Section 1

Chapter 1: Overview & environment setup

Chapter 2: Expression, Variables and functions

Chapter 3: Vectors

Chapter 4: Matrices

Chapter 5: Factors

Chapter 6: List

Chapter 7: Data Frames

Section 2: R Data Interfaces

Chapter 1: R – CSV files

Chapter 2: R – XML Files

Chapter 3: R – JSON files

Section 3: R Charts and Graphs

Chapter 1: R – Pie charts

Chapter 2: R – Bar Charts

Chapter 3 – R boxplots

Chapter 4: R Histograms

Chapter 5: Scatterplots

Section 4: R Statistics

Chapter 1: Mean, Median & Mode

Chapter 2: Standard Deviation and Variance

Section 1

Chapter 1: Introduction to R

Why should you learn R?

Few reasons to learn R:

- If you have a need to run statistical calculations in your application. Learn and deploy R! It integrates with programming languages such as Java, C++, Python, Ruby.
- If you need to run your own analysis, think of R.
- If you are working on an optimization problem, use R.
- If there is a need to use reusable libraries to solve a complex problem. Leverage the 2000+ free libraries provided by R.
- If you wish to create compelling charts, leverage the power of R.
- If you aspire to be a Data Scientist in future, you will learn R.
- If you wish to have fun with statistics, you will learn R.

What is R?

R is a scripting/programming language and environment for statistical computing and graphics. It was inspired by, and is mostly compatible with, the statistical language S developed at Bell laboratory (formerly AT & T, now Lucent technologies). Although there are some very important differences between R and S, nevertheless much of the code written for S runs unaltered on R. R has become popular as the single most important tool for computational statistics, visualization and data science.

Why R?

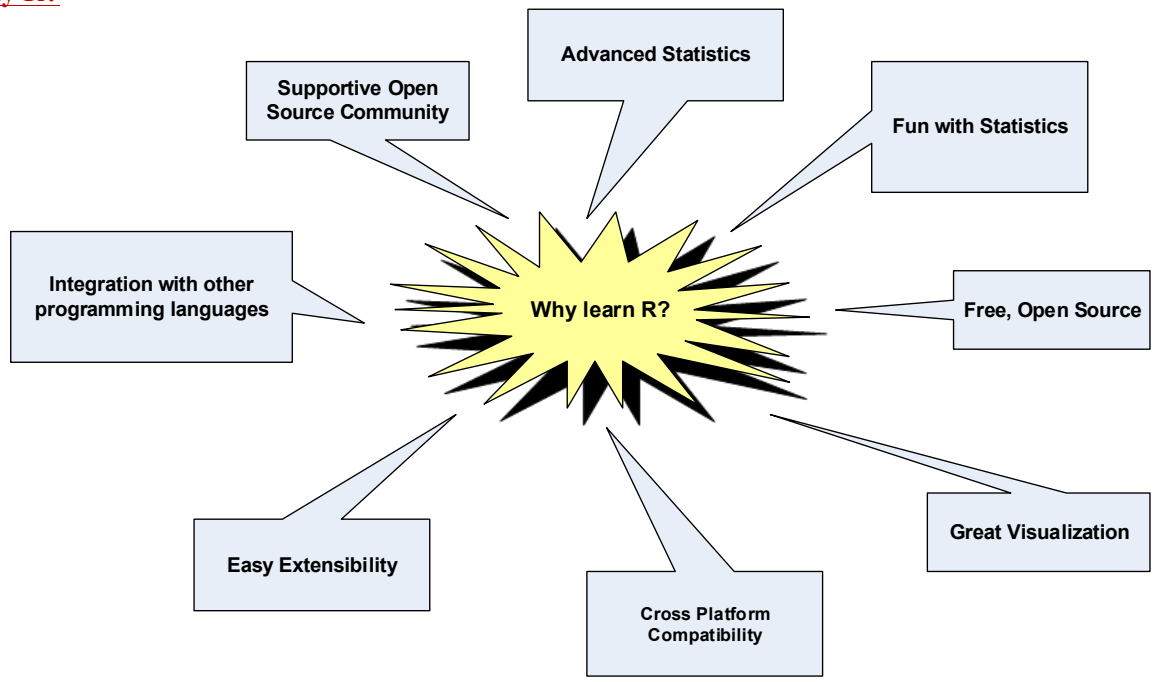


Figure 1: Why learn R?

Refer to Figure 1.

- R is free. It is available under the terms of the Free Software Foundation's GNU General Public License in source code form
- Available for Windows, Macs, wide variety of Unix platforms (including FreeBSD, Linux, etc.)
- in addition to enabling statistical operations, it's a general programming language, so that you can
- automate your analyses and create new functions
- has excellent tools for creating graphics from bar charts to scatter plots to multi-panel lattice charts
- object-oriented and functional programming structure
- Support from a robust, vibrant community
- Has a flexible analysis tool kit: this makes it easy to access data in various formats, manipulate it (transform, merge, aggregate, etc.), and subject it to traditional and modern statistical models (such as regression, ANOVA, tree models etc.)
- R can be extended easily via packages
- R relates easily to other programming languages. Existing software as well as emerging software can be integrated with R packages to make them more productive
- R can easily import data from MS Excel, MS Access, MySQL, SQLite, Oracle etc. It can easily connect to databases using ODBC (Open Database Connectivity Protocol) and ROracle Package

History of R

R was first created by Ross Ihaka and Robert Gentleman at the University of Auckland in 1993.

Where R is being used?

- Google
- LinkedIn
- Facebook
- IBM
- Bing
- Mozilla
- SAP
- Oracle
- New York Times
- etc.

Download and install

Step 1: Visit the site: http://cran.r-project.org/
Step 2: One can download R for windows. It comes with a GUI called the RGui. Click on "Download R for Windows"
Step 3: Follow the instructions of the R setup wizard for successfully installing R.

CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2015-12-10, Wooden Christmas-Tree) R-3.2.3.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

What are R and CRAN?

R is 'GNU S', a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc. Please consult the R project homepage for further information.

CRAN is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R. Please use the CRAN mirror nearest to you to minimize network load.

