that NULL objects cannot have attributes (hence not classes) and attempting to assign a class is an error.

When a generic function fun is applied to an object with class attribute c("first", "second"), the system searches for a function called fun.first and, if it finds it, applies it to the object. If no such function is found, a function called fun.second is tried. If no class name produces a suitable function, the function fun.default is used (if it exists). If there is no class attribute, the implicit class is tried, then the default method.

Formal classes

An additional mechanism of *formal* classes, nicknamed "S4", is available in package **methods** which is attached by default. For objects which have a formal class, its name is returned by class as a character vector of length one and method dispatch can happen on *several* arguments, instead of only the first. However, S3 method selection attempts to treat objects from an S4 class as if they had the appropriate S3 class attribute, as does inherits. Therefore, S3 methods can be defined for S4 classes. See the 'Introduction' and 'Methods_for_S3' help pages for basic information on S4 methods and for the relation between these and S3 methods.

The replacement version of the function sets the class to the value provided. For classes that have a formal definition, directly replacing the class this way is strongly deprecated. The expression as(object, value) is the way to coerce an object to a particular class.

The analogue of inherits for formal classes is is. The two functions behave consistently with one exception: S4 classes can have conditional inheritance, with an explicit test. In this case, is will test the condition, but inherits ignores all conditional superclasses.

Functions oldClass and oldClass<- behave in the same way as functions of those names in S-PLUS 5/6, *but* in **R** UseMethod dispatches on the class as returned by class (with some interpolated classes: see the link) rather than oldClass. *However*, group generics dispatch on the oldClass for efficiency, and internal generics only dispatch on objects for which is.object is true.

In older versions of **R**, assigning a zero-length vector with class removed the class: it is now an error (whereas it still works for oldClass). It is clearer to always assign NULL to remove the class.

Examples

```
x <- 10
class(x) # "numeric"
oldClass(x) # NULL
inherits(x, "a") #FALSE
class(x) <- c("a", "b")
inherits(x,"a") #TRUE
inherits(x, "a", TRUE) # 1
inherits(x, c("a", "b", "c"), TRUE) # 1 2 0
class( quote(pi) )          # "name"
## regular calls
class( quote(sin(pi*x)) )    # "class"
## special calls
class( quote(x <- 1) )       # "<-"
class( quote((1 < 2)) )      # "("
class( quote( if(8<3) pi ) ) # "if"
```

Exercises

1. Add another line of code to calculate the sum of 15 and 32 in the following console. Add another comment Calculate 15 + 32 before the summation.
2. Calculate 2 to the power 5 using the ^ operator and calculate 28 modulo 6 using the % operator in the following console.
3. Write a code to assign the value 42 to a variable x in the editor. When users ask R to print x, the value 42 appears.
4. (a) Suppose we have a fruit basket with 20 apples. Store the number of apples in a variable my_apples.
   (b)Every tasty fruit basket needs oranges, so we decide to add six oranges. As a data analyst, the reflex is to immediately create a variable my_oranges and assign the value 6 to it. Next, calculate how many pieces of fruit we have in total in the variable my_fruit.
5. Follow the instructions given below and apply it in R console:
   (a) List the contents of the workspace to check that the workspace is empty.
   (b) Clear an entire workspace.
   (c) Create a variable, horses, equal to 3.
   (d) Create another variable, dogs, and set it to 7.
   (e) Create a new variable, animal that is equal to the sum of horses and dogs.
   (f) Inspect the contents of the workspace again to see that these three variables are available. (g) Eliminate the dogs variable from the workspace.
   (h) Finally, inspect the objects in the workspace once more to see that only horses and animals remain.
6. Compute the volume of a donut. The volume of a donut can be expressed as V = 2Π2r2R where r is the minor radius and R is the major radius. This is the same as computing the area of the cylindrical portion of the donut (Πr2) and multiplying it by the circumference of the donut (2Πr).
7. A horse is grazing in a rectangular field of length 25m and width 10m. It is tethered with a rope half as long as the width of the field. What is the area of the field that the horse is unable to graze? Hint: Calculate the Area = (1/4)*(22/7)*5*5= 19.64m
8. Carry out the following operations:
   (a) Assign 42 to a variable named my_numeric.
   (b) Initialize a variable my_character to “forty-two”.
   (c) Set a variable my_logical to FALSE
9. Perform the following operations in R:
   (a) Create a variable called my_apples and assign it a value.
   (b) Display the value of my_apples.
   (c) Create a variable called my_oranges, initialize it with a text value, and display it.

(d) To the variable my_fruit, assign the addition of a numeric and a character variable.

10. Implement the following operations using R:
    (a) Create a logical variable var
    (b) Create a numeric variable var2.
    (c) Create a string variable var3.
    (d) Assign var1 to var1_char by converting it into a character variable.
    (e) Making use of the is.character() function, verify whether var1_char is in fact a character.
    (f) Assign var2 to var2_log by converting it to a logical variable.
    (g) Display the class of var2_log using the class() function.
    (h) Does coercing var3 to a numeric type and subsequently assigning to var3_num prove to be successful?
11. Perform the following operations using R:
    (a) Initialize 3 character variables named age, employed, and salary.
    (b) Transform age to a numeric type and store in the variable age_clean.
    (c) Initialize employed_clean with the result obtained by converting employed to logical type.
    (d) Convert the respondent's salary to a numeric and store it in the variable salary_clean.
12. Create a vector named boolean_vector, with three elements TRUE, FALSE, and TRUE.
13. Write an R code to perform the task of record maintenance where the following information is recorded.
    (a) Each day's total gain or loss in the variable total_daily.
    (b) Overall profit or loss per day of the week.
    (c) Total money lost over the week.
    (d) The overall status of the gain/loss in poker or in roulette.

14 Perform the following operations in R:
    (a) Create two vectors namely vector1 containing 3 elements and vector2 containing 6 elements.
    (b) Convert vectors into array of two 3 * 3 matrices. Give appropriate dimnames.
    (c) Extract the 3rd row of the 2nd matrix of this array.
    (d) Extract the element belonging to the 1st row and 3rd column of the 1st array.
    (e) Extract the 2nd matrix of the array.

15 Perform the following operations in R:
    (a) Create a 2 * 3 matrix with values as 5.
    (b) Create a 2 * 3 matrix with values as 3.
    (c) Create a 2 * 3 * 2 array with the 1st level [,,1] populated with mat1 (5's),and the 2nd level [,,2] populated with mat2 (3's).
    (d) Display the array.

16 Perform the following in R:
    (a) Create a vector of length 5 containing values either as TRUE or FALSE.
    (b) Create another vector of length 4 containing numeric values.
    (c) Create another vector containing the names of 3 students.
    (d) For each of the vectors, check for the class they belong to.

17 Create a list with student details (such as name, usn, gpa) and define its class as Student. Display the details of each student in the list including the class name.

R Programming Structures: Control Statements - Loops, Looping Over Non-vector Sets, If-Else, Arithmetic and Boolean Operators and values, Default Values for Argument, Return Values - Deciding Whether to explicitly call return, Returning Complex Objects, Functions are Objective, No Pointers in R, Recursion - A Quick sort Implementation, Extended Extended Example: A Binary Search Tree.

Control Statements: The statements in an R program are executed sequentially from the top of the program to the bottom. But some statements are to be executed repetitively, while only executing other statements if certain conditions are met. R has the standard control structures.

Loops: Looping constructs repetitively execute a statement or series of statements until a condition isn't true. These include the for, while and repeat structures with additional clauses break and next.

1) **FOR** :- The for loop executes a statement repetitively until a variable's value is no longer contained in the sequence seq.

- The syntax is
```
for (var in sequence)
{
        statement
}
```
  Here, sequence is a vector and var takes on each of its value during the loop. In each iteration, statement is evaluated.
- for (n in x) { - - - }
  It means that there will be one iteration of the loop for each component of the vector x, with taking on the values of those components—in the first iteration, n = x[1]; in the second iteration, n = x[2]; and so on.
- In this example *for (i in 1:10) print("Hello")* the word Hello is printed 10 times.

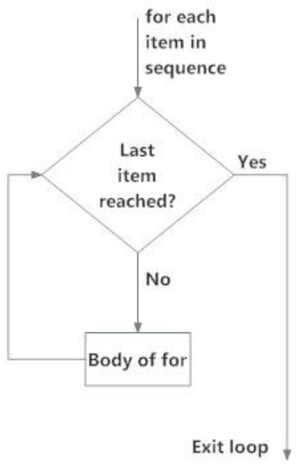


Fig: operation of for loop

- Square of every element in a vector:
```
> x <- c(5,12,13)
> for (n in x)   print(n^2)
[1] 25
[1] 144
[1] 169
```

- Program to find the multiplication
```
# take input from the user
        num = as.integer(readline(prompt = "Enter a number: "))
# use for loop to iterate 10 times
        for(i in 1:10) {
                print(paste(num,'x', i, '=', num*i)) }
```

2) **WHILE:-** A while loop executes a statement repetitively until the condition is no longer true.

Syntax:
```
while (expression)
{
        statement
}
```
- Here, expression is evaluated and the body of the loop is entered if the result is TRUE.

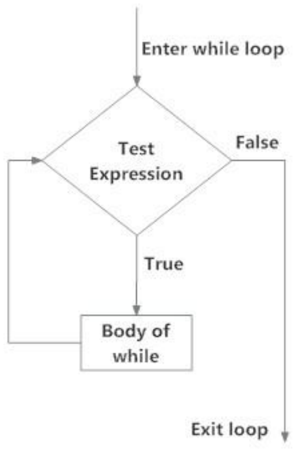


Fig: operation of while loop

- The statements inside the loop are executed and the flow returns to evaluate the expression again.
- This is repeated each time until expression evaluates to FALSE, in which case, the loop exits.
- Example

```
> i <- 1
> while (i<=10)           i <- i+4
> i
[1] 13
```

- Program to find the sum of first n natural numbers

```
sum = 0
# take input from the user
num = as.integer(readline(prompt = "Enter a number: "))
# use while loop to iterate until zero
   while(num > 0)
   {
      sum = sum + num
      num = num - 1
   }
   print(paste("The sum is", sum))
```

Output:

```
Enter a number: 4
[1] "The sum is 10"
```

3) Break statement: A break statement is used inside a loop (repeat, for, while) to stop the iterations and flow the control outside of the loop.In a nested looping situation, where there is a loop inside another loop, this statement exits from the innermost loop that is being evaluated.

Syntax:- **break**

Example

```
x <- 1:5
for (val in x) {
   if (val == 3){
      break
   }
   print(val)
}
```

Output:

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

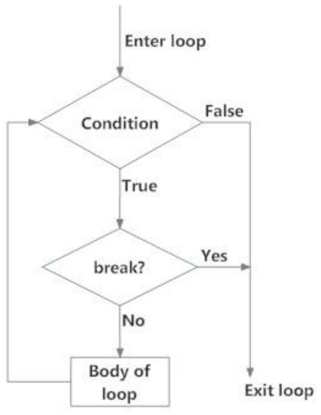


Fig: flowchart of break

4) Next statement:- A next statement is useful when we want to skip the current iteration of a loop without terminating it. On encountering next, the R parser skips further evaluation and starts next iteration of the loop.

Syntax:- **next**

Example

```
x <- 1:5
for (val in x) {
   if (val == 3){
      next
   }
   print(val)
}
```

Output:

```
[1]  1
[1]  2
[1]  4
[1]  5
```

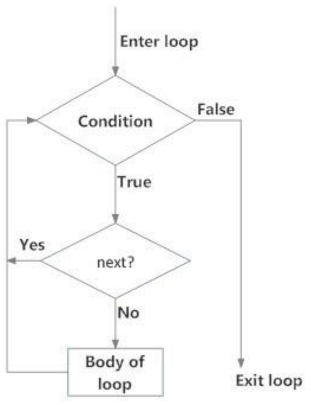


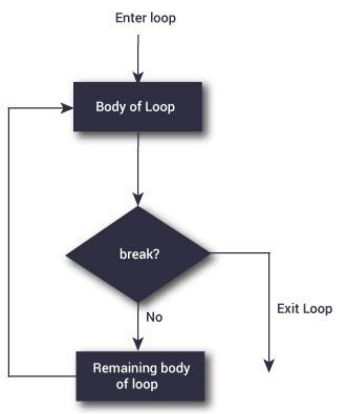
Fig: flowchart of next

5) Repeat:- Repeat loop is used to iterate over a block of code multiple number of times. There is no condition check in repeat loop to exit the loop.

We must ourselves put a condition explicitly inside the body of the loop and use the break statement to exit the loop. Failing to do so will result into an infinite loop.

Syntax:

```
repeat
{
        statement
}
```



Example:

```
x <- 1
repeat
{
        print(x)
        x = x+1
        if (x == 6)
                break
}
```

Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Looping Over Non-vector Sets:- R does not directly support iteration over nonvector sets, but there are a couple of indirect yet easy ways to accomplish it:

- ***apply( ):-*** Applies on 2D arrays (matrices), data frames to find aggregate functions like sum, mean, median, standard deviation.
  syntax:- apply(matrix,margin,fun,....)
  margin = 1 indicates row
  = 2 indicates col

```
> x <- matrix(1:20,nrow = 4,ncol=5)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> apply(x,2,sum)
[1] 10 26 42 58 74
```

- Use ***lapply( )***, assuming that the iterations of the loop are independent of each other, thus allowing them to be performed in any order. Lapply( ) can be applies on dataframes,lists and vectors and return a list. lapply returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.
  Syntax: lapply(X, FUN, ...)

```
> x <- matrix(1:4,2,2)
> x
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> lapply(x,sqrt)
[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051
```

lapply( ) function is applied on every elements of the object.

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

- Use ***get( ),*** As its name implies, this function takes as an argument a character string representing the name of some object and returns the object of that name. It sounds simple, but get() is a very powerful function.

  Syntax: get("character string")

```
> get("sum")
   function (..., na.rm = FALSE)  .Primitive("sum")
> get("g")
     function(x)
     {
       return(x+1)
     }
> get("num")
     [1] "45"
```

Note:- Reserved words in R programming are a set of words that have special meaning and cannot be used as an identifier (variable name, function name etc.).This list can be viewed by typing help(reserved) or ?reserved at the R command prompt.

Reserved words in R

if	*else*	*repeat*	*while*	*function*
for	*in*	*next*	*break*	*TRUE*
FALSE	*NULL*	*Inf*	*NaN*	*NA*
*NA_integer_*	*NA_real_*	*NA_complex_*	*NA_character_*	*...*

If –Else:- The if-else control structure executes a statement if a given condition is true. Optionally, a different statement is executed if the condition is false.

The syntax is

```
if (cond)
{
        statements
}
```

```
if (cond)
{
        statement1
} else
{
        statement2
}
```

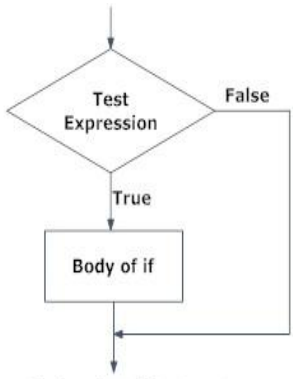


Fig: Operation of if statement

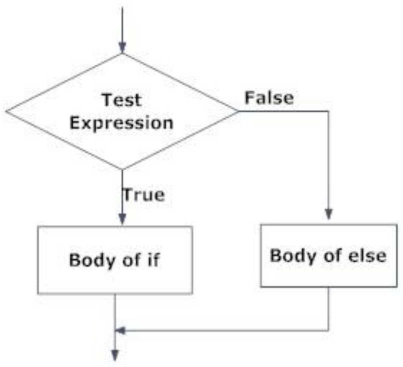


Fig: Operation of if...else statement

```
x  <-  8
if(x>3)  {y  <-  10
}  else  {y<-0}
print(y)
```

Output:
```
[1]  10
```

```
y <- ifelse(x>3, 10, 0)
y
```

Output:
```
[1]  0
```

```
x  <-  4
if(x==4)
  x  <-  1
else
{
  x  <-  3
  y  <-  4
}
```

Output: Error.
The right brace before the else is used by the R parser to deduce that this is an if-else rather than just an if.

