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.

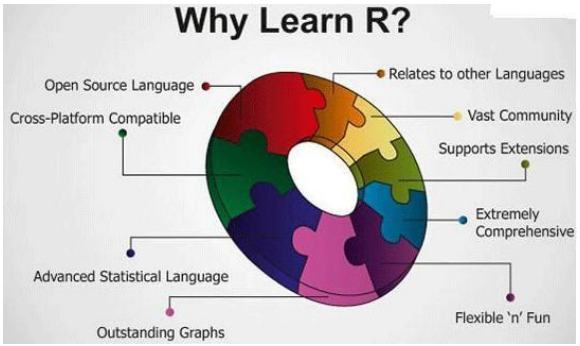
Introduction: R is a scripting language (are often interpreted rather than compiled) for statistical data manipulation and analysis. It was inspired by, and is mostly compatible with, the statistical language S developed by AT&T.R is designed by Ross ihanka and Robert Gentleman, developed by R core team.

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.



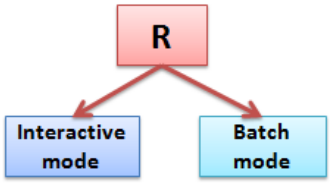
R Features

- 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 - *C*ode 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.

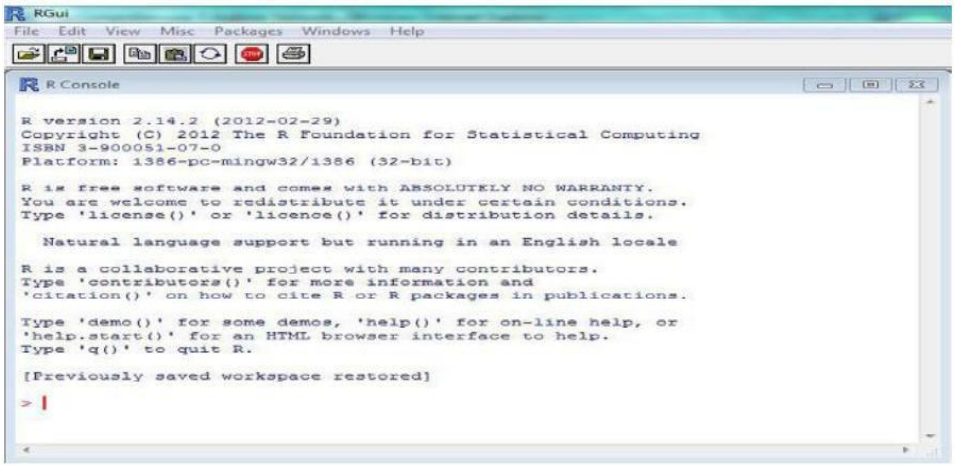
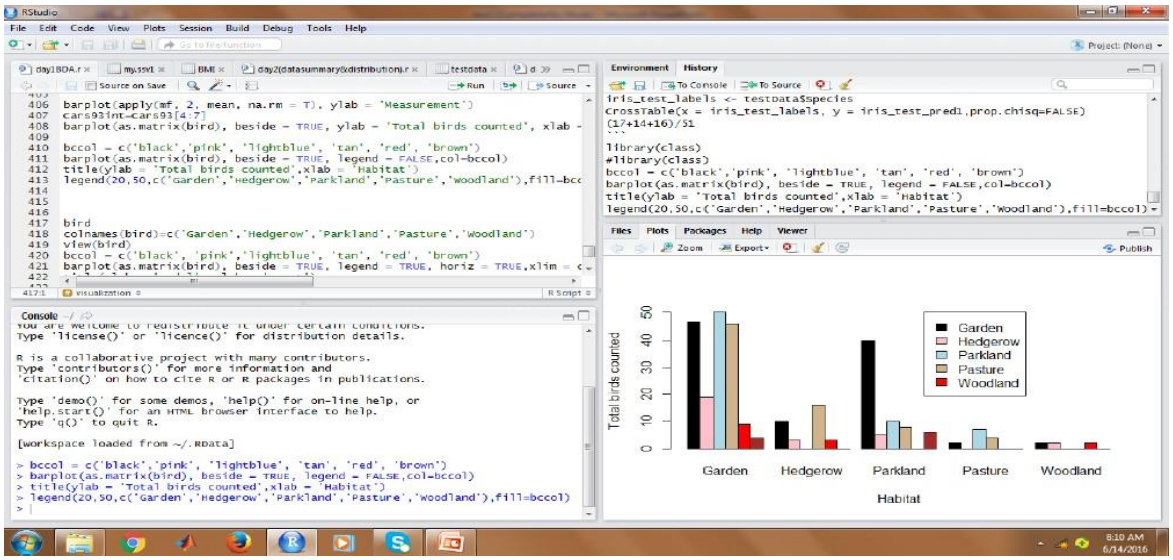