Submitting to CRAN

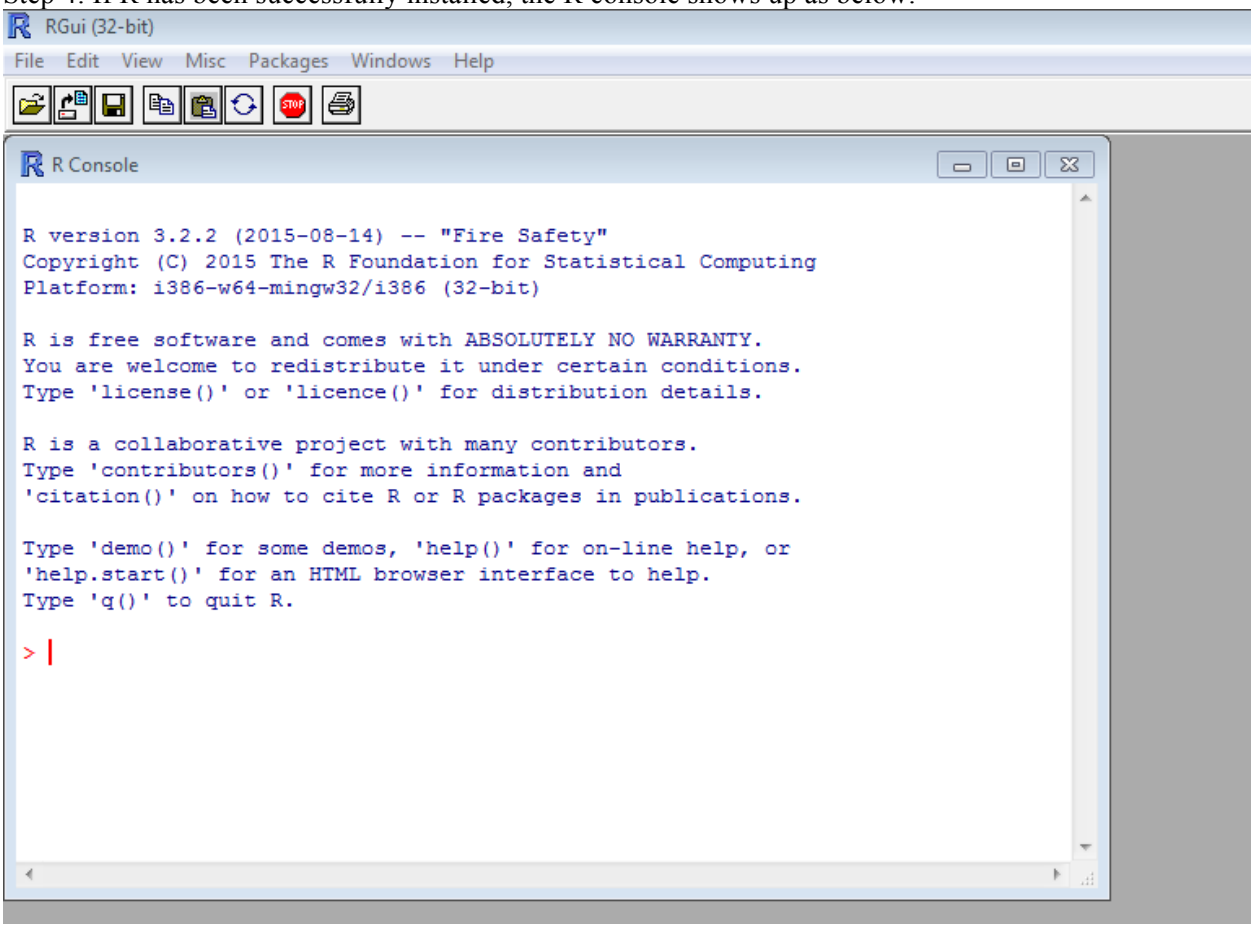
To "submit" a package to CRAN, check that your submission meets the CRAN Repository Policy and then use the web form.

If this fails, upload to ftp://CRAN.R-project.org/incoming/ and send an email to CRAN@R-project.org following the policy. Please do not attach submissions to emails, because this will clutter up the mailboxes of half a dozen people.

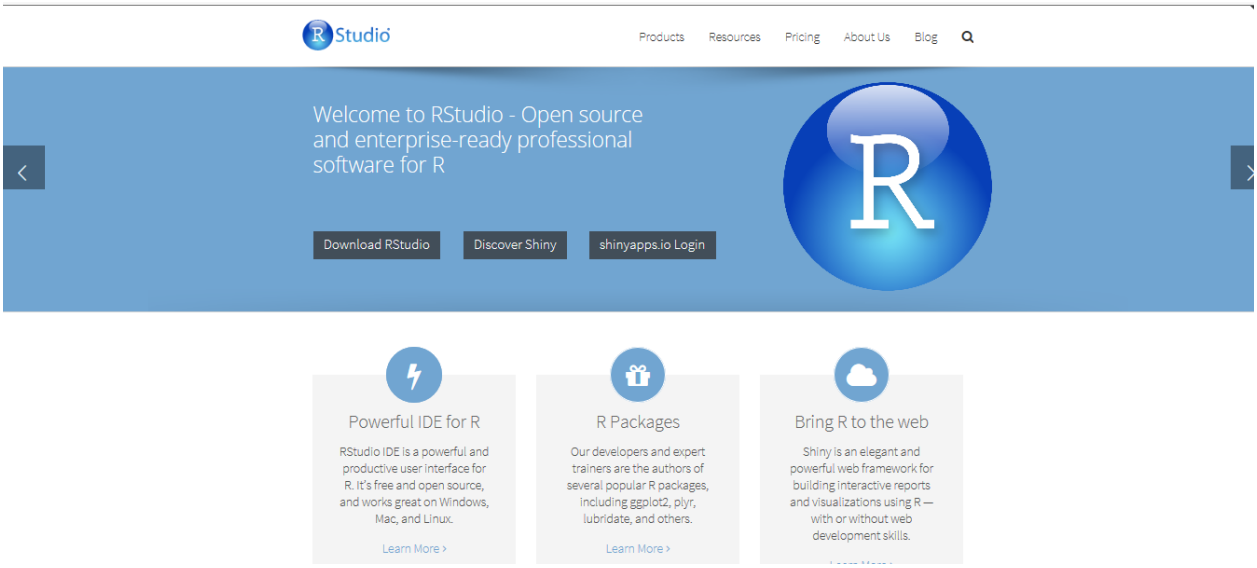
Note that we generally do not accept submissions of precompiled binaries due to security reasons. All binary distribution listed above are compiled by selected maintainers, who are in charge for all binaries of their platform, respectively.