An if-else statement works as a function call, and as such, it returns the last value assigned.

v <- if (cond) expression1 else expression2

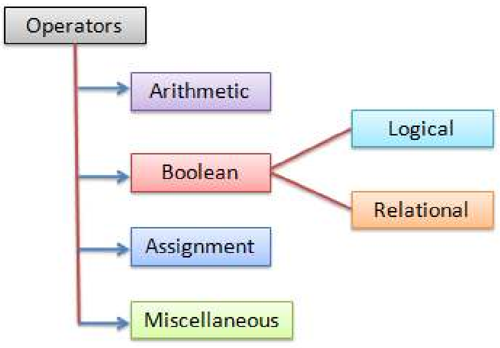
This will set v to the result of expression1 or expression2, depending on whether cond is true. You can use this fact to compact your code. Here's a simple example:

```
> x <- 2
> y <- if(x == 2)  x    else  x+1
> y
[1] 2

> x <- 2
>if(x == 2) y <- x   else  y <- x+1
> y
[1] 2
```

Operators:- R has many operators to carry out different mathematical and logical operations.

Types of operators

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



1.Arithmetic operators:- These operators are used to carry out mathematical operations like addition and multiplication. Here is a list of arithmetic operators available in R.

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

2.Relational Operator:- Relational operators are used to compare between values.Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example

>	Checks if each element of the first vector is greater than the corresponding element of the second vector.	v <- c(2,5.5,6,9)
		t <- c(8,2.5,14,9)
		print(v>t)
		it produces the following result –
		[1] FALSE TRUE FALSE FALSE

<	Checks if each element of the first vector is less than the corresponding element of the second vector.	v <- c(2,5.5,6,9)
		t <- c(8,2.5,14,9)
		print(v < t)
		it produces the following result –
		[1] TRUE FALSE TRUE FALSE

==	Checks if each element of the first vector is equal to the corresponding element of the second vector.	v <- c(2,5.5,6,9)
		t <- c(8,2.5,14,9)
		print(v == t)
		it produces the following result −
		[1] FALSE FALSE FALSE TRUE

<=	Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.	v <- c(2,5.5,6,9)
		t <- c(8,2.5,14,9)
		print(v<=t)
		it produces the following result −
		[1] TRUE FALSE TRUE TRUE

>=	Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.	v <- c(2,5.5,6,9)
		t <- c(8,2.5,14,9)
		print(v>=t)
		it produces the following result −
		[1] FALSE TRUE FALSE TRUE

!=	Checks if each element of the first vector is unequal to the corresponding element of the second vector.	v <- c(2,5.5,6,9)
		t <- c(8,2.5,14,9)
		print(v!=t)
		it produces the following result −
		[1] TRUE TRUE TRUE FALSE

3)Logical Operators:- It is applicable only to vectors of type logical, numeric or complex. Zero is considered FALSE and non-zero numbers are taken as TRUE. Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example

&	It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE.	v <- c(3,1,TRUE,2+3i)
		t <- c(4,1,FALSE,2+3i)
		print(v&t)
		it produces the following result −
		[1] TRUE TRUE FALSE TRUE

|	It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if one the elements is TRUE.	v <- c(3,0,TRUE,2+2i)
		t <- c(4,0,FALSE,2+3i)
		print(v|t)
		it produces the following result −
		[1] TRUE FALSE TRUE TRUE

!	It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value.	v <- c(3,0,TRUE,2+2i)
		print(!v)
		it produces the following result −
		[1] FALSE TRUE FALSE FALSE

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

Operator	Description	Example

&&	Called Logical AND operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE.	v <- c(3,0,TRUE,2+2i)
		t <- c(1,3,TRUE,2+3i)
		print(v&&t)
		it produces the following result –
		[1] TRUE

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

4) Assignment Operators:- These operators are used to assign values to vectors.

Operator	Description	Example

<−
or
=
or
<<−
	Called Left Assignment	v1 <- c(3,1,TRUE,2+3i)
		v2 <<- c(3,1,TRUE,2+3i)
		v3 = c(3,1,TRUE,2+3i)
		print(v1)
		print(v2)
		print(v3)
		it produces the following result –
		[1] 3+0i 1+0i 1+0i 2+3i
		[1] 3+0i 1+0i 1+0i 2+3i
		[1] 3+0i 1+0i 1+0i 2+3i

-> or ->>	Called Right Assignment	c(3,1,TRUE,2+3i) -> v1 c(3,1,TRUE,2+3i) ->> v2 print(v1) print(v2) it produces the following result – [1] 3+0i 1+0i 1+0i 2+3i [1] 3+0i 1+0i 1+0i 2+3i

5) Miscellaneous Operators:- These operators are used to for specific purpose and not general mathematical or logical computation.

Operator	Description	Example

:	Colon operator. It creates the series of numbers in sequence for a vector.	v <- 2:8
		print(v)
		it produces the following result –
		[1] 2 3 4 5 6 7 8

%in%	This operator is used to identify if	v1 <- 8
		v2 <- 12
		t <- 1:10
		print(v1 %in% t)

	an element belongs to a vector.	print(v2 %in% t) it produces the following result – [1] TRUE [1] FALSE

%*%	This operator is used to multiply a matrix with its transpose.	M = matrix(c(2,6,5,1,10,4), nrow = 2,ncol = 3,byrow = TRUE)
		t = M %*% t(M)
		print(t)
		it produces the following result –
		     [,1] [,2]
		[1,] 65 82
		[2,] 82 117

Functions:- A function is a block or chunck of code having a specific structure, which is often singular or atomic nature, and can be reused to accomplish a specific nature. A function helps to divide a large program into modules to enhance readability and code reuse. or A function is a group of instructions that takes inputs, uses them to compute other values, and returns a result.

Built-in functions in R

structure of a function

```
function_name <- function(arguments)
{
        statements
}
```

- The word 'function' is a keyword which is used to specify the statements enclosed within the curly braces are part of the function.
- Function_name is used to identify the function
- Function consists of formal arguments and body
- The function is called using the following statement: *function_name(arguments)*

Example:

```
say.hello <- function()
{
        print("Hello, World!")
}
say.hello()
  [1] "Hello, World!"
```

```
g <- function(x)
{
  x<- x+1
  return(x)
}
g(2)
  [1] 3
```

- ✓ Functions are assigned to objects just like any other variable, using <- operator.
- ✓ Function are a set of parentheses that can either be empty – not have any arguments – or contain any number of arguments.
- ✓ The body of the function is enclosed in curly braces ({ and }).This is not necessary if the function contains only one line.
- ✓ A semicolon(;) can be used to indicate the end of the line but is not necessary.

```
# counts the number of odd integers in x
                > oddcount <- function(x)
                {
                  k <- 0                              # assign 0 to k
                  for (n in x) {
                  if (n %% 2 == 1) k <- k+1           # %% is the modulo operator
                  }
```

```
 return(k)
}
> oddcount(c(1,3,5))
    [1] 3
> oddcount(c(1,2,3,7,9))
    [1] 4
```

Variables created outside functions are global and are available within functions as well. Example:

```
> f <- function(x) return(x+y)
> y <- 3
> f(5)
[1] 8
```

Here y is a global variable. A global variable can be written to from within a function by using R's superassignment operator, <<-.

Default Arguments:- R also makes frequent use of *default arguments*. Consider a function definition like this:

```
> g <- function(x,y=2,z=T) { ... }
```

Here y will be initialized to 2 if the programmer does not specify y in the call. Similarly, z will have the default value TRUE. Now consider this call:

```
> g(12,z=FALSE)
```

Here, the value 12 is the actual argument for x, and we accept the default value of 2 for y, but we override the default for z, setting its value to FALSE.

Default Values for Arguments:-

```
> my_matrix<- matrix (1:12,4,3,byrow= TRUE)
```

The argument *byrow= TRUE* tells R that the matrix should be filled in row wise. In this example the default argument is byrow = FALSE, the matrix is filled in column wise.

Lazy Evaluation of Function:- Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.

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

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

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

Return Values:- Functions are generally used for computing some value, so they need a mechanism to supply that value back to the caller.This is called returning.

- The return value of a function can be any R object. Although the return value is often a list, it could even be another function.
- You can transmit a value back to the caller by explicitly calling return(). Without this call, the value of the last executed statement will be returned by default.
- If the last statement in the call function is a for( ) statement, which returns the value NULL.

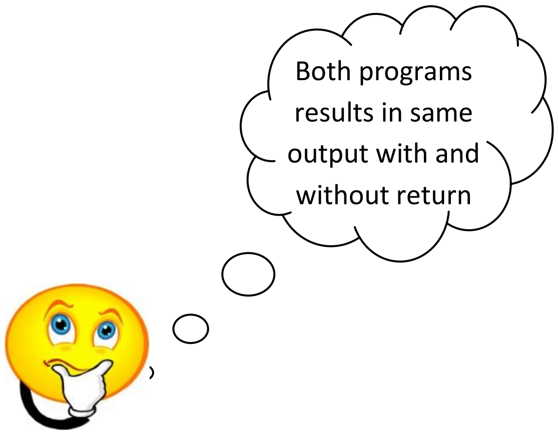
```
 # First build it without an explicit return
 num <- function(x)
 {
    x*2
 }
> num(5)
[1] 10
```

```
 # Now build it with an explicit return
 num <- function(x)
 {
     return(x*2)
 }
> num(5)
[1] 10
```

```
# build it again, this time with another argument after the explicit return
num <- function(x)
{
    return(x*2)
        #below here is not executed because the return function already exists.
    print("VISHNU")
    return(17)
 }
> num(5)
[1] 10

# if the last statement is for loop or any empty statement then it return NULL.
num <- function(x)
{   }
> num(5)
[1] NULL
```

Deciding Whether to Explicitly Call return():-The R idiom is to avoid explicit calls to return(). One of the reasons cited for this approach is that calling that function lengthens execution time. However, unless the function is very short, the time saved is negligible, so this might not be the most compelling reason to refrain from using return(). But it usually isn't needed.

```
#Example to count the odd numbers with no return statement
oddcount <- function(x) {
 k <- 0
 for (n in x) {
   if (n %% 2 == 1) k <- k+1
 }
 k
}
> oddcount(c(12,2,5,9,7))
[1] 3

#Example to count the odd numbers with return statement
oddcount <- function(x) {
 k <- 0
 for (n in x) {
   if (n %% 2 == 1) k <- k+1
 }
return(k)
}
> oddcount(c(12,2,5,9,7))
[1] 3
```


Both programs results in same output with and without return

Good software design, can glance through a function's code and immediately spot the various points at which control is returned to the caller. The easiest way to accomplish this is to use an explicit return() call in all lines in the middle of the code that cause a return.

Returning Complex Objects:- The return value can be any R object, you can return complex objects. Here is an example of a function being returned:

```
g<-function() {
        x<- 3
        t <- function(x) return(x^2)
        return(t)
    }

    > g()
function(x) return(x^2)
<environment: 0x16779d58>
```

If your function has multiple return values, place them in a list or other container.

Functions are Objective:- R functions are first-class objects (of the class "function"), meaning that they can be used for the most part just like other objects. This is seen in the syntax of function creation:

```
g <- function(x)
{
   return(x+1)
}
```

function( ) is a built-in R function whose job is to create functions.On the right-hand side, there are really two arguments to function(): The first is the formal argument list for the function i.e, x and the second is the body of that function return(x+1). That second argument must be of class "expression". So, the point is that the right-hand side creates a function object, which is then assigned to g.

> ?"{" Its job is the make a single unit of what could be several statements.

- ✓ formals( ) :- Get or set the formal arguments of a function

```
> formals(g)          # g is a function with formal arguments "x"
  $x
```

- ✓ body( ) :- Get or set the body of a function

```
> body(g)          # g is a function
{
  x <- x + 1
  return(x)
}
```

- ✓ Replacing body of the function: quote( ) is used to substitute expression

```
> g <- function(x)   return(x+1)
> body(g) <- quote(2*x+3)
> g
function (x)
2 * x + 3
```

- ✓ Typing the name of an object results in printing that object to the screen which is similar to all objects

```
> g
function(x)
{
   return(x+1)
}
```

- ✓ Printing out a function is also useful if you are not quite sure what an R library function does. Code of a function is displayed by typing the built-in function name.

```
> sd
function (x, na.rm = FALSE)
sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
   na.rm = na.rm))
<bytecode: 0x17b49740> <environment: namespace:stats>
```

- ✓ Some of R's most fundamental built-in functions are written directly in C, and thus they are not viewable in this manner.

```
> sum
function (..., na.rm = FALSE)  .Primitive("sum")
```

- ✓ Since functions are objects, you can also assign them, use them as arguments to other functions, and so on.

```
> f1 <- function(a,b) return(a+b)
> f2 <- function(a,b) return(a-b)
> f <- f1                          # Assigning function object to other object
> f(3,2)
[1] 5
> g <- function(h,a,b) h(a,b)     # passing function object as an arguments
> g(f1,3,2)
[1] 5
> g(f2,3,2)
[1] 1
```

No Pointers in R:- R does not have variables corresponding to pointers or references like C language. This can make programming more difficult in some cases.

The fundamental thought is to create a class constructor and have every instantiation of the class be its own environment. One can then pass the object/condition into a function and it will be passed by reference instead of by value, because unlike other R objects, environments are not copied when passed to functions. Changes to the object in the function will change the object in the calling frame. In this way, one can operate on the object and change internal elements without having to create a copy of the object when the function is called, nor pass the entire object back from the function. For large objects, this saves memory and time.

For example, you cannot write a function that directly changes its arguments.

```
> x <- c(12,45,6)
> sort(x)
[1]  6 12 45
> x
[1] 12 45  6
```

The argument to sort() does not change. If we do want x to change in this R code, the solution is to reassign the arguments:

```
> x <- sort(x)
> x
[1]  6 12 45
```

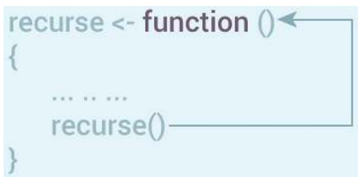
If a function has several output then a solution is to gather them together into a list, call the function with this list as an argument, have the function return the list, and then reassign to the original list.
An example is the following function, which determines the indices of odd and even numbers in a vector of integers:

```
> y <-function(v){
      odds <- which(v %% 2 == 1)
      evens <- which(v %% 2 == 0)
      list(o=odds,e=evens)
      }
> y(c(2,34,1,5))
$o
[1] 3 4

$e
[1] 1 2
```

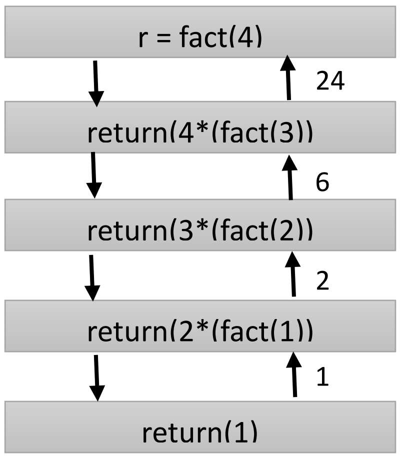
Recursion:- Recursion is a programming technique in which, a function calls itself repeatedly for some input.

To solve a problem of type X by writing a recursive function f():

1. Break the original problem of type X into one or more smaller problems of type X.
2. Within f(), call f() on each of the smaller problems.
3. Within f(), piece together the results of (b) to solve the original problem.



```
# Recursive function to find factorial
recursive.factorial <- function(x)
 {
  if (x == 0)    return (1)
  else           return (x * recursive.factorial(x-1))
}
> recursive.factorial(5)
[1] 120
```



A Quicksort Implementation:- Quick sort is also known as Partition-Exchange sort and is based on Divide and conquer Algorithm design method. This was proposed by C.A.R Hoare. The basic idea of quick sort is very simple. We consider one element at a time (pivot element). We have to move the pivot element to the final position that it should occupy in the final sorted list. While identifying this position, we arrange the elements, such that the elements to the left of the pivot element will be less than pivot element & elements to the right of the pivot element will be greater than pivot element. There by dividing the list by 2 parts. We have to apply quick sort on these 2 parts recursively until the entire list is sorted.