Figure 1. R console

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/.



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

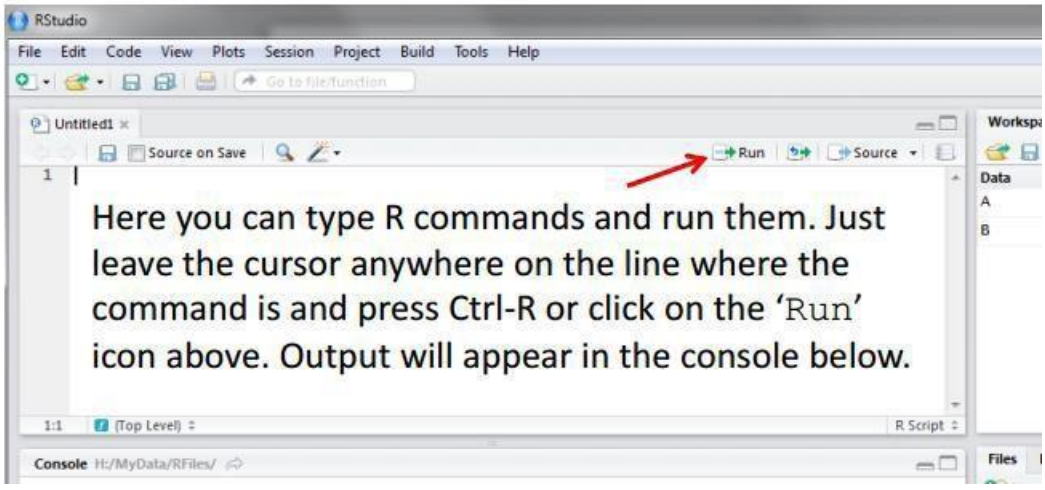
1. Console.
2. Workspace and history.
3. Files, plots, packages and help.
4. The R script(s) and data view.

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

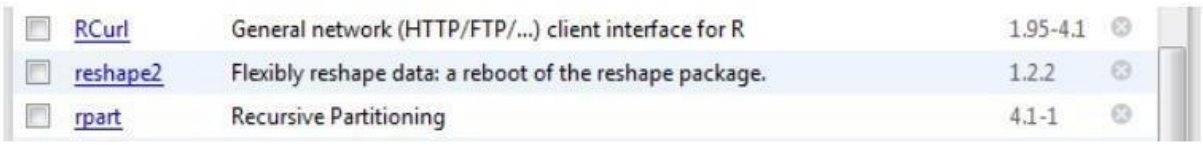
Create a new R script file:

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

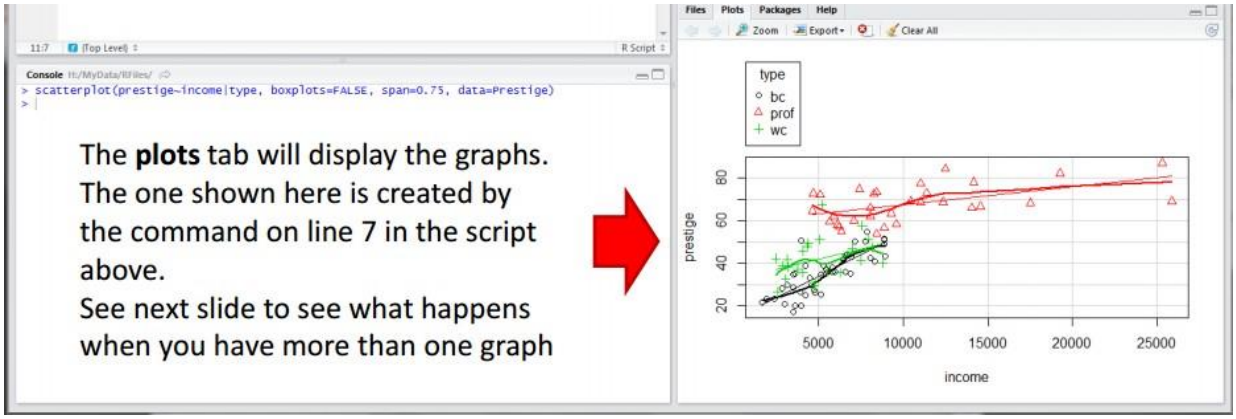
Running the R commands on R Script file:



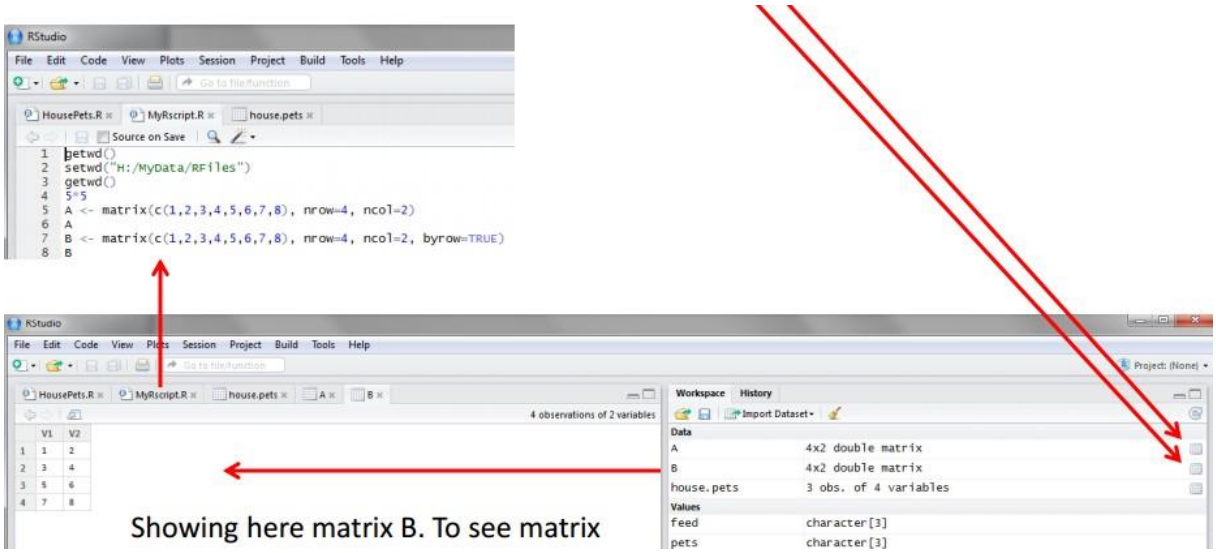
Installing Packages:



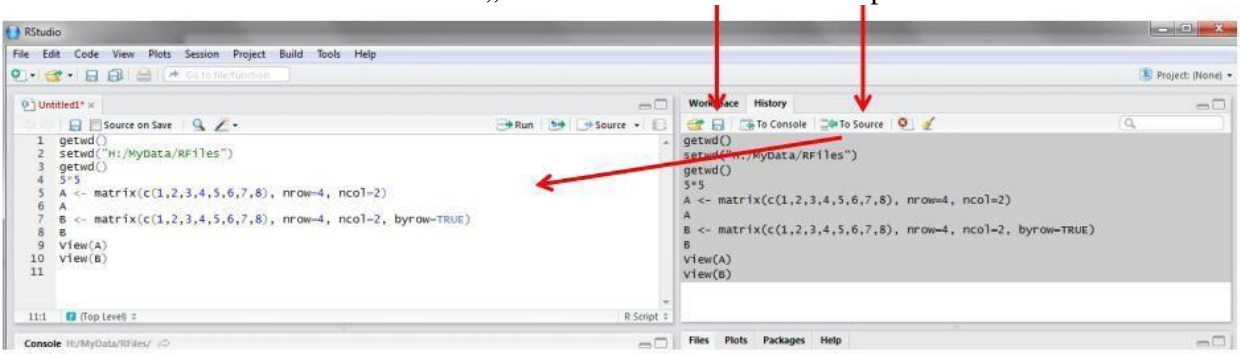
Plots to display:



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.



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.



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.

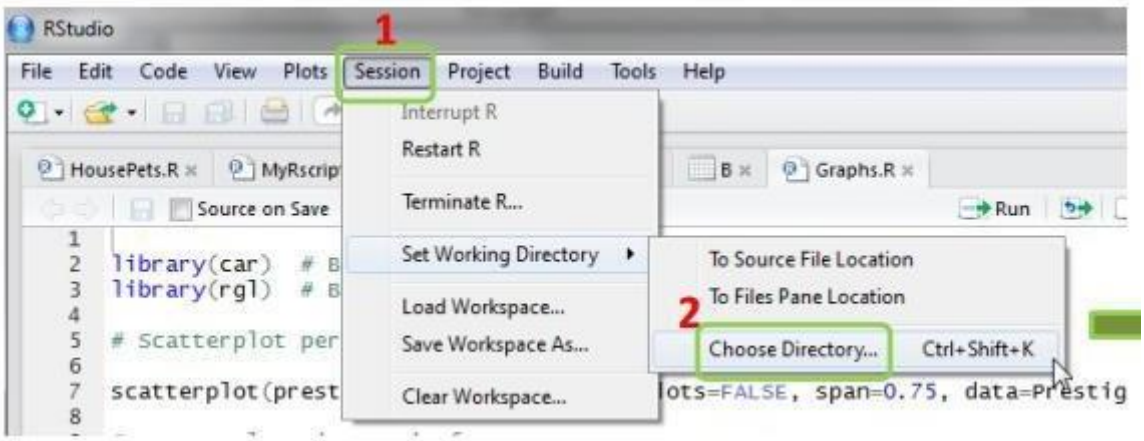
Changing the working directory:

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")
```



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]

		[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.

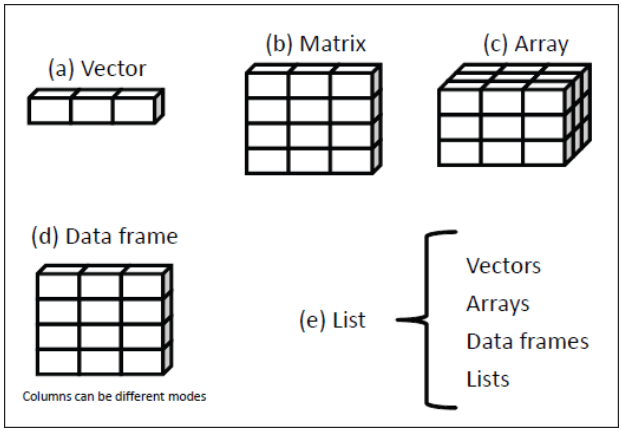
- 'class' is a property assigned to an object that determines how generic functions operate with it. It is not a mutually exclusive classification. If an object has no specific class assigned to it, such as a simple numeric vector, it's class is usually the same as its mode, by convention.Changing the 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)
```


jntuk396.blogspot.com

```
> 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 = 2¤2r2R 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 (¤r2) and multiplying it by the circumference of the donut (2¤r).
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.