This server is hosted by the Institute for Statistics and Mathematics of WU (Wirtschaftsuniversität Wien).

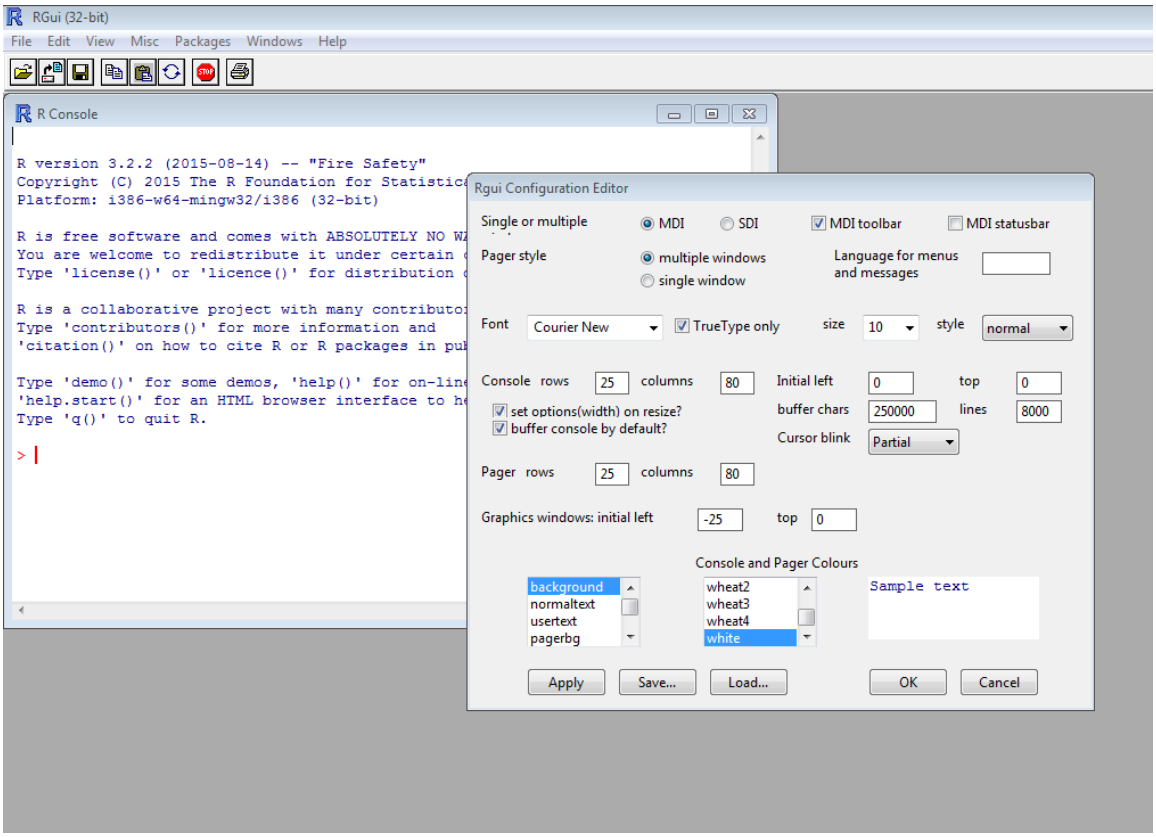
Step 4: If R has been successfully installed, the R console shows up as below:



Alternatively one can use another GUI called R studio. This can be downloaded from Rstudio.com



Let us take a look at the R console and set few GUI preferences.



Use Clear console from the menu option or Ctrl + L to clear the console or screen.

Installing R packages

R comes with some standard packages that are installed when you install R. However follow the steps given below on "how to install R package" to install additional R packages

1. To start R, follow either step 2 or 3. The assumption is that R is already installed on your machine.
2. If there is an "R" icon on the desktop of the computer that you are using, double-click on the "R" icon to start R. If there is no "R" icon on the desktop then click on the "Start" button at the bottom left of your computer screen, and then choose "All programs", and start R by selecting "R" (or R X.X.X, where X.X.X gives the version of R, eg. R 2.10.0) from the menu of programs.
3. The R console should show up.
4. Once you have started R, you can now install an R package (eg. the "ggplot2" package) by choosing "Install package(s)" from the "Packages" menu at the top of the R console. This will ask you for the website that you wish to download the package from, you can choose "Iceland" (or another country, if you prefer). It will also bring up a list of available packages that you can install, and you can choose the package that you want to install from that list (eg. "ggplot2").
5. This will install the "ggplot2" package.
6. The "ggplot2" package is now installed. Whenever you want to use the "ggplot2" package after this, after having successfully started R, you first have to load the package by typing into the R console:
   library("ggplot2").
7. You can get help on a package by tying the following at the R prompt.
   help(package = "ggplot2")

Let us get familiar with the interface by studying a few important commands / functions.

getwd() function

Description: The getwd() function returns the absolute path representing the current working directory. It returns a character string or NULL if the current working directory is not available.

Syntax: getwd()

```
> getwd()
[1] "C:/Users/seema_acharya/Documents"
```

setwd() function

Description: The setwd() function sets the working directory to dir.

Syntax: setwd(dir)

```
> setwd("D:/PracticeRProgramming")
```

To confirm, "D:/PracticeRProgramming" is now the working directory, let us run the getwd() function.

```
> getwd()
[1] "D:/PracticeRProgramming"
```

Dir() function

This is equivalent to list.files() function.

Description: This function returns a character vector of the names of files or directories in the named directory.

Syntax:

```
dir(path = ".", pattern = NULL, all.files = FALSE,
        full.names = FALSE, recursive = FALSE,
        ignore.case = FALSE, include.dirs = FALSE, no.. = FALSE)
```

or

```
list.files(path = ".", pattern = NULL, all.files = FALSE,
        full.names = FALSE, recursive = FALSE,
        ignore.case = FALSE, include.dirs = FALSE, no.. = FALSE)
```

```
> dir()
character(0)

> list.files()
character(0)
```

The above command implies that there are no files or directories in the current directory.