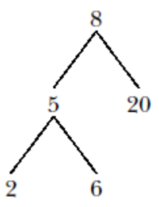
For instance, suppose we wish to sort the vector (5,4,12,13,3,8,88). We first compare everything to the first element, 5, to form two subvectors: one consisting of the elements less than 5 and the other consisting of the elements greater than or equal to 5. That gives us subvectors (4,3) and (12,13,8,88). We then call the function on the subvectors, returning (3,4) and (8,12,13,88). We string those together with the 5, yielding (3,4,5,8,12,13,88), as desired. R's vector-filtering capability and its c() function make implementation of Quicksort quite easy.

```
# Quicksort recursive function
qs <- function(x) {
        if (length(x) <= 1) return(x)
        pivot <- x[1]
        therest <- x[-1]
        sv1 <- therest[therest < pivot]
        sv2 <- therest[therest >= pivot]
        sv1 <- qs(sv1)
        sv2 <- qs(sv2)
        return(c(sv1,pivot,sv2)) }
```

```
> qs(c(12,6,7,34,3))
[1]  3  6  7 12 34
```

Binary search tree:- The nature of binary search trees implies that at any node, all of the elements in the node's left subtree are less than or equal to the value stored in this node, while the right subtree stores the elements that are larger than the value in this mode. In our example tree, where the root node contains 8, all of the values in the left subtree-5, 2 and 6-are less than 8, while 20 is greater than 8.

Here is an example:



The code follows. Note that it includes only routines to insert new items and to traverse the tree.

```
# storage is in a matrix, say m, one row per node of the tree; a link i in the tree means the vector
#m[i,] = (u,v,w); u and v are the left and right links, and w is the stored value; null links have the value
#NA; the matrix is referred to as the list (m,nxt,inc), where m is the matrix, nxt is the next empty row to
#be used, and inc is the number of rows of expansion to be allocated when the matrix becomes full

# initializes a storage matrix, with initial stored value firstval
newtree <- function(firstval,inc) {
  m <- matrix(rep(NA,inc*3),nrow=inc,ncol=3)
  m[1,3] <- firstval
  return(list(mat=m,nxt=2,inc=inc))
}

# inserts newval into nonempty tree whose head is index hdidx in the storage space treeloc; note that
#return value must be reassigned to tree; inc is as in newtree() above
ins <- function(hdidx,tree,newval,inc) {
  tr <- tree
  # check for room to add a new element
  tr$nxt <- tr$nxt + 1
  if (tr$nxt > nrow(tr$mat))
    tr$mat <- rbind(tr$mat,matrix(rep(NA,inc*3),nrow=inc,ncol=3))
  newidx <- tr$nxt  # where we'll put the new tree node
  tr$mat[newidx,3] <- newval
  idx <- hdidx  # marks our current place in the tree
  node <- tr$mat[idx,]
  nodeval <- node[3]
  while (TRUE) {
    # which direction to descend, left or right?
    if (newval <= nodeval) dir <- 1 else dir <- 2
    # descend
    # null link?
    if (is.na(node[dir])) {
      tr$mat[idx,dir] <- newidx
      break
    } else {
      idx <- node[dir]
      node <- tr$mat[idx,]
      nodeval <- node[3]
    }
  }
  return(tr)
}
```

```
# print sorted tree via inorder traversal
printtree <- function(hdidx,tree) {
  left <- tree$mat[hdidx,1]
  if (!is.na(left)) printtree(left,tree)
  print(tree$mat[hdidx,3])
  right <- tree$mat[hdidx,2]
  if (!is.na(right)) printtree(right,tree)
}
```

sapply( ):- sapply is wrapper class to lapply with difference being it returns vector or matrix instead of list object.

Syntax: `sapply(X, FUN, ...,)`

```
    # create a list with 2 elements
     x = (a=1:10,b=11:20)  # mean of values using sapply
sapply(x, mean)
  a    b
 5.5 15.5
```

tapply( ):- tapply() applies a function or operation on subset of the vector broken down by a given factor variable.

To understand this, imagine we have ages of 20 people (male/females), and we need to know the average age of males and females from this sample. To start with we can group ages by the gender (male or female), ages of 12 males, and ages of 8 females, and later calculate the average age for males and females.

Syntax of tapply: ***tapply(X, INDEX, FUN, …)***

X = a vector, INDEX = list of one or more factor, FUN = Function or operation that needs to be applied, … optional arguments for the function

```
> ages <- c(25,26,55,37,21,42)
> affils <- c("R","D","D","R","U","D")
> tapply(ages,affils,mean)
          D R U
          41 31 21
```

Doing Math and Simulation in R:- Math Function, Extended Example Calculating Probability- Cumulative Sums and Products-Minima and Maxima- Calculus, Functions Fir Statistical Distribution, Sorting, Linear Algebra Operation on Vectors and Matrices, Extended Example: Vector cross Product- Extended Example: Finding Stationary Distribution of Markov Chains, Set Operation, Input /out put, Accessing the Keyboard and Monitor, Reading and writer Files

Math Function : R contains built-in functions for various math operations and for statistical distributions.

Function	Explanation	Example
exp( )	Exponential function, base e	> exp(2) [1] 7.389056
log( )	Natural logarithm	> log(10) [1] 2.302585
log10( )	Logarithm base 10	> log10(10) [1] 1
sqrt( )	Square root	> sqrt(16) [1] 4
abs( )	Absoluate value	> abs(-12.4) [1] 12.4
sin( )	Trig functions	> sin(40) [1] 0.7451132
cos( )	Trig functions	> cos(12) [1] 0.843854
min( )	Minimum value within a vector	> x <- c(1,4,-423,8,-2,23) > min(x) [1] -423
max( )	Maximum value within a vector	> x <- c(1,4,-423,8,-2,23) > max(x) [1] 23
which.min( )	Index of minimal element of the vector	> x <- c(1,4,-423,8,-2,23) > which.min(x) [1] 3
which.max( )	Index of maximal element of the vector	> x <- c(1,4,-423,8,-2,23) > which.max(x) [1] 6
pmin( )	Element-wise minima of several vectors	> x <- c(4,-5,56) > y <- c(3,2,7) > pmin(x,y) [1] 3 -5 7
pmax( )	Element-wise maxima of several vectors	> x <- c(4,-5,56) > y <- c(3,2,7) > pmax(x,y) [1] 4 2 56
sum( )	Sum of the elements of the vector	> x [1] 4 -5 56 > sum(x) [1] 55
prod( )	Product of the elements of the vector	> y <- 1:3 > prod(y) [1] 6
cumsum( )	Cumulative sum of the elements of a vector	> y [1] 1 2 3 > cumsum(y)

		[1] 1 3 6
cumprod( )	Cumulative product of the elements of a vector	> z <- c(2,5,3) > cumprod(z) [1] 2 10 30
round( )	Round of the closest integer	> round(12.4) [1] 12 > round(2.43,digits=1) [1] 2.4
floor( )	Round of the closest integer below	> floor(12.4) [1] 12
ceiling( )	Round of the closest integer above	> ceiling(12.4) [1] 13
factorial( )	Factorial function	> factorial(5) [1] 120

sin(), cos(), tan() and so on: Trig functions, the arguments will be in radians,asin(), acos(), atan() inverse trignometry functions.

```
> tan(45*pi/180)
     [1] 1
> a<-tan(45*pi/180)
> b<-atan(a)
> b
   [1] 0.7853982
> b*180/pi
   [1] 45
```

sum(): sum returns the sum of all the values present in its arguments.

sum(..., na.rm = FALSE)

... : numeric or complex or logical vectors.
na.rm : logical. Should missing values (including NaN) be removed?

Example1:-

```
>x
 [1]  4 -5 56
> sum(x)
[1] 55
```

Example2:-

```
y <- c(2,3,NA,1)
>sum(y)
 [1] NA
>sum(y, na.rm=TRUE)
 [1]                                    6
```

prod(): prod returns the product of all the values present in its arguments.

prod(..., na.rm = FALSE)

... : numeric or complex or logical vectors.
na.rm : logical. Should missing values (including NaN) be removed?

Example1:-

```
> x <- c(1,3,5)
>prod(x)
 [1] 15
```

Example 2:-

```
> y
  [1] 2 3 NA 1
>prod(y)
 [1] NA
> prod(y, na.rm=TRUE)
[1] 6
```

Extended Example: Calculating a probability :

Now we see how to find the probability that exactly one event occur: If three friends x, y, z appeared for an examination x has17% chance of failure, y has 7% chance of failure, and Z has 26% chance of failure. What is the probability that exactly one of them will fail in the exams?

P(X fails, but not others) = 0.17 * 0.93 * 0.74,
P(Y fails, but not others) = 0.83 * 0.07 * 0.74,
P(Z fails, but not others) = 0.83 * 0. 93 * 0.26.

The probability can be calculated using the prod() function. Let us assume that there are ‘n’ independent events with the ith event having the p_i probability of occurrence.
What is the probability of exactly one of these events occurring?

Considering an example where the value of n is 3. The events are named A, B, and C. Then we break down the computation as follows:
P(exactly one event occurs) = P(A and not B and not C) +
P(not A and B and not C) +
P(not A and not B and C)
P(A and not B and not C) would be $p_A(1 - p_B)(1 - p_C)$, and so on.
For general n, that is calculated as follows

$$\sum_{i=1}^{n} p_i(1-p_1)....(1-p_{i-1})(1-p_{i+1})....(1-p_n)$$

(The ith term inside the sum is the probability that event i occurs and all the others do not occur.)
Here's code to compute this, with our probabilities pi contained in the vector p:

```
exactlyone <- function(p) {
  notp <- 1 - p
  tot <- 0.0
  for (i in 1:length(p))
    tot <- tot + p[i] * prod(notp[-i])
  return(tot)
}
```

notp <- 1 – p :- creates a vector of all the "not occur" probabilities $1 - p_j$, using recycling.
The expression notp[-i] computes the product of all the elements of notp, except the ith

Cumulative Sums and Products:-

A cumulative product is a sequence of partial products of a given sequence. For example, the cumulative products of the sequence {a,b,c,.....} are a,ab,abc , ... Returns a vector whose elements are the cumulative product.

```
> x <- c(2,4,3)
> cumprod(x)
 [1]  2  8 24
```

A cumulative sum is a sequence of partial sum of a given sequence. For example, the cumulative sum of the sequence {a,b,c,.....} are a,ab,abc , ... Returns a vector whose elements are the cumulative sum.

```
> x <- c(2,4,3)
> cumprod(x)
  [1] 2 6 9
```

Minima and maxima:-

max() function computes the maximun value of a vector.
min() function computes the minimum value of a vector.

- x: number vector
- na.rm: whether NA should be removed, if not, NA will be returned

- max(..., na.rm = FALSE)

```
> max(c(12,4,6,NA,34))
[1] NA
> max(c(12,4,6,NA,34),na.rm=FALSE)
[1] NA
> max(c(12,4,6,NA,34),na.rm=TRUE)
[1] 34
> x <- c(2,-4,6,-34)
> min(x[1],x[4])
[1] -34
```

- min(..., na.rm = FALSE)

```
> min(c(12,4,6,NA,34))
[1] NA
> min(c(12,4,6,NA,34),na.rm=TRUE)
```

```
[1] 4
> min(c(12,4,6,NA,34),na.rm=FALSE)
[1] NA
> x <- c(2,-4,6,-34)
> max(x[2],x[3])
[1] 6
```

which.min() and which.max(): Index of the minimal element and maximal element of a vector.

```
>x <- c(1,4,-423,8,-2,23)
> which.min(x)
    [1] 3
> which.max(x)
    [1] 6
```

pmin() and pmax(): Element-wise minima and maxima of several vectors.
There is quite a difference between min() and pmin(). The former simply combines all its arguments into one long vector and returns the minimum value in that vector. In contrast, if pmin() is applied to two or more vectors, it returns a vector of the pair-wise minima, hence the name pmin.
The max() and pmax() functions act analogously to min() and pmin().

- pmax(..., na.rm = FALSE)

```
> x <- c(12,4,6,NA)
> y <- c(2,34,56,1)
> pmax(x,y)
[1] 12 34 56 NA
> pmax(x,y,na.rm=TRUE)
[1] 12 34 56  1
```

- pmin(..., na.rm = FALSE)

```
> x
[1] 12  4 NA  3
> y
[1] 1 2 3 4
> pmin(x,y)
[1]  1  2 NA  3
> pmin(x,y,na.rm=TRUE)
[1] 1 2 3 3
```

pmin and **pmax** are the 'parallel' versions of the min and max function, meaning that they can take vector arguments and return vectors back.

Function minimization/maximization can be done via nlm() and optim(). For example, let's find the smallest value of f(x) = x2 − sin(x).

```
> nlm(function(x) return(x^2-sin(x)),8)
$minimum
[1] -0.2324656
$estimate
[1] 0.4501831
$gradient
[1] 4.024558e-09
$code
[1] 1
$iterations
[1] 5
```

Here, the minimum value was found to be approximately −0.23, occurring at x = 0.45. A Newton-Raphson method (a technique from numerical analysis for approximating roots) is used, running five iterations in this case. The second argument specifies the initial guess, which we set to be 8.

Calculus:- R also has some calculus capabilities, including symbolic differentiation and numerical integration.

```
> D(expression(exp(x^2)),"x")          # derivative
    exp(x^2) * (2 * x)
```

```
> integrate(function(x) x^2,0,1)
0.3333333 with absolute error < 3.7e-15
```

Here, R reported $\frac{d}{dx}e^{x^2} = 2xe^{x^2}$ and $\int_0^1 x^2 dx = 0.33333333$

R packages for differential equations , for interfacing R with the Yacas symbolic math system (ryacas), and for other calculus operations. These packages, and thousands of others, are available from the Comprehensive R Archive Network (CRAN)

Functions Fir Statistical Distribution:- R has functions available for most of the famous statistical distributions.
Prefix the name as follows:

- With d for the density or probability mass function (pmf)
- With p for the cumulative distribution function (cdf)
- With q for quantiles
- With r for random number generation

The rest of the name indicates the distribution. Table 8-1 lists some common statistical distribution functions.

Distribution	Density/pmf	cdf	Quantiles	Random numbers
Normal	dnorm()	pnorm()	qnorm()	rnorm()
Chi square	dchisq()	pchisq()	qchisq()	rchisq()
Binomial	dbinom()	pbinom()	qbinom()	rbinom()

As an example, simulate 1,000 chi-square variates with 2 degrees of freedom and find their mean.

```
> mean(rchisq(1000,df=2))
[1] 1.938179
```

The r in rchisq specifies that we wish to generate random numbers— in this case, from the chi-square distribution. As seen in this example, the first argument in the r-series functions is the number of random variates to generate.

These functions also have arguments specific to the given distribution families. In our example, we use the df argument for the chi-square family, indicating the number of degrees of freedom.
Let's also compute the 95th percentile of the chi-square distribution with two degrees of freedom:

```
> qchisq(0.95,2)
[1] 5.991465
```

Here, we used q to indicate quantile—in this case, the 0.95 quantile, or the 95th percentile. The first argument in the d, p, and q series is actually a vector so that we can evaluate the density/pmf, cdf, or quantile function at multiple points. Let's find both the 50th and 95th percentiles of the chi-square distribution with 2 degrees of freedom.

```
qchisq(c(0.5,0.95),df=2)
[1] 1.386294 5.991465
```

Sorting:- Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order.

```
> x <- c(12,4,25,4)
> sort(x)
[1]  4  4 12 25
> x
[1] 12  4 25  4
```

The vector x did not change actually as printed in the very last line of the code. In order to sort the indexes as such, the order function is used in the following manner.

```
> order(x)
[1] 2 4 1 3
```

The console represents that there are two smallest values in vector x. The third smallest value being x[1], and so on. The same function order can also be used along with indexing for sorting data frames. This function can also be used to sort the characters as well as numeric values.
Another function which specifies the rank of every single element present in a vector is called rank( )

```
> x
[1] 12  4 25  4
> rank(x)
[1] 3.0 1.5 4.0 1.5
```

The above console demonstrates that the value 12 lies at rank 4th, which means that the 3rd smallest element in x is 12. Now, 4 number appears two times in the vector x. So, the rank 1.5 is allocated to both the numbers.

Example:- using order function on a dataframe.

```
> age <- c(12,4,34,14)
> names <- c("A","B","C","D")
> df <- data.frame(age,names)
> df
  age names
1  12     A
2   4     B
3  34     C
4  14     D
> df[order(df$age),]
  age names
2   4     B
1  12     A
4  14     D
3  34     C
```

Linear Algebra Operation on Vectors and Matrices:- The vector quantity can be multiplied to a scalar quantity as demonstrated:

```
> x <- c(13,5,12,5)
> y <-2*x
> y
[1] 26 10 24 10
```

To compute the inner product (or dot product) of two vectors, use crossprod(),

```
> a<-c(3,7,2)
> b<-c(2,5,8)
> crossprod(a,b)
     [,1]
[1,]   57
```

The function computed $3 \cdot 2+7 \cdot 5+ 2 \cdot 8 = 57$.