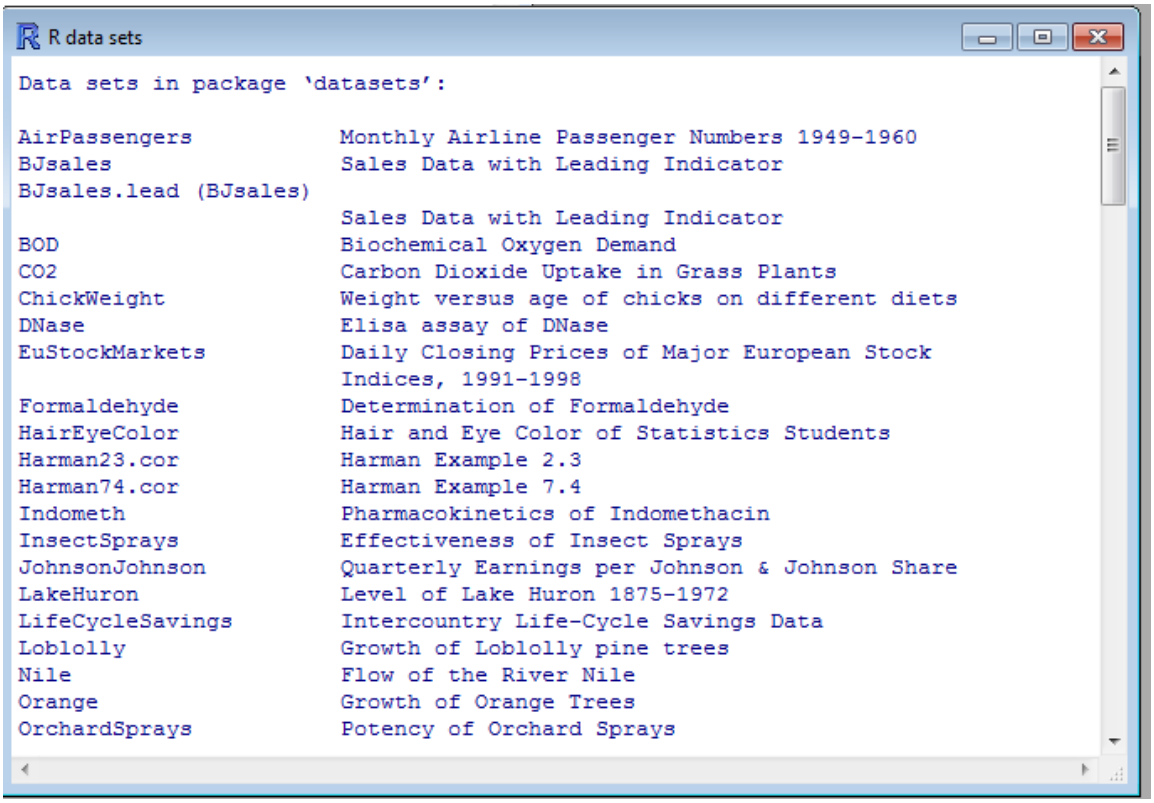
Data() function

Description: The data() function lists the available datasets.

Syntax:

```
> data()
```

Output:



Data(trees) function loads the dataset, “trees”.

Syntax:

```
> data(trees)
```

Let us look at the data held in the dataset, “trees”

```
> trees
   Girth Height Volume
1    8.3     70   10.3
2    8.6     65   10.3
3    8.8     63   10.2
4   10.5     72   16.4
5   10.7     81   18.8
6   10.8     83   19.7
7   11.0     66   15.6
8   11.0     75   18.2
9   11.1     80   22.6
10  11.2     75   19.9
11  11.3     79   24.2
12  11.4     76   21.0
13  11.4     76   21.4
14  11.7     69   21.3
15  12.0     75   19.1
16  12.9     74   22.2
17  12.9     85   33.8
18  13.3     86   27.4
19  13.7     71   25.7
20  13.8     64   24.9
21  14.0     78   34.5
22  14.2     80   31.7
23  14.5     74   36.3
24  16.0     72   38.3
25  16.3     77   42.6
26  17.3     81   55.4
27  17.5     82   55.7
28  17.9     80   58.3
29  18.0     80   51.5
30  18.0     80   51.0
31  20.6     87   77.0
```

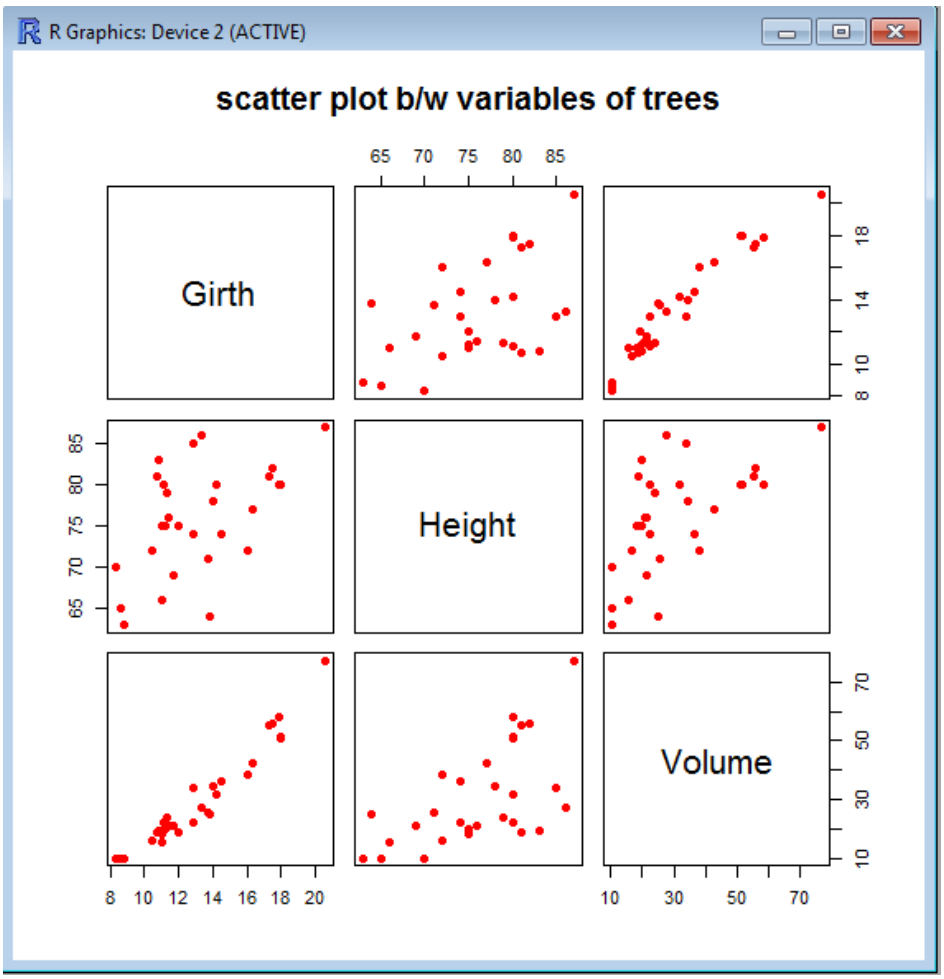
This data set provides measurements of the girth, height and volume of timber in 31 felled black cherry trees.

Let us look at a summary analysis on this dataset.

```
> summary(trees)
     Girth           Height       Volume     
 Min.   : 8.30   Min.   :63   Min.   :10.20  
 1st Qu.:11.05   1st Qu.:72   1st Qu.:19.40  
 Median :12.90   Median :76   Median :24.20  
 Mean   :13.25   Mean   :76   Mean   :30.17  
 3rd Qu.:15.25   3rd Qu.:80   3rd Qu.:37.30  
 Max.   :20.60   Max.   :87   Max.   :77.00  
```

Let us get our first taste of visualization by plotting a scatter plot between the variables of the dataset, “trees”.

```
> plot(trees, col="red", pch=16,main="scatter plot b/w variables of trees")
```



save.image()

Description: It writes an external representation of R objects to the specified file. At a later point in time when it is required to read back the objects, one can use the load or attach function.

Syntax:

```
save.image(file = ".RData", version = NULL, ascii = FALSE,
        safe = TRUE)
```

The file is to be given an extension of RData.

Note: The “R” and “D” in “RData” should be in capitals.

ascii :- If ascii = TRUE will save an ascii representation of the file. The default is ascii = FALSE. With ascii being set to false, a binary representation of the file is saved.

version :- this is used to specify the current workspace format version. The value of NULL specifies the current default format.

safe :- this is set to a logical value. A value of TRUE means that a temporary file is used to create the saved workspace. This temporary file is renamed to file if the save succeeds.

Chapter 2: Expression, Variables and Functions

Let us get familiar using the R interface. We will start out by practicing expressions, variables and functions.

Expressions

Let us look at a few arithmetic operations such as addition, subtraction, multiplication, and division, exponentiation, finding the remainder (modulus), integer division and computing the square root.

Operation	Operator	Description	Example
Addition	x + y	y added to x	`> 4 + 8` `[1] 12`
Subtraction	x - y	y subtracted from x	`> 10 - 3` `[1] 7`
Multiplication	x * y	x multiplied by y	`> 7 * 8` `[1] 56`
Division	x / y	x divided by y	`> 8 / 3` `[1] 2.666667`

Exponentiation	x ^ y
	
	x ** y
		x raised to the power y	`> 2 ^ 5`
			`[1] 32`
			Or
			`> 2 ** 5`
			`[1] 32`

Modulus	x %% y	Remainder of (x divided by y)	`> 5 %% 3` `[1] 2`
Integer division	x%/%y	x divided by y but rounded down	`> 5%/%2` `[1] 2`
Computing the Square root	sqrt(x)	Computing the square root of x	`> sqrt(25)` `[1] 5`

Let us take a look at how R treats strings.

Strings

String values have to be enclosed within double quotes.

```
> "R is a statistical programming language"
[1] "R is a statistical programming language"
```

Operation	Example

paste	`> fname <- "seema"`
	`> lname <- "acharya"`
	`> paste(fname,lname)`
	`[1] "seema acharya"`
	
	Concatenates the vectors, “fname” and “lname”.

sprintf	`> sprintf("%s loves %s programming", "Sam", "r")`
	`[1] "Sam loves r programming"`
	Returns a character vector that contains a combination of text and variable values.

substr	`> substr("R is a statistical programming language", start = 20, stop =26)`
	`[1] "program"`
	
	Extracts the substring "program" which is at position, 20 and has 7 characters.

sub	`> sub("pizza","pasta", "I love pizza")`
	`[1] "I love pasta"`
	
	Substitues "pasta" for "pizza" in the string, "I love pizza".

Logical values

Logical values are TRUE and FALSE or T and F. Note that these are case sensitive. The equality operator is ==.

```
> 8 < 4
[1] FALSE

> 3 * 2 == 5
[1] FALSE

> 3 * 2 == 6
[1] TRUE

> F == FALSE
[1] TRUE

> T == TRUE
[1] TRUE
```

Test your understanding:

Step 1: x is a vector of 10 elements with values ranging from 1 to 10.

```
> x <- c(1:10)
```

Step 2: Display the content of the vector, x

```
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

Step 3: Print the values of those elements whose values are either greater than 7 or less than 5

Using the **OR** operator

```
> x[(x>7) | (x<5)]
[1]  1  2  3  4  8  9 10
```

Explanation:

We will do this in two parts:

Display the values of elements only if it is more than 7.

```
> x>7
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

Display the values of elements only if it is less than 5.

```
> x<5
 [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

Using the **AND** operator

```
> x[(x>7) & (x<10)]
[1] 8 9
```

Variables

Assign a value of 50 to the variable by the name, “Var”.

```
> Var <-50
```

Or

```
> Var=5
```

Print the value in the variable, “Var”.

```
> Var
[1] 50
```

Perform arithmetic operations on the variable, “Var”.

```
> Var + 10
[1] 60
> Var / 2
[1] 25
```

Variables can be reassigned values either of the same data type or of a different data type.

Reassign a string value to the variable, “Var”.