Note that the name crossprod() is a misnomer, as the function does not compute the vector cross product.

For matrix multiplications, the operator to use is %*% not *.

```
> c<-matrix(1:4,ncol=2)
> c
     [,1] [,2]
[1,] 1    3
[2,] 2    4
> d<-matrix(rep(1,4),ncol=2)
> d
     [,1] [,2]
[1,] 1    1
[2,] 1    1
> c%*%d
     [,1] [,2]
[1,] 4    4
[2,] 6    6
```

The function solve() will solve systems of linear equations and even find matrix inverses. For example, let's solve this system:

$$x_1 + x_2 = 2$$

$$-x_1 + x_2 = 4$$

$$\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

```
> a<-matrix(c(1,1,-1,1),ncol=2,byrow=T)
> b<-c(2,4)
> solve(a,b)
[1] -1  3
> solve(a)
     [,1] [,2]
[1,]  0.5 -0.5
[2,]  0.5  0.5
```

In the second call solve(), we are not giving second argument so it computers inverse of the matrix.
Few other linear algebra functions are,

- t(): Matrix transpose
- qr(): QR decomposition
- chol(): Cholesky decomposition
- det(): Determinant
- eigen(): Eigen values/eigen vectors
- diag(): Extracts the diagonal of a square matrix (useful for obtaining variances from a covariance matrix and for constructing a diagonal matrix).
- sweep(): Numerical analysis sweep operations

Note the versatile nature of diag(): If its argument is a matrix, it returns a vector, and vice versa. Also, if the argument is a scalar, the function returns the identity matrix of the specified size.

```
> x<-matrix(1:9,ncol=3)
> diag(x)
[1] 1 5 9
> a<-c(1,2,3)
> diag(a)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
> diag(3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

The sweep() function is capable of fairly complex operations. As a simple example, let's take a 3-by-3 matrix and add 1 to row 1, 4 to row 2, and 7 to row 3.

```
> a
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> sweep(a,1,c(3,4,5),"+")
     [,1] [,2] [,3]
[1,]    4    7   10
[2,]    6    9   12
[3,]    8   11   14
> sweep(a,2,c(3,4,5),"+")
     [,1] [,2] [,3]
[1,]    4    8   12
[2,]    5    9   13
[3,]    6   10   14
```

The first two arguments to sweep() are like those of apply(): the array and the margin, which is 1 for rows in this case. The fourth argument is a function to be applied, and the third is an argument to that function.

Vector Cross Product:- Let's consider the issue of vector cross products. The definition is very simple: The cross product of vectors (x_1, x_2, x_3) and (y_1, y_2, y_3) in three dimensional space is a new three-dimensional vector, as ($x_2y_3 - x_3y_2$, $-x_1y_3 + x_3y_1$, $x_1y_2 - x_2y_1$)
This can be expressed compactly as the expansion along the top row of the determinant, Here, the elements in the top row are merely placeholders.

$$\begin{pmatrix} - & - & - \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{pmatrix}$$

The point is that the cross product vector can be computed as a sum of subdeterminants. For instance, the first component in Equation 8.1, $x2y3 - x3y2$, is easily seen to be the determinant of the submatrix obtained by deleting the first row and first column.

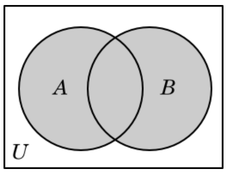
$$\begin{pmatrix} x_2 & x_3 \\ y_2 & y_3 \end{pmatrix}$$

Function to calculate cross product of vectors:

```
xprod <- function(x,y)
{
  m <- rbind(rep(NA,3),x,y)
  xp <- vector(length=3)
  for (i in 1:3)
    xp[i] <- -(-1)^i * det(m[2:3,-i])
  return(xp)
}
> xprod(c(12,4,2),c(2,1,1))
          [1]  2 -8  4
```

Set Operations:- R includes some handy set operations, including these:

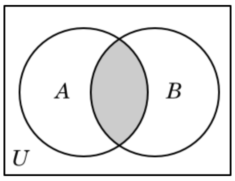
1) ***union(x,y):*** The union of two sets is defined as the set of all the elements that are members of set A, set B or both and is denoted by $A \cup B$ read as A union B.



$$A \cup B = \{x \mid x \in \text{A or x} \in \text{B}\}$$

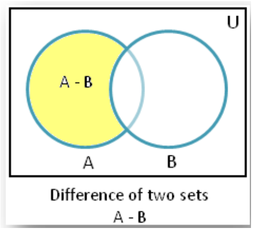
Eg: A = {1,2,3,4,5,a,b} B = {a,b,c,d,e}
$A \cup B$ = {1,2,3,4,5,a,b} $\cup$ {a,b,c,d,e}
= {1,2,3,4,5,a,b,c,d,e}

2) ***intersect(x,y):*** The intersection of any two sets A and B is the set containing of all the elements that belong to both A and B is denoted by $A \cap B$ read as A intersection B.



$$A \cap B = \{x \mid x \in \text{A and x} \in \text{B}\}$$

Eg: A = {1,2,3,4,5,a,b} B = {a,b,c,d,e}
$A \cap B$ = {1,2,3,4,5,a,b} $\cap$ {a,b,c,d,e}
= {a,b}



Difference of two sets
A - B

3) ***setdiff(x,y):*** The set difference of any two sets A and B is the set of elements that belongs to A but not B. It is denoted by A-B and read as 'A difference B'. A-B is also denoted by A|B or A~B. It is also called the relative complement of B in A.

Eg: A = {1,2,3,4,5,6} B = {3,5,7,9}

A-B = {1,2,4,6}

B-A = {7,9}

4) ***setequal(x,y):*** Test for equality between x and y. If both x and y are equal it returns TRUE otherwise returns FALSE

5) ***c %in% y:*** Membership, testing whether c is an element of the set y. It checks every corresponding element of 'c' with 'y',if both elements are equal it returns TRUE else return FALSE.

6) ***choose(n,r):*** Number of possible subsets of size k chosen from a set of size n

```
Eg:-  > choose(2,1)
      [1] 2
```

choose() function computes the combination n_{Cr}.

n: n elements

r: r subset elements

...

$n_{Cr} = n!/(r! * (n-r)!)$

```
> x <- c(1,5,3)
> y <- c(34,2,5)
> union(x,y)
[1] 1 5 3 34 2
>intersect(x,y)
[1] 5
> setdiff(x,y)
[1] 1 3
```

```
> setequal(x,y)
[1] FALSE
>choose(5,2)
[1] 10
>x %in% y
[1] FALSE TRUE FALSE
> 5 %in% y
[1] TRUE
```

? Code the symmetric difference between two sets— that is, all the elements belonging to exactly one of the two operand sets. Because the symmetric difference between sets x and y consists exactly of those elements in x but not y and vice versa.

```
function(a,b)
{
        sdfxy <- setdiff(x,y)
        sdfyx <- setdiff(y,x)
        return(union(sdfxy,sdfyx))
}
```

```
>x
 [1] 1 2 5
>y
  [1] 5 1 8 9
> symdiff(x,y)
  [1] 2 8 9
```

? Write a binary operand for determining whether one set u is a subset of another set v.
Hint: A bit of thought shows that this property is equivalent to the intersection of u and v being equal to u.

```
"%subsetof%" <- function(u,v)
{
   return(setequal(intersect(u,v),u))
}
> c(2,8) %subsetof% 1:10
[1] TRUE
> c(12,8) %subsetof% 1:10
[1] FALSE
```

combn() :-The function combn() generates combinations. Let's find the subsets of {1,2,3} of size 2.

```
> x <- combn(1:3,2)

> x
```

```
     [,1] [,2] [,3]
[1,] 1    1    2
[2,] 2    3    3
> class(x)
[1] "matrix"
```

The results are in the columns of the output. We see that the subsets of {1,2,3} of size 2 are (1,2), (1,3), and (2,3).

Input /output:- I/O plays a central role in most real-world applications of computers. Just consider an ATM cash machine, which uses multiple I/O operations for both input—reading your card and reading your typed-in cash request—and output—printing instructions on the screen, printing your receipt, and most important, controlling the machine to output your money!

R is not the tool you would choose for running an ATM, but it features a highly versatile array of I/O capabilities.

1.Accessing the keyboard and monitor:- R provides several functions for accesssing the keyboard and monitor. Few of them are scan(), readline(), print(), and cat() functions.

Using the scan( ) Function:-You can use scan() to read in a vector or a list, from a file or the keyboard. Suppose we have files named z1.txt, z2.txt.

```
z1.txt contains the following
123
4 5
6
z2.txt contains the follwing
abc
de f
g
        > scan("z1.txt")
                Read 4 items
                [1] 123   4   5   6
        > scan("z2.txt")
                Error in scan("z2.txt") : scan() expected 'a real', got 'abc'
        > scan("z2.txt",what="")
                Read 4 items
                [1] "abc" "de"  "f"   "g"
```

The scan() function has an optional argument named what, which specifies mode, defaulting to double mode. So, the non-numeric contents of the file z2 produced an error. But we then tried again, with what="". This assigns a character string to what, indicating that we want character mode.

By default, scan() assumes that the items of the vector are separated by whitespace, which includes blanks, carriage return/line feeds, and horizontal tabs. You can use the optional sep argument for other situations.

You can use scan() to read from the keyboard by specifying an empty string for the filename:

```
> scan("")
   1: 43 23 65 12
   5:
   Read 4 items
   [1] 43 23 65 12
> scan("",what="")
   1: "x" "y" "z" "srikanth" "Preethi" "omer"
   7:
   Read 6 items
   [1] "x" "y" "z" "srikanth" "Preethi" "omer"
```

readline() function:- If you want to read in a single line from the keyboard, readline() is very handy. readline() is called with its optional prompt.

```
> readline()
    Hai how are u
    [1] "Hai how are u"
> readline("Enter your Name")
    Enter your Name VIT
    [1] "VIT"
```

Printing to the Screen:- At the top level of interactive mode, you can print the value of a variable or expression by simply typing the variable name or expression. This won't work if you need to print from within the body of a function. In that case, you can use the print() function,

```
> x <- 1:3
> print(x^2)
[1] 1 4 9
```

print() is a generic function, so the actual function called will depend on the class of the object that is printed. If, for example, the argument is of class "table", then the print.table() function will be called.

It's a little better to use cat() instead of print(), as the latter can print only one expression and its output is numbered, which may be a nuisance. Compare the results of the functions:

```
> print("abc")
[1] "abc"
> cat("abc\ndef")
abc
def
```

Note that we needed to supply our own end-of-line character, "\n", in the call to cat(). Without it, our next call would continue to write to the same line. The arguments to cat() will be printed out with intervening spaces:

```
> x
[1] 1 2 3
> cat(x,"abc","de\n")
1 2 3 abc de
```

If you don't want the spaces, set sep to the empty string "", as follows:

```
> cat(x,"abc","de\n",sep="")
123abcde
```

Any string can be used for sep. Here, we use the newline character:

```
>cat(x,"abc","de\n",sep="\n")
    1
    2
    3
    abc
    de
```

Set sep can be used with a vector of strings, like this:

```
> x <- c(5,12,13,8,88)
> cat(x,sep=c(".",".",".","\n","\n"))
5.12.13.8
88
```

2. Reading and Writing Files:- It includes reading data frames or matrices from files, working with text files, accessing files on remote machines, and getting file and directory information.

Reading a Data Frame or Matrix from a File:- read.table() is used to read a data frame from the file.

```
> read.table("z1.txt",header=TRUE)
     name     nature
1  Hemant   Obidient
2 Sowjanya  Hardworking
```

```
3   Girija      Friendly
4 Preethi             Calm
```

scan() would not work here, as our data-frame has mixture of character and numeric data.We can read a matrix using scan as

```
Mat<-matrix(scan("abc.txt"), nrow=2, ncol=2, byrow=T)
```

We can do this generally by using read.table() as

```
read.matrix<-function(filename){
        as.matrix(read.table(filename))
}
```

Reading a Text-File: readLines() is used to read in a text file, either one line at a time or in a single operation. For example, suppose we have a file z1 with the following contents:

```
John 25
Mary 28
Jim 19
```

We can read the file all at once, like this:

```
>z1 <- readLines("z1")
>z1
[1] "John 25" "Mary 28" "Jim 19"
```

Since each line is treated as a string, the return value here is a vector of strings—that is, a vector of character mode.

There is one vector element for each line read, thus three elements here. Alternatively, we can read it in one line at a time. For this, we first need to create a connection, as described next.

Introduction to Connections: Connection is R's term for a fundamental mechanism used in various kinds of I/O operations. The connection is created by calling file(), url(), or one of several other R functions. ?connection

```
> c <- file("z1","r")
> readLines(c,n=1)
[1] "John 25"
> readLines(c,n=1)
[1] "Mary 28"
> readLines(c,n=1)
[1] "Jim 19"
> readLines(c,n=1)
character(0)
```

We opened the connection, assigned the result to c, and then read the file one line at a time, as specified by the argument n=1. When R encountered the end of file (EOF), it returned an empty result.We needed to set up a connection so that R could keep track of our position in the file as we read through it.

```
c <- file("z","r")
while(TRUE)
{
        rl <- readLines(c,n=1)
        if (length(rl) == 0)
         {
                print("reached the end")
                break
        } else print(rl)
}
```

```
OUTPUT:
[1] "John 25"
[1] "Mary 28"
[1] "Jim 19"
[1] "reached the end"
```

Accessing files on remote machines via urls: Certain I/O functions, such as read.table() and scan(), accept web URLs as arguments.

```
uci <- "http://archive.ics.uci.edu/ml/machine-learning-databases/echocardiogram/   echocardiogram. data"
> ecc <- read.csv(uci)
```

Writing to a file: The function write.table() works very much like read.table(), except that it writes a data frame instead of reading one.

```
> kids <- c("Jack","Jill")
> ages <- c(12,10)
> d <- data.frame(kids,ages,stringsAsFactors=FALSE)
>d kids ages
1 Jack 12
2 Jill 10
> write.table(d,"kds.txt")
```

In the case of writing a matrix to a file, just state that you do not want row or column names, as follows:

```
write.table(xc, "xcnew", row.names=FALSE, col.names=FALSE)
```

The function cat() can also be used to write to a file, one part at a time.

```
> cat("abc\n",file="u")
> cat("de\n",file="u",append=TRUE)
```

The first call to cat() creates the file u, consisting of one line with contents "abc". The second call appends a second line. The file is automatically saved after each operation.
writeLines() function can also be used, the counterpart of readLines(). If you use a connection, you must specify "w" to indicate you are writing to the file, not reading from it:

```
> c <- file("www","w")
> writeLines(c("abc","de","f"),c)
> close(c)
```

The file www will be created with these contents:

```
abc
de
f
```

Linear Models, Simple Linear Regression, -Multiple Regression Generalized Linear *Models,Logistic Regression, - Poisson Regression- other Generalized Linear Models-Survival Analysis,Nonlinear Models - Splines, Decision, Random Forests.*

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:- 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 −

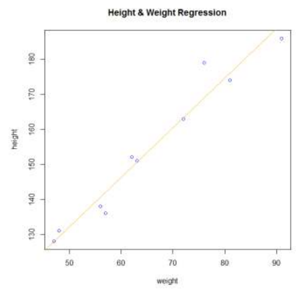
- y is the response variable.
- x is the predictor variable.
- a and b are constants which are called the coefficients.

lm() Function:-This function creates the relationship model between the predictor and the response variable.
Syntax: ***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.

Example:-

```
> height <- c(151, 174, 138, 186, 128, 136, 179, 163, 152, 131)
> weight <- c(63, 81, 56, 91, 47, 57, 76, 72, 62, 48)
> relation <- lm(weight~height)
> print(relation)

Call:
lm(formula = weight ~ height)

Coefficients:
(Intercept)       height
   -38.4551       0.6746
> plot(weight,height,col = "blue",main = "Height & Weight
Regression")
> abline(lm(height~weight),col="orange")
```



Advantages / Limitations of Linear Regression Model :

- Linear regression implements a statistical model that, when relationships between the independent variables and the dependent variable are almost linear, shows optimal results.
- Linear regression is often inappropriately used to model non-linear relationships.
- Linear regression is limited to predicting numeric output.
- A lack of explanation about what has been learned can be a problem.

Regression equation of x on y

$$x-\bar{x}=r.\frac{\sigma_x}{\sigma_y}(y-\overline{y})$$

Regression equation of y on x

$$y-\bar{y}=r.\frac{\sigma_y}{\sigma_x}(x-\overline{x})$$

where

$$r = \frac{\frac{1}{n}\sum xy - \bar{x} - \bar{y}}{\sqrt{\frac{1}{n}\sum (x-\bar{x})^2}\sqrt{\frac{1}{n}\sum (y-\bar{y})^2}}$$

Multiple Regression :- **Multiple regression** is an extension of simple **linear regression**. It is used when we want to predict the value of a variable based on the value of two or more other variables. The variable we want to predict is called the dependent variable

The general mathematical equation for multiple regression is – $x_1 = a_0 + a_1 x_2 + a_2 x_3$

Following is the description of the parameters used −

- x_1 is the response variable.
- a_0, a_1, a_2...bn are the coefficients.
- x_1, x_2, ...xn are the predictor variables.

The Normal equations for estimating a_0,a_1 and a_2 .

$$\sum x_1 = na_0 + a_1\sum x_2 + a_2\sum x_3$$
$$\sum x_1x_2 = a_0\sum x_2 + a_1\sum x_2^{\ 2} + a_2\sum x_2x_3$$
$$\sum x_1x_3 = a_0\sum x_3 + a_1\sum x_2x_3 + a_2\sum x_3^{\ 2}$$

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 :- *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

```
> lm(mpg~disp+hp+wt,data=mtcars)

Call:
lm(formula = mpg ~ disp + hp + wt, data = mtcars)

Coefficients:
(Intercept)         disp           hp           wt
  37.105505    -0.000937    -0.031157    -3.800891
```

Create Equation for Regression Model

Based on the above intercept and coefficient values, we create the mathematical equation.

$Y = a+disp.x_1+hp.x_2+wt.x_3$

or

$Y = 37.15+(-0.000937)*x_1+(-0.0311)*x_2+(-3.8008)*x_3$

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 −

$y = 1/(1+e\text{^}-(a+b_1x_1+b_2x_2+b_3x_3+...))$

- **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 :- *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.

For example, in the built-in data set mtcars, the data column am represents the transmission type of the automobile model (0 = automatic, 1 = manual). With the logistic regression equation, we can model the probability of a manual transmission in a vehicle based on its engine horsepower and weight data.

```
> am.glm = glm(formula=am ~ hp + wt, data=mtcars, family=binomial)
> am.glm

Call:  glm(formula = am ~ hp + wt, family = binomial, data = mtcars)

Coefficients:
(Intercept)           hp           wt
   18.86630      0.03626     -8.08348

Degrees of Freedom: 31 Total (i.e. Null);  29 Residual
Null Deviance:      43.23
Residual Deviance: 10.06      AIC: 16.06
```

Poisson Regression:- Poisson Regression involves regression models in which the response variable is in the form of counts and not fractional numbers. For example, the count of number of births or number of wins in a football match series. Also the values of the response variables follow a Poisson distribution.
The general mathematical equation for Poisson regression is −

$\log(y) = a + b_1x_1 + b_2x_2 + b_nx_n.....$

Following is the description of the parameters used −

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

The function used to create the Poisson regression model is the glm()function.
We have the in-built data set "warpbreaks" which describes the effect of wool type (A or B) and tension (low, medium or high) on the number of warp breaks per loom. Let's consider "breaks" as the response variable which is a count of number of breaks. The wool "type" and "tension" are taken as predictor variables.

```
> output <-glm(formula = breaks ~ wool+tension,data = warpbreaks,
+ family = poisson)
> print(summary(output))

Call:
glm(formula = breaks ~ wool + tension, family = poisson, data = warpbreaks)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-3.6871  -1.6503  -0.4269   1.1902   4.2616

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.69196    0.04541  81.302  < 2e-16 ***
woolB       -0.20599    0.05157  -3.994 6.49e-05 ***
tensionM    -0.32132    0.06027  -5.332 9.73e-08 ***
tensionH    -0.51849    0.06396  -8.107 5.21e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 297.37  on 53  degrees of freedom
Residual deviance: 210.39  on 50  degrees of freedom
AIC: 493.06

Number of Fisher Scoring iterations: 4
```

CURVE FITTING:- Curve fitting is the process of constructing a **curve**, or mathematical function, that has the best **fit** to a series of data points, possibly subject to constraints.

Type of curve	Equation	Normal equations
Fitting of a straight line	y = a + bx	$\sum y = na + b\sum x$ $\sum xy = a\sum x + b\sum x^2$
Fitting of a second degree polynomial	$y = a + bx + cx^2$	$\sum y = na + b\sum x + c\sum x^2$ $\sum xy = a\sum x + b\sum x^2 + c\sum x^3$ $\sum x^2 y = a\sum x^2 + b\sum x^3 + c\sum x^4$
Power curve	$y = a.b^x$	Applylog on both sides $\log y = \log a + x\log b$ $Y = A + Bx$ $\sum Y = nA + B\sum x$ $\sum xY = A\sum x + B\sum x^2$ where Y = log y, A = log a and B = log b
Exponential curve	$y = a.\ e^{bx}$	Applylog on both sides $\log y = \log a + bx$ $Y = A + bx$ $\sum Y = nA + b\sum x$ $\sum xY = A\sum x + b\sum x^2$ where Y = log y and A = log a
Exponential curve	$y = a.\ x^b$	Applylog on both sides $\log y = \log a + b\log x$ $Y = A + bX$ $\sum Y = nA + b\sum X$ $\sum XY = A\sum X + b\sum X^2$ where Y = log y , X=log x and A = log a

Problem:- Fit a straight line to the following data

x	*0*	*1*	*2*	*3*	*4*
y	*1*	*1.8*	*3.3*	*4.5*	*6.3*

Solution:-

Straight line is y = a+bx
The two normal equations are

$$\sum y = na + b\sum x$$
$$\sum xy = a\sum x + b\sum x^2$$

x	x^2	y	xy
0	0	1	0
1	1	1.8	1.8
2	4	3.3	13.2
3	9	4.5	40.5

4	16	6.3	100.8
$\sum x = 10$	$\sum x^2 = 30$	$\sum y = 16.9$	$\sum xy = 156.3$

Substituting the values, we get

5a+10b = 16.9 ....(1)

10a+30b = 156.3 ....(2)

Solving (1) and (2), we get

Multiply eq (1) with 2

10a+20b = 33.2 ....(3)

Subtract (3) and (2)

10a + 20b = 33.2

10a + 30b = 156.3

0 -10b =-123.1

Therefore b=12.3 now substitute in (1) and a = -21.24.

Thus the equation of the straight line is y = a + bx

y = -21.24+12.3x

Problem:- **Fit a parabola to the following data**

x	1	2	3	4	5
y	10	12	8	10	14

Solution:-

Polynomial equation line is $y = a + bx + cx^2$

The three normal equations are

$$\sum y = na + b\sum x + c\sum x^2$$
$$\sum xy = a\sum x + b\sum x^2 + c\sum x^3$$
$$\sum x^2 y = a\sum x^2 + b\sum x^3 + c\sum x^4$$

x	x^2	x^3	x^4	y	xy	x^2y
1	1	1	1	10	10	10
2	4	8	16	12	24	64
3	9	27	81	8	24	72
4	16	64	256	10	40	160
5	25	125	625	14	70	350
$\sum x = 15$	$\sum x^2 = 55$	$\sum x^3 = 225$	$\sum x^4 = 979$	$\sum y = 54$	$\sum xy = 168$	$\sum x^2 y = 656$

Substituting the values, we get

5a+15b+55c = 54 ....(1)

15a+55b+225c = 168 ....(2)

55a+225b+979c = 656 ....(3)

Solving (1) and (2), we get

Multiply eq (1) with 3 and subtract with (2)

15a+45b+165c = 162

(-)15a+55b+225c = 168

0 -10b+60c = -6 ....(4)

Solving (1) and (3), we get

Multiply eq (1) with 11 and subtract with (3)

55a+165b+605c = 594

(-)55a+225b+979c = 656

0 -60b-370c = -62

60 b+ 370 c = 62.......(5)

Solve (4) and (5)

Multiply eq (4) with 6 and add with (5)

$$\begin{array}{r} -60b + 360c = -36 \\ \underline{60b + 370c = 62} \\ 70c = 26 \\ c = 0.37 \end{array}$$

substitute in equation (4)

$-10b + 60(0.37) = -6$

$b = 2.82$

Substitute c and b values in equation (1)

$5a+15b+55c = 54$

$5a+15(2.82)+55(0.37) = 54$

$a = -1.73$

Thus the equation of the polynomial is $y = a + bx + cx^2$

$y = -1.73+2.82x + 0.37x^2$

Problem:- **Fit a curve of the type $y=ae^{bx}$ to the following data**

x	0	1	2	3
y	1.05	2.1	3.85	8.3

Solution:-Exponential curve equation line is $y = a . e^{bx}$

The two normal equations are

Apply logarithm on both sides

$$\log y = \log a + bx$$

$$Y = A + bx$$

$$\sum Y = nA + b\sum x$$

$$\sum xY = A\sum x + b\sum x^2$$

x	y	Y = log y	xY	x^2
0	1.05	0.021	0	0
1	2.1	0.324	0.32	1
2	3.85	0.585	1.17	4
3	8.3	0.919	2.75	9
$\sum x = 6$	$\sum y = 15.3$	$\sum Y = 1.849$	$\sum xY = 4.24$	$\sum x^2 = 14$

The equations are

$4A + 6b = 1.84$ …….(1)

$6A + 14b = 4.24$ …….(2)

Solve (1) and (2) equations

Multiply (1) with 3 and (2) with 2 then subtract them

$$\begin{array}{r} 12A+18b = 5.52 \\ (-)\underline{12A+28b = 8.48} \\ -10b = -2.96 \\ b = 0.296 \end{array}$$

Substitute b value in (1)

$4A + 6(0.296) = 1.84$

$A = 0.016$

$a = \text{antilog}(A)$

$= \text{antilog}(0.016) = 1.061$

Therefore the exponential curve is $y = (1.061).e^{0.296x}$

Survival analysis: Survival analysis is generally defined as a set of methods for analyzing data where the outcome variable is the time until the occurrence of an event of interest. The event can be death, occurrence of a disease, marriage, divorce, etc.
In survival analysis, there is a special structure for right-censored survival data. To use this, one first must load the "survival" package, which is included in the main R distribution,

library(survival)

The basic syntax for creating survival analysis in R is −

Surv(time,event)
survfit(formula)

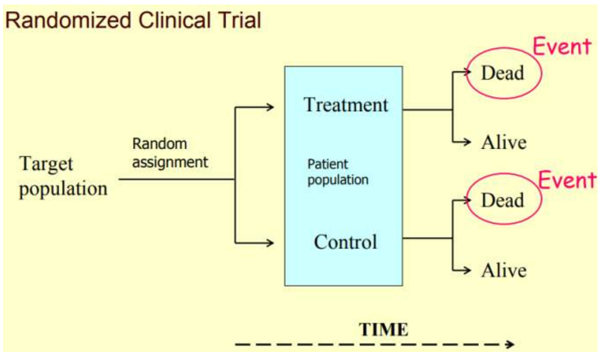
Following is the description of the parameters used −

- time is the follow up time until the event occurs.
- event indicates the status of occurrence of the expected event.
- formula is the relationship between the predictor variables.

Next, define the survival times "tt" and the censoring indicator "status", where "status = 1" indicates that the time is an observed event, and "status = 0" indicates that it is censored. Then the "Surv" function binds them into a single object. In the following example, time 6 is right censored, while the others are observed event times,

```
> tt <- c(2, 5, 6, 7, 8)
> status <- c( 1, 1, 0, 1, 1)
> Surv(tt, status) # Create a survival data structure
 [1] 2 5 6+ 7 8
```

Example:-



Nonlinear Models

Decision trees:- Decision tree is a graph to represent choices and their results in form of a tree. The nodes in the graph represent an event or choice and the edges of the graph represent the decision rules or conditions. It is mostly used in Machine Learning and Data Mining applications using R.
Examples:

- Predicting an email as spam or not spam,
- Predicting of a tumor is cancerous
- Predicting a loan as a good or bad credit risk based on the factors in each of these.

The package "party" has the function **ctree()** which is used to create and analyze decison tree.

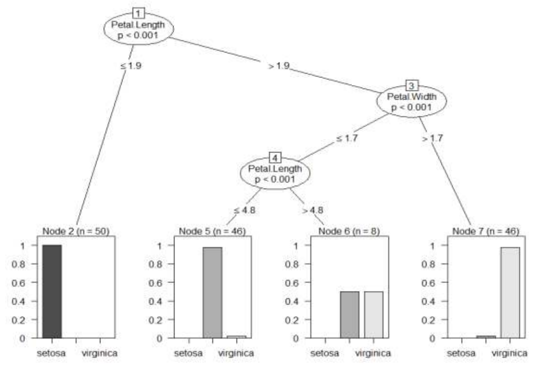
Syntax : *ctree(formula, data)*

Following is the description of the parameters used −

- **formula** is a formula describing the predictor and response variables.
- **data** is the name of the data set used.

Example:

```
library(party)
model2<-ctree(Species ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width, data=mydata)
plot(model2)
```



Advantages of Decision Trees

- Simple to understand and interpret.
- Requires little data preparation.
- Works with both numerical and categorical data.
- Possible to validate a model using statistical tests. Gives you confidence it will work on new data sets.
- Robust.
- Scales to big data

Limitations of Decision Trees

- Learning globally optimal tree is NP-hard.
- Easy to overfit the tree
- Complex.

Random Forests:- In the random forest approach, a large number of decision trees are created. Every observation is fed into every decision tree. The most common outcome for each observation is used as the final output.
The package "randomForest" has the function **randomForest()** which is used to create and analyze random forests.

Syntax :- ***randomForest(formula, data)***

Following is the description of the parameters used −

- **formula** is a formula describing the predictor and response variables.
- **data** is the name of the data set used.

Advantages

- It is one of the most accurate learning algorithms available. For many data sets, it produces a highly accurate classifier.
- It runs efficiently on large databases.
- It can handle thousands of input variables without variable deletion.
- It gives estimates of what variables are important in the classification.
- It generates an internal unbiased estimate of the generalization error as the forest building progresses.
- It has an effective method for estimating missing data and maintains accuracy when a large proportion of the data are missing.

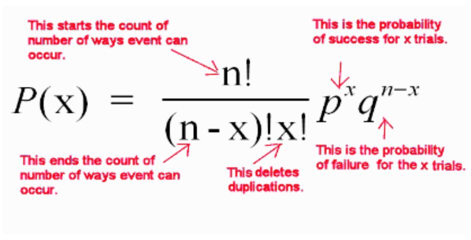
Disadvantages

- Random forests have been observed to overfit for some datasets with noisy classification/regression tasks.
- For data including categorical variables with different number of levels, random forests are biased in favor of those attributes with more levels. Therefore, the variable importance scores from random forest are not reliable for this type of data.

Splines: A linear spline is a continuous function formed by connecting linear segments. The points where the segments connect are called the knots of the spline.

UNIT-V: Probability Distributions, Normal Distribution- Binomial Distribution- Poisson Distributions, Other Distribution, Basic Statistics, Correlation and Covariance, T-Tests, ANOVA.

BINOMIAL DISTRIBUTION:- The binomial distribution is a discrete probability distribution. It describes the outcome of n independent trials in an experiment. Each trial is assumed to have only two outcomes, either success or failure. If the probability of a successful trial is p, then the probability of having x successful outcomes in an experiment of n independent trials is as follows.



$$P(x) = \frac{n!}{(n-x)!\,x!}\, p^x q^{n-x}$$

This starts the count of number of ways event can occur.
This is the probability of success for x trials.
This ends the count of number of ways event can occur.
This deletes duplications.
This is the probability of failure for the x trials.

$\text{Mean} = \mu = E(x) = np$

$\text{Variance} = \sigma^2 = np(1-p)$

$\text{Standard Deviation} = \sigma = \sqrt{np(1-p)}$

where

n – number of trials

p – probability of success

1-p – probability of other outcome (failure)

R has four in-built functions to generate binomial distribution. They are described below.

- ***dbinom(x, size, prob)*** :- This function gives the probability density distribution at each point.
- ***pbinom(x, size, prob)*** :- This function gives the cumulative probability of an event. It is a single value representing the probability.
- ***qbinom(p, size, prob)*** :- This function takes the probability value and gives a number whose cumulative value matches the probability value.
- ***rbinom(n, size, prob)*** :- This function generates required number of random values of given probability from a given sample.

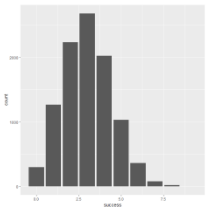
Following is the description of the parameters used −

- ✓ x is a vector of numbers.
- ✓ p is a vector of probabilities.
- ✓ n is number of observations.
- ✓ size is the number of trials.
- ✓ prob is the probability of success of each trial.

Examples:

- ➢ *rbinom(n=1,size=10,prob=0.4)* - It generates 1 random number from the binomial distribution basesd on number of successes of 10 independent trails.
- ➢ *rbinom(n=5,size=10,prob=0.4)* - It generates 5 random number from the binomial distribution basesd on number of successes of 10 independent trails with probability 0.4.
- ➢ *rbinom(n=5,size=1,prob=0.4)* – Setting size to 1 turns the numbers into a bernoulli random variable, which can take only value 1 (success) or 0 (failure).
- ➢ To visualize the binomial distribution we randomly generate 10,000 experiments, each with 10 trails and 0.3 probability.