```
> Var <- "R is a Statistical Programming Language"
```

Print the value in the variable, “Var”.

```
> Var
[1] "R is a Statistical Programming Language"
```

Reassigning a logical value to the variable, “Var”.

```
> Var <- TRUE
```

```
> Var
[1] TRUE
```

Functions

In this section we will try out a few functions such as sum, min, max, seq, rep, grep, toupper, tolower and substr.

sum()

Description: This function returns the sum of all values in its arguments.

Syntax:

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

… → numeric or complex or logical vectors

na,rm → logical. Should missing values (including NaN(Not a Number)) be removed?

Example:

```
> sum(1,2,3)
[1] 6

> sum(1,5,NA, na.rm=FALSE)
[1] NA
```

If na.rm is FALSE, an NA or NaN value in any of the argument will cause NA or NaN to be returned.

```
> sum(1,5,NA, na.rm=FALSE)
[1] NA

> sum(1,5,NaN, na.rm=FALSE)
[1] NaN

> sum(1,5,NA,NaN,na.rm=FALSE)
[1] NA
```

If na.rm is TRUE, an NA or NaN value in any of the argument will be ignored.

```
> sum(1,5,NA, na.rm=TRUE)
[1] 6

> sum(1,5,NA,NaN,na.rm=TRUE)
[1] 6
```

min()

Description: this function returns the minimum of all the values present in their arguments.

Syntax:

min(…, na.rm=FALSE)

… →numeric or character arguments

na,rm → logical. Should missing values (including NaN) be removed?

Example:

```
> min(1,2,3)
[1] 1
```

If na.rm is FALSE, an NA or NaN value in any of the argument will cause NA or NaN to be returned.

```
> min(1,2,3,NA,na.rm=FALSE)
[1] NA

> min(1,2,3,NaN,na.rm=FALSE)
[1] NaN

> min(1,2,3,NA,NaN, na.rm=FALSE)
[1] NA
```

If na.rm is TRUE, an NA or NaN value in any of the argument will be ignored.

```
> min(1,2,3,NA, NaN, na.rm=TRUE)
[1] 1
```

max()

Description: this function returns the maximum of all the values present in their arguments.

Syntax:

max(…, na.rm=FALSE)

… →numeric or character arguments

na,rm → logical. Should missing values (including NaN) be removed?

Example:

```
> max(44,78,66)
[1] 78
```

If na.rm is FALSE, an NA or NaN value in any of the argument will cause NA or NaN to be returned.

```
> max(44,78,66,NA,na.rm=FALSE)
[1] NA

> max(44,78,66,NaN,na.rm=FALSE)
[1] NaN

> max(44,78,66,NA,NaN,na.rm=FALSE)
[1] NA
```

If na.rm is TRUE, an NA or NaN value in any of the argument will be ignored.

```
> max(44,78,66,NA,NaN,na.rm=TRUE)
[1] 78
```

seq()

Description: this function generates a regular sequence

Syntax:

seq(*start from* **,** *end at***,** *interval , length.out***)**

start from: the start value of the sequence.

End at: the maxim.al or end value of the sequence.

Interval: increment of the sequence

length.out: desired length of the sequence

Example:

```
> seq(1,10,2)
[1] 1 3 5 7 9

> seq(1,10,length.out=10)
 [1]  1  2  3  4  5  6  7  8  9 10

> seq(18)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
```

Or

```
> seq_len(18)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

> seq(1,6,by=3)
[1] 1 4
```

rep()

Description:

rep(): this function repeats a given argument, a specified number of times. In the below example, the string, “statistics” is repeated three times.

Example:

```
> rep("statistics",3)
[1] "statistics" "statistics" "statistics"
```

grep()

Description:

grep(): in the below example, the function grep finds the index position at which the string, " statistical" is

present.

Example:

```
> grep("statistical",c("R","is","a","statistical","language"),fixed=TRUE)
[1] 4
```

toupper()

Description: this function converts the given character vector to upper case.

Syntax: toupper(x)

x → is a character vector

Example:

```
> toupper("statistics")
[1] "STATISTICS"
```

Or

```
> casefold("r programming language",upper=TRUE)
[1] "R PROGRAMMING LANGUAGE"
```

tolower()

Description: this function converts the given character vector to lower case.

Syntax: tolower(x)

x → is a character vector

Example:

```
> tolower("STATISTICS")
[1] "statistics"
```

Or

```
> casefold("R PROGRAMMING LANGUAGE",upper=FALSE)
[1] "r programming language"
```

substr()

Description: Extract or replace substrings in a character vector

Syntax:

substr(x, start, stop)

x→ character vector

start → start position of extraction or replacement

stop → stop or end position of extraction or replacement

Example:

```
> substr("statistics",7,9)
[1] "tic"
```

Chapter 3: Vectors

A vector can have a list of values. The values can be numbers, strings or logical. All the values in the vector should be of the same data type.

A few points to remember about vectors in R.

- Vectors are stored like arrays in C
- Vector indices begin at 1
- All Vector elements must have the same mode such as integer, numeric (floating point number), character (string), logical (Boolean), complex, object etc.

Let us begin by creating a few vectors:

Objective: To create a vector of numbers

Act:

```
> c(4,7,8)
[1] 4 7 8
```

The c function (c is short for combine) creates a new vector consisting of three values: 4, 7, and 8.

Objective: To create a vector of string values

Act:

```
> c("R", "SAS", "SPSS")
[1] "R"    "SAS"  "SPSS"
```

Objective: To create a vector of logical values

Act:

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

A vector cannot hold values of different data types. Consider the example below. We are trying to place, integer, string and boolean values together in a vector.

```
> c(4,8,"R",FALSE)
[1] "4"     "8"     "R"     "FALSE"
```

Note: All the values are converted to the same data type, i.e. “character”.

Objective: To declare a vector by the name “Project” of length, 3 and store values in it.

Act:

```
> Project <- vector(length = 3)
> Project[1] <- "Finance Project"
> Project[2] <- "Retail Project"
> Project[3] <- "Energy Project"
```

Outcome:

```
> Project
[1] "Finance Project" "Retail Project"  "Energy Project"
> length(Project)
[1] 3
```

Sequence Vector

A sequence vector can be created with start:end notation.

Objective: To create a sequence of numbers between 1 and 5 (both inclusive).

Act:

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

Or

```
> seq(1:5)
[1] 1 2 3 4 5
```

The default increment with seq is 1. However it also allows the use of increments other than 1.