```
b <- data.frame(success=rbinom(n=10000,size=10,prob=0.3))
ggplot(b,aes(x=success))+geom_bar()
```



Problem: ***Suppose a die is tossed 5 times. What is the probability of getting exactly 2 fours?***

Solution: This is a binomial experiment in which the number of trials is equal to 5, the number of successes is equal to 2, and the probability of success on a single trial is 1/6 or about 0.167. Therefore, the binomial probability is:

$b(2; 5, 0.167) = {}^5C_2 * (0.167)^2 * (0.833)^3$

$b(2; 5, 0.167) = 0.161$

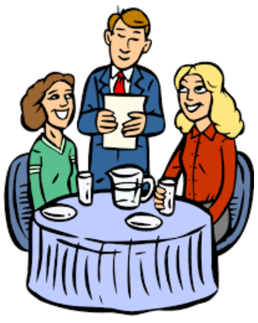
R Code:

```
> dbinom(2, size=5, prob=0.167)
[1] 0.1612
```

Problem: *In a restaurant seventy percent of people order for Chinese food and thirty percent for Italian food. A group of three persons enter the restaurant. Find the probability of at least two of them ordering for Italian food.*

Solution:-

The probability of ordering Chinese food is 0.7 and the probability of ordering Italian food is 0.3. Now, if at least two of them are ordering Italian food then it implies that either two or three will order Italian food.



Probability for two ordering Italian food,

$P(X=2) = {}^3C_2(0.3)^2(0.7)^1$

$= 3\times0.09\times0.7$

$= 0.189$

Probability for all three ordering Italian food,

$P(X=3) = {}^3C_3(0.3)^3(0.7)^0$

$= 1\times0.027\times1$

$= 0.027$

Hence, the probability for at least two persons ordering Italian food is,

$P(X \geq 2) = P(X=2)+P(X=3) \quad = 0.189+0.027=0.216$

R code:-

```
> dbinom(2,size=3,prob=0.3)+
+ dbinom(3,size=3,prob=0.3)
[1] 0.216
```

Cumulative Binomial Probability:- A cumulative binomial probability refers to the probability that the binomial random variable falls within a specified range (e.g., is greater than or equal to a stated lower limit and less than or equal to a stated upper limit).

Problem:*What is the probability of obtaining 45 or fewer heads in 100 tosses of a coin?*

Solution: To solve this problem, we compute 46 individual probabilities, using the binomial formula. The sum of all these probabilities is the answer we seek.

Thus,

$b(x \leq 45; 100, 0.5) = b(x = 0; 100, 0.5) + b(x = 1; 100, 0.5) + \ldots + b(x = 45; 100, 0.5)$

$= 0.184$

R code:-

```
> pbinom(45,size=100,prob=0.5)
[1] 0.1841008
```

Problem: *Suppose there are twelve multiple choice questions in an English class quiz. Each question has five possible answers, and only one of them is correct. Find the probability of having four or less correct answers if a student attempts to answer every question at random.*

Solution:

Since only one out of five possible answers is correct, the probability of answering a question correctly by random is 1/5=0.2.

- To find the probability of having exactly 4 correct answers by random attempts as follows.

```
> dbinom(4, size=12, prob=0.2)
[1] 0.1329
```

- To find the probability of having four or less correct answers by random attempts, we apply the function dbinom with x = 0,…,4.

```
> dbinom(0, size=12, prob=0.2) + dbinom(1, size=12, prob=0.2) +
+ dbinom(2, size=12, prob=0.2) +  dbinom(3, size=12, prob=0.2) +
+ dbinom(4, size=12, prob=0.2)
[1] 0.9274
```

- Alternatively, we can use the cumulative probability function for binomial distribution pbinom.

```
> pbinom(4, size=12, prob=0.2)
[1] 0.92744
```

Answer:-The probability of four or less questions answered correctly by random in a twelve question multiple choice quiz is 92.7%.

Problem: *Fit an appropriate binomial distribution and calculate the theoretical distribution*

x :	*0*	*1*	*2*	*3*	*4*	*5*
f :	*2*	*14*	*20*	*34*	*22*	*8*

Solution:

Here n = 5 , N = 100

$$\text{Mean} = \frac{\sum x_i f_i}{\sum f_i} = 2.84$$

np = 2.84

p = 2.84/5 = 0.568

q = 0.432

$p(r) = 5C_r\,(0.568)^r\,(0.432)^{5-r}$, r = 0,1,2,3,4,5

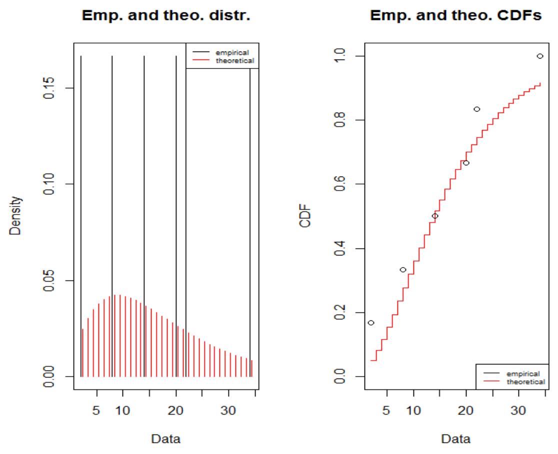
Theoretical distributions are

Calculation of Expected Frequency as follows

r	p(r)	N* p(r)
0	0.0147	100 * 0.0147 =1.47 = 1
1	0.097	100 * 0.097 =9.7 =10
2	0.258	100 * 0.258 =25.8 =26
3	0.342	100 * 0.342 =34.2 =34
4	0.226	100 * 0.226 =22.6 =23
5	0.060	100 * 0.060 = 6 =6
		Total = 100

R code:-

```
> x <- 0:5
> f <- c(2,14,20,34,22,8)
> df <-data.frame(x,f)
> fitbin <- fitdist(df$f,"nbinom")
> summary(fitbin)
Fitting of the distribution ' nbinom ' by maximum
likelihood
Parameters :
        estimate     Std. Error
size  2.192416     1.441296
mu   16.664004   4.886713
Loglikelihood:  -22.387   AIC:  48.774   BIC:  48.35752
Correlation matrix:
              size                mu
size 1.0000000000    0.0003165092
mu   0.0003165092    1.0000000000

> plot(fitbin)
```



Poisson Distribution :- The **Poisson distribution** is the probability distribution of independent event occurrences in an interval. If λ is the mean occurrence per interval, then the probability of having x occurrences within a given interval is:

$$p(x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

$p(x)$ = Probability of x given λ

λ = Expected (mean) number 'successes'

e = 2.71828 (base of natural logs)

x = Number of 'successes' in per unit

Mean $\mu = E(x) = \lambda$ **Standard Deviation** $\sigma = \sqrt{\lambda}$

Examples:

1. The number of defective electric bulbs manufactured by a reputed company.
2. The number of telephone calls per minute at a switch board
3. The number of cars passing a certain point in one minute.
4. The number of printing mistakes per page in a large text.

R has four in-built functions to generate binomial distribution. They are described below.

- ***dpois(x, lambda, log = FALSE)*** :- This function gives the probability density distribution at each point.
- ***ppois(q, lambda, lower.tail = TRUE, log.p = FALSE)*** :- This function gives the cumulative probability of an event. It is a single value representing the probability.
- ***qpois(p, lambda, lower.tail = TRUE, log.p = FALSE)***:- This function takes the probability value and gives a number whose cumulative value matches the probability value.
- ***rpois(n, lamda)*** :- This function generates required number of random values of given probability from a given sample.

Following is the description of the parameters used −

- ✓ x is a vector of numbers.
- ✓ p is a vector of probabilities.
- ✓ n is number of observations.
- ✓ size is the number of trials.
- ✓ prob is the probability of success of each trial.

Problem:- *If there are twelve cars crossing a bridge per minute on average, find the probability of having seventeen or more cars crossing the bridge in a particular minute.*

Solution:-

The probability of having *sixteen or less* cars crossing the bridge in a particular minute is given by the function ppois.

```
> ppois(16, lambda=12)   # lower tail
[1] 0.89871
```

Hence the probability of having seventeen or more cars crossing the bridge in a minute is in the *upper tail* of the probability density function.

```
> ppois(16, lambda=12, lower=FALSE)   # upper tail
[1] 0.10129
```

Answer:- If there are twelve cars crossing a bridge per minute on average, the probability of having seventeen or more cars crossing the bridge in a particular minute is 10.1%.

Problem:- *The average number of homes sold by the Acme Realty company is 2 homes per day. What is the probability that exactly 3 homes will be sold tomorrow?*

Solution: This is a Poisson experiment in which we know the following:

- μ = 2; since 2 homes are sold per day, on average.
- x = 3; since we want to find the likelihood that 3 homes will be sold tomorrow.
- e = 2.71828; since e is a constant equal to approximately 2.71828.

We plug these values into the Poisson formula as follows:

$P(x; \mu) = (e^{-\mu}) (\mu^x) / x!$
$P(3; 2) = (2.71828^{-2}) (2^3) / 3!$
$= (0.13534) (8) / 6$
$= 0.180$

R Code:-

```
> dpois(3,lambda = 2)
[1] 0.180447
```

Cumulative Poisson Probability:- A **cumulative Poisson probability** refers to the probability that the Poisson random variable is greater than some specified lower limit and less than some specified upper limit.

Problem:-*Suppose the average number of lions seen on a 1-day safari is 5. What is the probability that tourists will see fewer than four lions on the next 1-day safari?*

Solution: This is a Poisson experiment in which we know the following:

- $\mu = 5$; since 5 lions are seen per safari, on average.
- x = 0, 1, 2, or 3; since we want to find the likelihood that tourists will see fewer than 4 lions; that is, we want the probability that they will see 0, 1, 2, or 3 lions.
- e = 2.71828; since e is a constant equal to approximately 2.71828.

To solve this problem, we need to find the probability that tourists will see 0, 1, 2, or 3 lions. Thus, we need to calculate the sum of four probabilities: P(0; 5) + P(1; 5) + P(2; 5) + P(3; 5). To compute this sum, we use the Poisson formula:

$P(x \leq 3, 5) = P(0; 5) + P(1; 5) + P(2; 5) + P(3; 5)$
$P(x \leq 3, 5) = [(e^{-5})(5^0) / 0!] + [(e^{-5})(5^1) / 1!] + [(e^{-5})(5^2) / 2!] + [(e^{-5})(5^3) / 3!]$
$P(x \leq 3, 5) = [(0.006738)(1) / 1] + [(0.006738)(5) / 1] + [(0.006738)(25) / 2] +[(0.006738)(125) / 6]$
$P(x \leq 3, 5) = [0.0067] + [0.03369] + [0.084224] + [0.140375]$
$P(x \leq 3, 5) = 0.2650$

Thus, the probability of seeing at no more than 3 lions is 0.2650.

R Code:-