```
> seq(1,10,2)
[1] 1 3 5 7 9
```

or

```
> seq(from=1, to=10, by=2)
[1] 1 3 5 7 9
```

Or

```
> seq(1, 10, by=2)
[1] 1 3 5 7 9
```

seq can also generate numbers in descending order.

```
> 10:1
 [1] 10  9  8  7  6  5  4  3  2  1
```

```
> seq(10,1,by=-2)
[1] 10  8  6  4  2
```

rep

The rep function is used to place the same constant into long vectors. The syntax is rep (z,k) which creates a vector of k*length(z) elements, each equal to z.

Objective: To demonstrate rep function.

Act:

```
> rep(3,4)
[1] 3 3 3 3
```

Or

```
> x <-rep(3,4)
> x
[1] 3 3 3 3
```

Vector Access

Objective: Let us create a variable by the name, "VariableSeq" and assign to it a vector consisting of string values.

Act:

```
> VariableSeq <- c("R", "is","a","programming","language")
```

Objective: To access values in a vector, specify the indices at which the value is present in the vector. Indices start at 1.

Act:

```
> VariableSeq[1]
[1] "R"
> VariableSeq[2]
[1] "is"
> VariableSeq[3]
[1] "a"
> VariableSeq[4]
[1] "programming"
> VariableSeq[5]
[1] "language"
```

Objective: To assign new values in an existing vector. For example, let us assign value, "good programming" at indices 4 in the existing vector, "VariableSeq".

Act:

```
> VariableSeq[4] <- "good programming"
```

Outcome:

```
> VariableSeq[4]
[1] "good programming"
```

Objective: To access more than one value from the vector.

Act:

```
> VariableSeq[c(1,5)]
[1] "R"        "language"

> VariableSeq[1:4]
[1] "R"                "is"                "a"                 "good programming"

> VariableSeq[c(1,4:5)]
[1] "R"                 "good programming" "language"
```

To get all the values in the variable, "VariableSeq"

```
> VariableSeq
[1] "R"                "is"                "a"                 "good programming"
[5] "language"
```

Vector Names

The names function helps to assign names to the vector elements.

This is accomplished in 2 steps:

```
> placeholder <- 1:5
> names(placeholder) <- c("r", "is", "a", "programming", "language")
```

The vector elements can then be retrieved using the indices position.

```
> placeholder
          r          is           a programming    language
          1           2           3           4           5

> placeholder[3]
a
3
> placeholder [1]
r
1
> placeholder[4:5]
programming    language
          4           5
```

Or

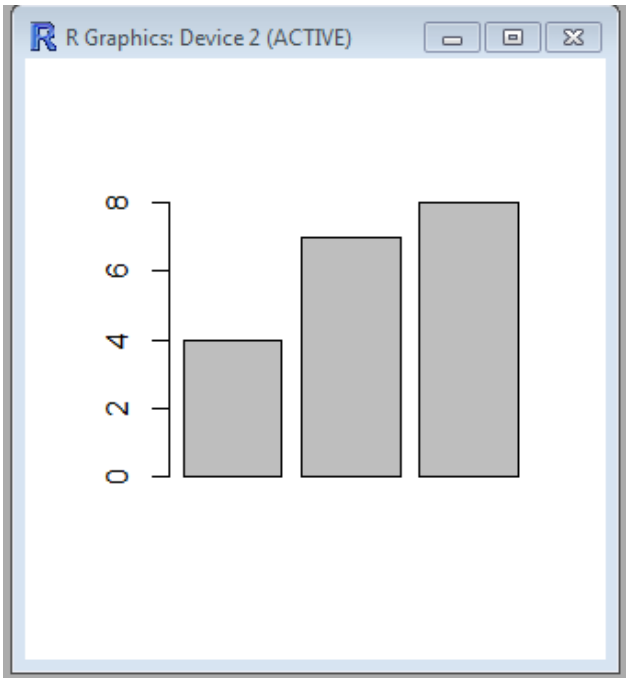
```
> placeholder["programming"]
programming
          4
```

Objective: To plot a bar graph using the barplot function. The barplot function uses a vector's values to plot a bar chart.

Act: The vector used is called BarVector.

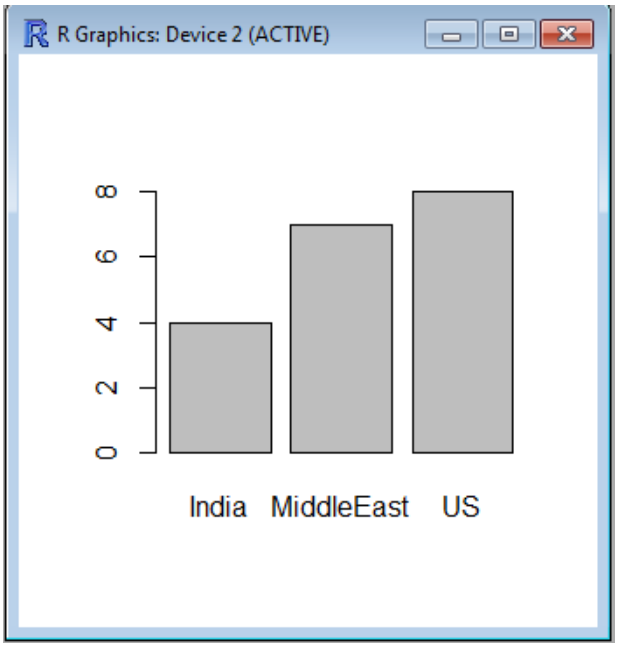
```
> BarVector <- c(4,7,8)
> barplot(BarVector)
```

Outcome:



Let us use the name function to assign names to the vector elements. These names will be used as labels in the barplot.

```
> names(BarVector) <- c("India", "MiddleEast", "US")
> barplot(BarVector)
```



Vector Math

Let us define a vector x with three values. Let us add a scalar value (single value) to the vector. This value will get added to each vector element

```
> x <- c(4,7,8)
> x +1
[1] 5 8 9
```

However the vector x will retain its individual elements.