```
> ppois(3,lambda = 5)
[1] 0.2650259
```

Normal Distribution:- A continuous random variable X follows a normal distribution with mean μ and variance σ^2 is a statistic distribution with probability density function

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{(\mu-x)^2}{2\sigma^2}}$$, on the domain $x \in (-\infty, \infty)$.

Standard Normal Distribution

It is the distribution that occurs when a normal random variable has a mean of zero and a standard deviation of one.

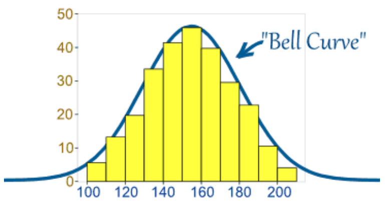
The normal random variable of a standard normal distribution is called a standard score or a z score. Every normal random variable X can be transformed into a z score via the following equation:

$$Z = (X - \mu) / \sigma$$

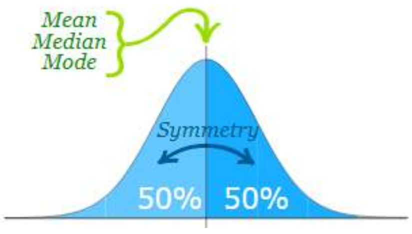
where X is a normal random variable, μ is the mean, and σ is the standard deviation.

yielding

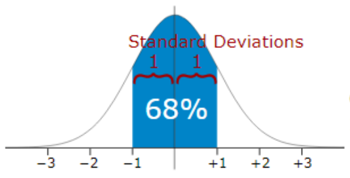
$$P(x)\,dx = \frac{1}{\sqrt{2\pi}} e^{-z^2/2}\,dz.$$



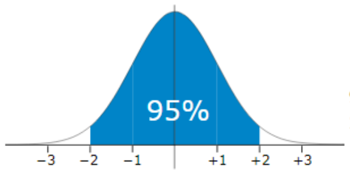
Standard Normal Curve:- One way of figuring out how data are distributed is to plot them in a graph. If the data is evenly distributed, you may come up with a bell curve. A bell curve has a small percentage of the points on both tails and the bigger percentage on the inner part of the curve. The shape of the standard normal distribution looks like this:



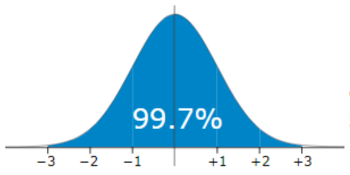
- mean = median = mode
- symmetry about the center
- 50% of values less than the mean and 50% greater than the mean



68% of values are within
1 standard deviation of the mean



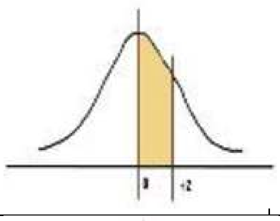
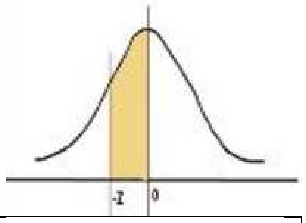
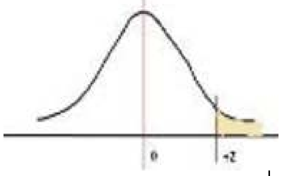
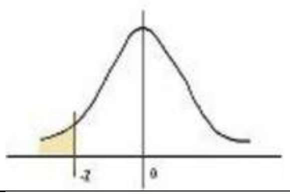
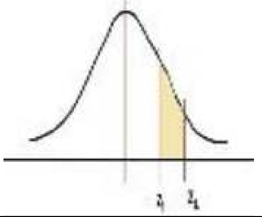
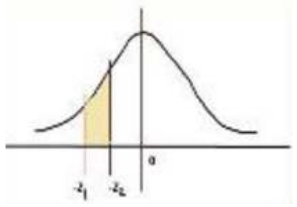
95% of values are within
2 standard deviations of the mean

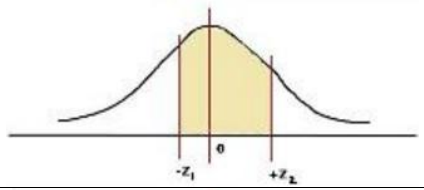
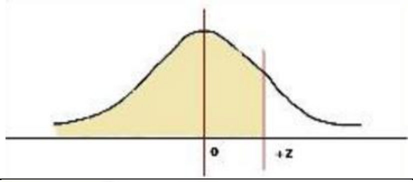
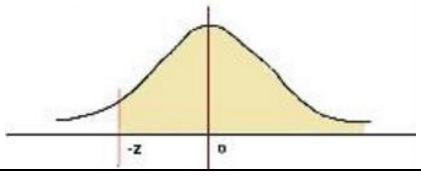


99.7% of values are within
3 standard deviations of the mean

R functions:

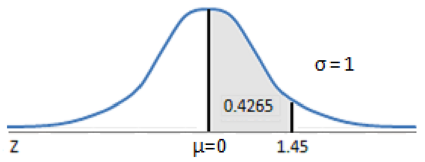
- ***dnorm(x, mean = 0, sd = 1, log = FALSE) :-*** This function gives the probability density distribution at each point.
- ***pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE):-*** This function gives the cumulative probability of an event. It is a single value representing the probability.
- ***qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE):-*** This function takes the probability value and gives a number whose cumulative value matches the probability value.
- ***rnorm(n, mean = 0, sd = 1) :-*** This function generates required number of random values of given probability from a given sample.

Procedure to find probability using positive Z-score table			
Case 1: Area between 0 and any z score	Area(z)		
Case 2: Area in any tail	0.5 – Area(z)		
Case 3: Area between two z-scores on the same side of the mean	| Area(z2)-Area(z1) |		

Case 4: Area between two z-scores on the opposite side of the mean	Area(z1)+Area(z2)	
Case 5: Area to the left of a positive Z score	0.5+ Area(z)	
Case 6: Area to the right of a negative Z score	0.5+ Area(z)	

Areas Under the One-Tailed Standard Normal Curve

This table provides the area between the mean and some Z score.
For example, when Z score = 1.45 the area = 0.4265.



Z	0.00	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09
0.0	0.0000	0.0040	0.0080	0.0120	0.0160	0.0199	0.0239	0.0279	0.0319	0.0359
0.1	0.0398	0.0438	0.0478	0.0517	0.0557	0.0596	0.0636	0.0675	0.0714	0.0753
0.2	0.0793	0.0832	0.0871	0.0910	0.0948	0.0987	0.1026	0.1064	0.1103	0.1141
0.3	0.1179	0.1217	0.1255	0.1293	0.1331	0.1368	0.1406	0.1443	0.1480	0.1517
0.4	0.1554	0.1591	0.1628	0.1664	0.1700	0.1736	0.1772	0.1808	0.1844	0.1879
0.5	0.1915	0.1950	0.1985	0.2019	0.2054	0.2088	0.2123	0.2157	0.2190	0.2224
0.6	0.2257	0.2291	0.2324	0.2357	0.2389	0.2422	0.2454	0.2486	0.2517	0.2549
0.7	0.2580	0.2611	0.2642	0.2673	0.2704	0.2734	0.2764	0.2794	0.2823	0.2852
0.8	0.2881	0.2910	0.2939	0.2967	0.2995	0.3023	0.3051	0.3078	0.3106	0.3133
0.9	0.3159	0.3186	0.3212	0.3238	0.3264	0.3289	0.3315	0.3340	0.3365	0.3389
1.0	0.3413	0.3438	0.3461	0.3485	0.3508	0.3531	0.3554	0.3577	0.3599	0.3621
1.1	0.3643	0.3665	0.3686	0.3708	0.3729	0.3749	0.3770	0.3790	0.3810	0.3830
1.2	0.3849	0.3869	0.3888	0.3907	0.3925	0.3944	0.3962	0.3980	0.3997	0.4015
1.3	0.4032	0.4049	0.4066	0.4082	0.4099	0.4115	0.4131	0.4147	0.4162	0.4177
1.4	0.4192	0.4207	0.4222	0.4236	0.4251	0.4265	0.4279	0.4292	0.4306	0.4319
1.5	0.4332	0.4345	0.4357	0.4370	0.4382	0.4394	0.4406	0.4418	0.4429	0.4441
1.6	0.4452	0.4463	0.4474	0.4484	0.4495	0.4505	0.4515	0.4525	0.4535	0.4545
1.7	0.4554	0.4564	0.4573	0.4582	0.4591	0.4599	0.4608	0.4616	0.4625	0.4633
1.8	0.4641	0.4649	0.4656	0.4664	0.4671	0.4678	0.4686	0.4693	0.4699	0.4706
1.9	0.4713	0.4719	0.4726	0.4732	0.4738	0.4744	0.4750	0.4756	0.4761	0.4767
2.0	0.4772	0.4778	0.4783	0.4788	0.4793	0.4798	0.4803	0.4808	0.4812	0.4817
2.1	0.4821	0.4826	0.4830	0.4834	0.4838	0.4842	0.4846	0.4850	0.4854	0.4857
2.2	0.4861	0.4864	0.4868	0.4871	0.4875	0.4878	0.4881	0.4884	0.4887	0.4890
2.3	0.4893	0.4896	0.4898	0.4901	0.4904	0.4906	0.4909	0.4911	0.4913	0.4916
2.4	0.4918	0.4920	0.4922	0.4925	0.4927	0.4929	0.4931	0.4932	0.4934	0.4936
2.5	0.4938	0.4940	0.4941	0.4943	0.4945	0.4946	0.4948	0.4949	0.4951	0.4952
2.6	0.4953	0.4955	0.4956	0.4957	0.4959	0.4960	0.4961	0.4962	0.4963	0.4964
2.7	0.4965	0.4966	0.4967	0.4968	0.4969	0.4970	0.4971	0.4972	0.4973	0.4974
2.8	0.4974	0.4975	0.4976	0.4977	0.4977	0.4978	0.4979	0.4979	0.4980	0.4981
2.9	0.4981	0.4982	0.4982	0.4983	0.4984	0.4984	0.4985	0.4985	0.4986	0.4986
3.0	0.4987	0.4987	0.4987	0.4988	0.4988	0.4989	0.4989	0.4989	0.4990	0.4990
3.1	0.4990	0.4991	0.4991	0.4991	0.4992	0.4992	0.4992	0.4992	0.4993	0.4993
3.2	0.4993	0.4993	0.4994	0.4994	0.4994	0.4994	0.4994	0.4995	0.4995	0.4995
3.3	0.4995	0.4995	0.4995	0.4996	0.4996	0.4996	0.4996	0.4996	0.4996	0.4997
3.4	0.4997	0.4997	0.4997	0.4997	0.4997	0.4997	0.4997	0.4997	0.4997	0.4998
3.5	0.4998	0.4998	0.4998	0.4998	0.4998	0.4998	0.4998	0.4998	0.4998	0.4998
3.6	0.4998	0.4998	0.4999	0.4999	0.4999	0.4999	0.4999	0.4999	0.4999	0.4999
3.7	0.4999	0.4999	0.4999	0.4999	0.4999	0.4999	0.4999	0.4999	0.4999	0.4999
3.8	0.4999	0.4999	0.4999	0.4999	0.4999	0.4999	0.4999	0.4999	0.4999	0.4999
3.9	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000	0.5000

Problem:- *X is a normally normally distributed variable with mean μ = 30 and standard deviation σ = 4. Find*

a) $P(x < 40)$
b) $P(x > 21)$
c) $P(30 < x < 35)$

Solution:

a) For x = 40, then

$z = x - \mu / \sigma$

$\Rightarrow z = (40 - 30) / 4$

$= 2.5$ ($=z_1$ say)

Hence $P(x < 40) = P(z < 2.5)$

$= 0.5 + A(z_1) = 0.9938$

b) For x = 21,

$z = x - \mu / \sigma$

$\Rightarrow z = (21 - 30) / 4$

$= -2.25$ ($= -z_1$ say)

Hence $P(x > 21) = P(z > -2.25)$

$= 0.5 - A(z_1) = 0.9878$

c) For x = 30

$z = x - \mu / \sigma \Rightarrow$,

$z = (30 - 30) / 4 = 0$ and

for x = 35,

$z = x - \mu / \sigma$

$\Rightarrow z = (35 - 30) / 4$

$= 1.25$

Hence $P(30 < x < 35) = P(0 < z < 1.25)$

= [area to the left of z = 1.25] - [area to the left of 0]

$= 0.8944 - 0.5 = 0.3944$

Problem:- *The length of life of an instrument produced by a machine has a normal ditribution with a mean of 12 months and standard deviation of 2 months. Find the probability that an instrument produced by this machine will last.*

a) less than 7 months.
b) between 7 and 12 months.

Solution:

a) $P(x < 7)$

for x = 7

$z = x - \mu / \sigma$

$\Rightarrow z = (7 - 12) / 2$

$= -2.5$ ($=z_1$ say)

Hence $P(x < 7) = P(z < -2.5)$

$= 0.0062$

b) $P(7 < x < 12)$

For x=12

$z = x - \mu / \sigma$

$\Rightarrow z = (12 - 12) / 2$

$= 0$ ($=z_1$ say)

Hence $P(7 < x < 12) = P(-2.5 < z < 0)$

$= 0.4938$

Problem:- *The Tahoe Natural Coffee Shop morning customer load follows a normal distribution with mean 45 and standard deviation 8. Determine the probability that the number of customers tomorrow will be less than 42.*

Solution:-

We first convert the raw score to a z-score. We have

$z = x - \mu / \sigma$

$\Rightarrow z = (42-45)/8 = -0.375$

Next, we use the table to find the probability. The table gives 0.3520. (We have rounded the raw score to -0.38).

We can conclude that

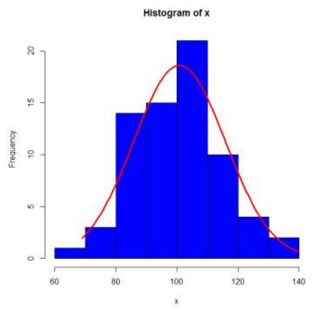
$P(x<42)=P(x<-0.38)$

$=0.352$

That is there is about a 35% chance that there will be fewer than 42 customers tomorrow.

Example:

```
> x <- c(92,117,109,85,117,107,82,83,119,113,101,106,101,84,126,69,82,79,84,100,104,111,109,92,93,107,
81,118,81,133,111,82,120,103,115,89,74,110,83,110,96,102,108,110,140,106,111,98,98,99,74,101,107,104,
128,87,95,109,104,91,83,98,99,103,126,123,85,98,93,100)

> h<-hist(x,col = "blue")
> m <- mean(x)
> s <- sd(x)
> xf  <- seq(min(x),max(x),length=70)
> dis <- dnorm(xf,m,s)
> dis <- dis*diff(h$mids[1:2]*length(x))
> lines(xf,dis,col="red",lwd=3)
```



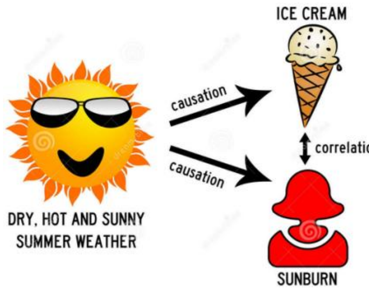
*Problem:-*Assume that the test scores of a college entrance exam fits a normal distribution. Furthermore, the mean test score is 72, and the standard deviation is 15.2. What is the percentage of students scoring 84 or more in the exam?

Solution:-

We apply the function pnorm of the normal distribution with mean 72 and standard deviation 15.2. Since we are looking for the percentage of students scoring higher than 84, we are interested in the *upper tail* of the normal distribution.

```
> pnorm(84, mean=72, sd=15.2, lower.tail=FALSE)
[1] 0.21492
```

Correlation:- A correlation is a relationship between two variables. Typically, we take x to be the independent variable. We take y to be the dependent variable. Data is represented by a collection of ordered pairs (x,y).



$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2 \sum_{i=1}^{n}(y_i-\bar{y})^2}}$$

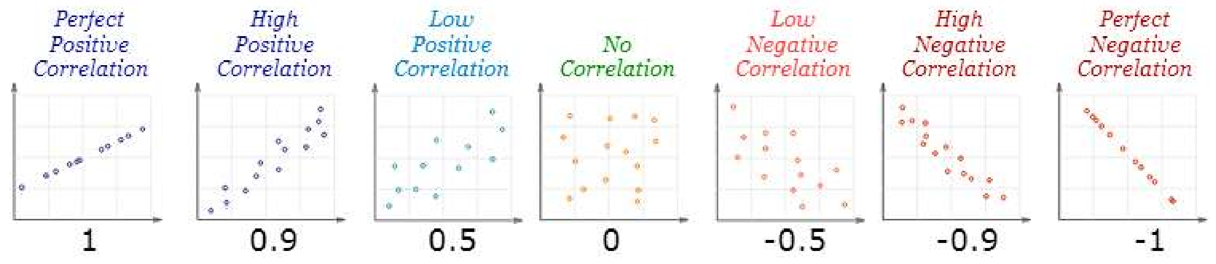
This will always be a number between -1 and 1 (inclusive).

- If r is close to 1, we say that the variables are positively correlated. This means there is likely a strong linear relationship between the two variables, with a positive slope.
- If r is close to -1, we say that the variables are negatively correlated. This means there is likely a strong linear relationship between the two variables, with a negative slope.
- If r is close to 0, we say that the variables are not correlated. This means that there is likely no linear relationship between the two variables, however, the variables may still be related in some other way.

To run a correlation test we type:

```
> cor.test(var1, var2, method = "method")
```

The default method is "pearson" so you may omit this if that is what you want. If you type "kendall" or "spearman" then you will get the appropriate significance test.



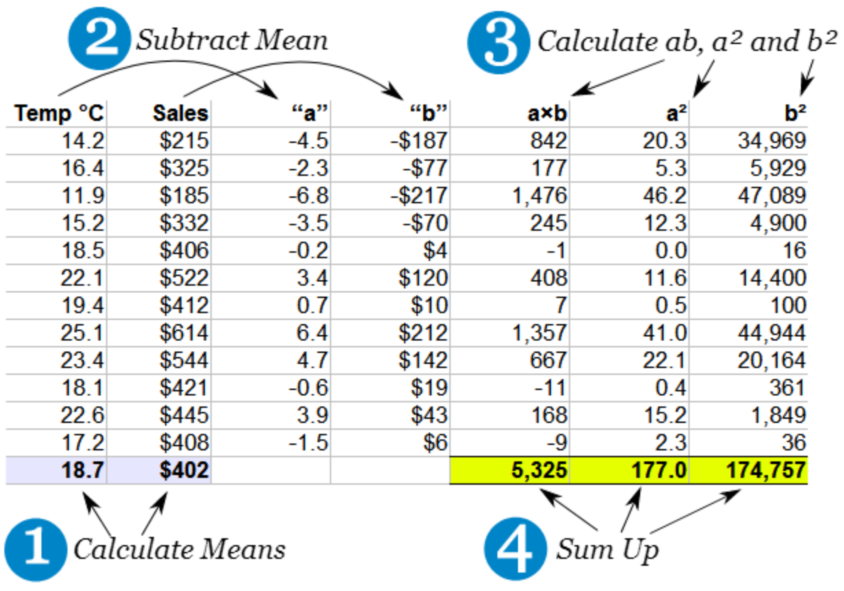
Problem:- The local ice cream shop keeps track of how much ice cream they sell versus the temperature on that day, here are their figures for the last 12 days:

Temperature oC	*14.2*	*16.4*	*11.9*	*15.2*	*18.5*	*22.1*	*19.4*	*25.1*	*23.4*	*18.1*	*22.6*	*17.2*
Ice cream sales	*$215*	*$325*	*$185*	*$332*	*$406*	*$522*	*$412*	*$614*	*$544*	*$421*	*$445*	*$408*

Solution:-

Formula for correlation coefficient:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2 \sum_{i=1}^{n}(y_i-\bar{y})^2}}$$

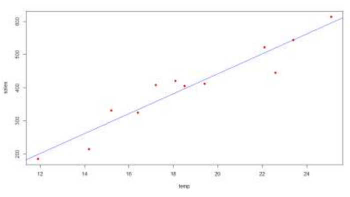

2 Subtract Mean
3 Calculate ab, a² and b²
1 Calculate Means
4 Sum Up

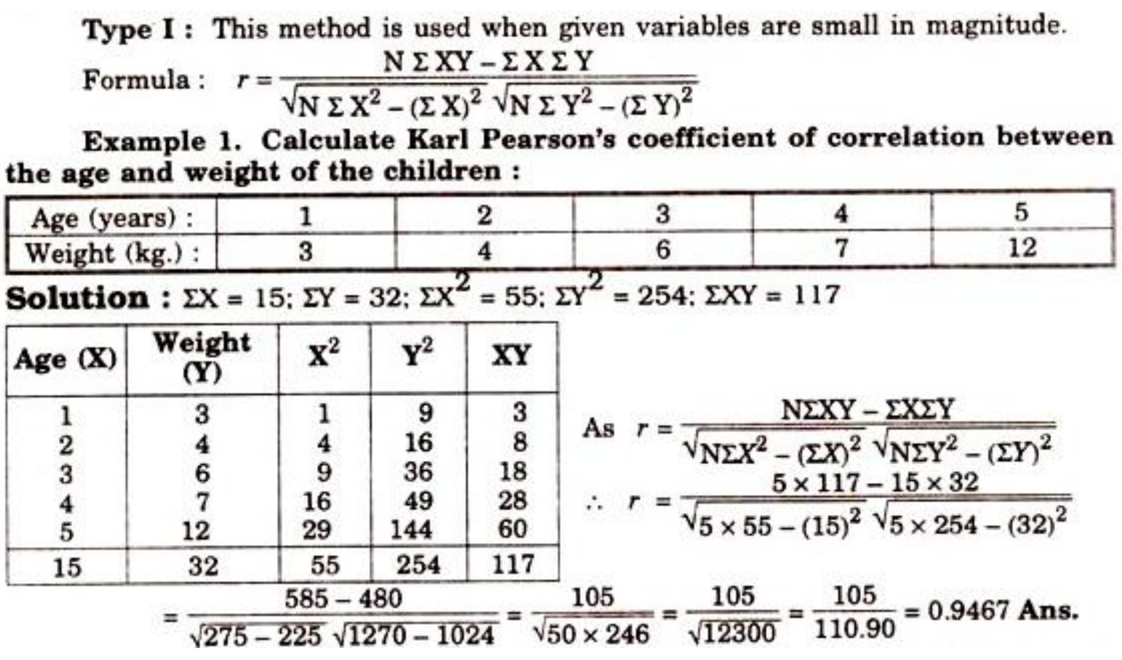
Temp °C	Sales	"a"	"b"	a×b	a²	b²
14.2	$215	-4.5	-$187	842	20.3	34,969
16.4	$325	-2.3	-$77	177	5.3	5,929
11.9	$185	-6.8	-$217	1,476	46.2	47,089
15.2	$332	-3.5	-$70	245	12.3	4,900
18.5	$406	-0.2	$4	-1	0.0	16
22.1	$522	3.4	$120	408	11.6	14,400
19.4	$412	0.7	$10	7	0.5	100
25.1	$614	6.4	$212	1,357	41.0	44,944
23.4	$544	4.7	$142	667	22.1	20,164
18.1	$421	-0.6	$19	-11	0.4	361
22.6	$445	3.9	$43	168	15.2	1,849
17.2	$408	-1.5	$6	-9	2.3	36
18.7	**$402**			**5,325**	**177.0**	**174,757**

5 $$\frac{5,325}{\sqrt{177.0 \times 174,757}} = 0.9575$$

R Code:-

```
> temp <- c(14.2,16.4,11.9,15.2,18.5,22.1,19.4,25.1,23.4,18.1,22.6,17.2)
> sales <- c(215,325,185,332,406,522,412,614,544,421,445,408)
> corr_coeff <- cor(temp,sales)
> corr_coeff
        [1] 0.9575066
> cov(temp,sales)
        [1] 484.0932
#Adds a line of best fit to your scatter plot
> plot(temp, sales, pch=16,col="red")
>abline(lm(sales~temp),col="blue")
```





Type I : This method is used when given variables are small in magnitude.

Formula : $r = \dfrac{N\,\Sigma XY - \Sigma X\,\Sigma Y}{\sqrt{N\,\Sigma X^2 - (\Sigma X)^2}\,\sqrt{N\,\Sigma Y^2 - (\Sigma Y)^2}}$

Example 1. Calculate Karl Pearson's coefficient of correlation between the age and weight of the children :

Age (years) :	1	2	3	4	5
Weight (kg.) :	3	4	6	7	12

Solution : $\Sigma X = 15$; $\Sigma Y = 32$; $\Sigma X^2 = 55$; $\Sigma Y^2 = 254$; $\Sigma XY = 117$

Age (X)	Weight (Y)	X^2	Y^2	XY
1	3	1	9	3
2	4	4	16	8
3	6	9	36	18
4	7	16	49	28
5	12	29	144	60
15	32	55	254	117

As $r = \dfrac{N\Sigma XY - \Sigma X\Sigma Y}{\sqrt{N\Sigma X^2 - (\Sigma X)^2}\,\sqrt{N\Sigma Y^2 - (\Sigma Y)^2}}$

$\therefore\ r = \dfrac{5 \times 117 - 15 \times 32}{\sqrt{5 \times 55 - (15)^2}\,\sqrt{5 \times 254 - (32)^2}}$

$$= \frac{585 - 480}{\sqrt{275 - 225}\,\sqrt{1270 - 1024}} = \frac{105}{\sqrt{50 \times 246}} = \frac{105}{\sqrt{12300}} = \frac{105}{110.90} = 0.9467 \text{ Ans.}$$

T-test for single mean:- One-sample t-test is used to compare the mean of a population to a specified theoretical mean (μ).

Let X represents a set of values with size n, with mean μ and with standard deviation S. The comparison of the observed mean (μ) of the population to a theoretical value μ is performed with the formula below:

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}$$

To evaluate whether the difference is statistically significant, you first have to read in t test table the critical value of Student's t distribution corresponding to the significance level alpha of your choice (5%). The degrees of freedom (df) used in this test are: df = n−1

Problem:-: *A professor wants to know if her introductory statistics class has a good grasp of basic math. Six students are chosen at random from the class and given a math proficiency test. The professor wants the class to be able to score above 70 on the test. The six students get scores of 62, 92, 75, 68, 83, and 95. Can the professor have 90 percent confidence that the mean score for the class on the test would be above 70?*

Solution:-

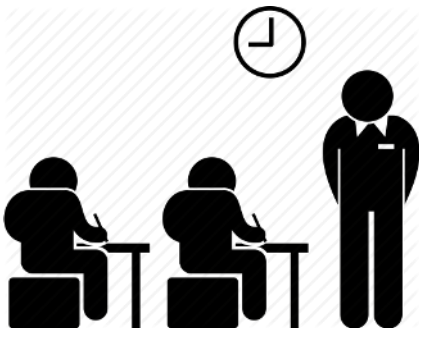
Null hypothesis: H_0: μ = 70

Alternative hypothesis: H_a : μ > 70

First, compute the sample mean and standard deviation:

$$\bar{x} = \frac{62+92+75+68+83+95}{6}$$

$$= \frac{475}{6} = 13.17$$



- *Null Hypothesis* H_0*:* The sample meet upto standard i.e μ >70 hours
- *Alternative Hypothesis* H_A: μ not greater than 70,
- *Level of Siginificance:* $\alpha = 0.05$
- *The test statistic is* $t = \dfrac{\bar{x} - \mu_0}{s/\sqrt{n}}$

$$t = \frac{79.71 - 70}{13.17/\sqrt{6}} = \frac{9.17}{5.38}$$

= 1.71(calculate value of t)

To test the hypothesis, the computed *t*-value of 1.71 will be compared to the critical value in the *t*-table with 5 df is 1.67, the calculate of t is more than table value of t, so null hypothsis is rejected.

R code:-

```
    > t.test(x,alternative="two.sided",mu=70)

            One Sample t-test

    data:  x
    t = 1.7053, df = 5, p-value = 0.1489
    alternative hypothesis: true mean is not equal to 70
    95 percent confidence interval:
     65.34888 92.98446
    sample estimates:
    mean of x
     79.16667
```

Problem:-: A Sample of 26 bulbs gives a mean life of 990 hours with S.D of 20 hours. The manufacurer claims that the mean life of bulbs is 1000 hours. Is sample meet upto the standard.

Solution: Here n = 26,

Sample mean $\bar{x}$ = 990 hours

S.D s = 20 hours

Population mean μ = 1000 hours

Df = n-1 = 26-1 = 25

- ***Null Hypothesis*** ***H_0:*** The sample meet upto standard i.e μ = 1000 hours
- ***Alternative Hypothesis*** ***H_A***: μ not equal to 1000,
- ***Level of Siginificance:*** $\alpha = 0.05$
- ***the test statistic is***

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}$$

t = 990-1000/20/√26

= 2.5 (calculate value of t)

Table value of t with 25 df is 1.708

The calculate value of t is more than table value of t, so null hypotheis is rejected at 5% level.

Paired comparisons(Paired t-test):- Sometimes data comes from non independent samples. An example might be testing "before and after" of cosmetics or consumer products. We could use a single random sample and do "before and after" tests on each person. A hypothesis test based on these data would be called a *paired comparisons test*. Since the observations come in pairs, we can study the difference, d, between the samples. The difference between each pair of measurements is called di.

Test statistic:- With a population of n pairs of measurements, forming a simple random sample from a normally distributed population, the mean of the difference, $\bar{d}$, is tested using the following implementation of *t*.

$$t = \frac{\bar{d} - \mu}{S/\sqrt{n}}$$

Problem :- The blood pressure of 5 women before and after intake of a certain drug are given below: Test whether there is significant change in blood pressure at 1% level of significance.

Before	*110*	*120*	*125*	*132*	*125*
After	*120*	*118*	*125*	*136*	*121*

Solution: Let μ be the mean of population of differences.

- ***Null Hypothesis*** ***H_0:*** $\mu_1 = \mu_2$ i,e, no change in B.P.
- ***Alternative Hypothesis*** ***H_A***: $\mu_1 \neq \mu_2$ i,e, no change in B.P.
- ***Level of Siginificance:*** $\alpha = 0.01$
- ***Computation :*** Differences d_i's (before and after drug) are

-10,2,0,14,4

$$\bar{d} = \frac{-10+2+0+-4+4}{5}$$

$$= \frac{-8}{5} = -1.6$$

$$S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(d_i - \bar{d})^2$$

$$= \frac{1}{4}\sum_{i=1}^{5}(d_i - \bar{d})^2$$

$$= \frac{1}{4}[(-10+1.6)^2 + (2+1.6)^2 + (0+1.6)^2 + (-4+1.6)^2 + (4+1.6)^2]$$

$$= \frac{123.20}{4} = 30.8$$

$$S = \sqrt{30.8} = 5.55$$

- ***Test statistic:*** The test statistic is t which is calculated as

$$t = \frac{\bar{d} - \mu}{S/\sqrt{n}}$$

$$= \frac{-1.16}{5.55/\sqrt{5}} = -0.645$$

Calculated |t| value is 0.645

Tabulates $t_{0.01}$ with 5-1 = 4 degrees of freedom is 3.747.

Since calculated t < $t_{0.01}$, we accept the Null hypothesis and conclude that there is no significant change in blood pressure.

R code:-

```
> x <- c(110,120,125,132,125)
> y <- c(120,118,125,136,121)
> t.test(x,y,paired=TRUE)

        Paired t-test

data:  x and y
t = -0.64466, df = 4,
p-value = 0.5543
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -8.490956  5.290956
sample estimates:
mean of the differences
         -1.6
```

T-test for difference of two population means :-

With a two-sample t-test, we compare the population means to each other and again look at the difference. We expect that $\bar{x} - \bar{y}$ would be close to $\mu_1 - \mu_2$. The test statistic will use both sample means, sample standard deviations, and sample sizes for the test.

A two-sample t-test follows

- Write the null and alternative hypotheses.

- State the level of significance and find the critical value. The critical value, from the student's t-distribution, has the lesser of n_1-1 and n_2 -1 degrees of freedom.
- Compute the test statistic.
- Compare the test statistic to the critical value and state a conclusion.

$$t = \frac{\bar{x}-\bar{y}}{S\sqrt{\frac{1}{n_1}+\frac{1}{n_2}}} \qquad \sim t_{n_1+n_2-2}$$

where

$$S^2 = \frac{n_1 s_1^2 + n_2 s_2^2}{n_1+n_2-2} \quad \text{or} \quad S^2 = \frac{\sum(x_i-\bar{x})^2 + \sum(y_i-\bar{y})^2}{n_1+n_2-2}$$

Problem:- *Two horses A and B were tested according to the time (in seconds) to run a particular track with the following results.*

Horse A	*28*	*30*	*32*	*33*	*33*	*29*	*34*
Horse B	*29*	*30*	*30*	*24*	*27*	*29*	

Test whether the two horses have the same running capacity.

Solution:- Given n1=7 and n2 = 6

We first compute the same means and standard deviations.

$$\bar{x} = \text{Mean of the first sample}$$

$$= \frac{1}{7}(28+30+32+33+33+29+34) = \frac{1}{7}(219) = 31.286$$

$$\bar{y} = \text{Mean of the second sample}$$

$$= \frac{1}{6}(29+30+30+24+27+29) = \frac{1}{6}(169) = 28.16$$

x	$x-\bar{x}$	$(x-\bar{x})^2$	y	$y-\bar{y}$	$(y-\bar{y})^2$
28	-3.286	10.8	29	0.84	0.7056
30	-1286	1.6538	30	1.84	3.3856
32	0.714	0.51	30	1.84	3.3856
33	1.714	2.94	24	-4.16	17.3056
33	1.714	2.94	27	-1.16	1.3456
29	-2.286	5.226	29	0.84	0.7056
34	2.714	7.366			
219		31.4359	169		26.8336

$$\text{Now, } S^2 = \frac{\sum(x_i-\bar{x})^2 + \sum(y_i-\bar{y})^2}{n_1+n_2-2}$$

$$= \frac{(31.4358+26.8336)}{7+6-2} = 5.23$$

$$\textit{Therefore} \quad S = \sqrt{5.23} = 2.3$$

- ***Null Hypothesis*** H_0: $\mu_1 = \mu_2$
- ***Alternative Hypothesis*** H_A: $\mu_1 \neq \mu_2$
- ***Level of Siginificance:*** $\alpha = 0.05$

- *Computation :* $t = \dfrac{\bar{x} - \bar{y}}{S\sqrt{\dfrac{1}{n_1} + \dfrac{1}{n_2}}} = \dfrac{31.286 - 28.16}{(2.3)\sqrt{\dfrac{1}{7} + \dfrac{1}{6}}} = 2.443$

Tabulates $t_{0.05}$ with 7+6-2 = 11 degrees of freedom at 5% level of significance is 2.2
Since calculated t > $t_{0.05}$, we reject the Null hypothesis and conclude that there is no significant change in blood pressure.

ANOVA:- (ANALYSIS OF VARIANCE)

When we have only two samples we can use the t-test to compare the means of the samples but it might become unreliable in case of more than two samples. If we only compare two means, then the t-test (independent samples) will give the same results as the ANOVA. Anova is performed with F-test.

Null hypothesis H0: There are no differences among the mean values of the groups being compared (i.e., the group means are all equal)–

H0: µ1 = µ2 = µ3 = …= µk

Alternative hypothesis H1: (Conclusion if H0 rejected)?
Not all group means are equal (i.e., at least one group mean is different from the rest).

ANOVA one-way classification:-

Step 1: Total number of all observations

$$T = \sum_i \sum_j X_{ij}$$

Step 2: Correlation factor

$$cf = \frac{T^2}{N} = \frac{T^2}{r \times s}$$

Step 3:Total sum of squares

$$\text{TSS} = \text{S}^2\text{T} = \sum_i \sum_j X_{ij}^{\ 2} - cf$$

Step 4: Treatment sum of squares

$$\text{TrSS} = \text{S}^2\text{Tr} = \sum \frac{T_j^{\ 2}}{N} - cf$$

Step 5: Error sum of squares

ESS = S^2E = TSS-TrSS

Source of variable	d.f	Sum of Squares	TSS	F-Test
Treatment (between sample)	k-1	$S^2Tr = \sum \frac{T_j^{\ 2}}{N} - cf$	$S^2Tr = \frac{ST_r^{\ 2}}{k-1}$	$F_{cal} = \frac{S^2Tr}{S^2E}$
Error	n-k	S^2E = TSS-TrSS	$S^2E = \frac{S^2E}{n-k}$	