```
> x
[1] 4 7 8
```

If the vector needs to be updated with the new values, type in the below statement.

```
> x <- x + 1
> x
[1] 5 8 9
```

We can run other arithmetic operations on the vectors as follows:

```
> x - 1
[1] 4 7 8
> x * 2
[1] 10 16 18
> x / 2
[1] 2.5 4.0 4.5
```

Let us practice these arithmetic operations on two vectors

```
> x
[1] 5 8 9
> y <- c(1,2,3)
> y
[1] 1 2 3
> x + y
[1]  6 10 12
```

Other arithmetic operations

```
> x - y
[1] 4 6 6
> x * y
[1]  5 16 27
```

Check if the two vectors are equal. The comparison takes place element by element.

```
> x
[1] 5 8 9
> y
[1] 1 2 3
> x == y
[1] FALSE FALSE FALSE
```

```
> x < y
[1] FALSE FALSE FALSE
```

```
> sin(x)
[1] -0.9589243  0.9893582  0.4121185
```

Vector recycling

If an operation is performed involving two vectors that requires them to be of the same length, the shorter one is recycled i.e. repeated until it is long enough to match the longer one.

Objective: To add two vectors wherein one has length 3 and the other has length 6.

Act:

```
> c(1,2,3) + c(4,5,6,7,8,9)
[1]  5  7  9  8 10 12
```

Objective: To multiply two vectors wherein one has length 3 and the other has length 6.

Act:

```
> c(1,2,3) * c(4,5,6,7,8,9)
[1]  4 10 18  7 16 27
```

Objective: To understand Scatter Plot. The function to use is plot. This function uses two vector, one for the x axis and another for the y axis. The objective is to understand the relationship between numbers and their sines. We will use two vectors: Vector x which will have a sequence of values between 1 and 25 at an interval of 0.1. Vector y stores the sines of all values held in vector x.

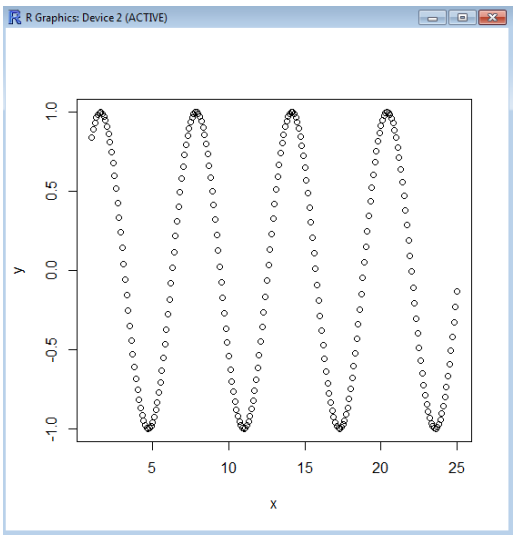
Act:

The plot function takes the values in vector x and plots it on the horizontal axis. It then takes the values in vector y and places it on the vertical axis.

```
> x <-seq(1,25,0.1)
```

```
> y <-sin(x)
```

```
> plot(x,y)
```


R Graphics: Device 2 (ACTIVE)
y
1.0
0.5
0.0
-0.5
-1.0
5
10
15
20
25
x

Chapter 4: Matrices

Matrices are nothing but 2-dimensional arrays.

Objective: Let us make a matrix which is 3 rows by 4 columns and set all its elements to 1

Act:

```
> matrix(1,3,4)
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    1    1    1    1
[3,]    1    1    1    1
```

Objective: To use a vector to create an array, 3 rows high and 3 columns wide.

Act:

Step 1: Begin by creating a vector that has elements from 10 to 90 with an interval of 10

```
> a <- seq(10,90, by = 10)
```

Step 2: Validate by printing the value of vector a

```
> a
[1] 10 20 30 40 50 60 70 80 90
```

Step 3: Call the matrix function with the vector a, the number of rows and the number of columns.

```
> matrix (a,3,3)
     [,1] [,2] [,3]
[1,]   10   40   70
[2,]   20   50   80
[3,]   30   60   90
```

Objective: To re-shape the vector itself into an array using the dim function

Act:

Step 1: Begin by creating a vector that has elements from 10 to 90 with an interval of 10

```
> a <- seq(10,90, by = 10)
```

Step 2: Validate by printing the value of vector a

```
> a
[1] 10 20 30 40 50 60 70 80 90
```

Step 3: Assign new dimensions to vector a by passing a vector having 3 rows and 3 columns (c (3, 3)).

```
> dim(a) <- c(3,3)
```

Step 4: Print the values of the vector a. You will notice that the values have shifted to form 3 rows by 3 columns. The vector is no longer 1 dimensional. It has been converted into a 2 dimensional matrix, 3 rows high and 3 columns wide.

```
> a
     [,1] [,2] [,3]
[1,]   10   40   70
[2,]   20   50   80
[3,]   30   60   90
```

Matrix access

Objective: To access the element of a 3 *4 matrix.

Act:

Step 1: create a matrix, "mat", 3 rows high and 4 columns wide using a vector

```
> x <- 1:12

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

> mat <- matrix(x,3,4)

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

Step 2: Access the element present in the 2nd row and 3rd column of the matrix, "mat".

```
> mat[2,3]
[1] 8
```

Objective: To access the 3rd row of an existing matrix.

Act:

Step 1: let us begin by printing the values of an existing matrix, "mat"

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

Step 2:

To access the 3rd row of the matrix, simply provide the row number and omit the column number.

```
> mat[3,]
[1]  3  6  9 12
```

Objective: To access the 2nd column of an existing matrix.

Act:

Step 1: let us begin by printing the values of an existing matrix, "mat"

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

Step 2:

To access the 2nd column of the matrix, simply provide the column number and omit the row number.

```
> mat[,2]
[1] 4 5 6
```

Objective: To access the 2nd and 3rd column of an existing matrix.

Act:

Step 1: let us begin by printing the values of an existing matrix, "mat"

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

Step 2:

To access the 2nd and 3rd column of the matrix, simply provide the column numbers and omit the row number.

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

Objective:

Act:

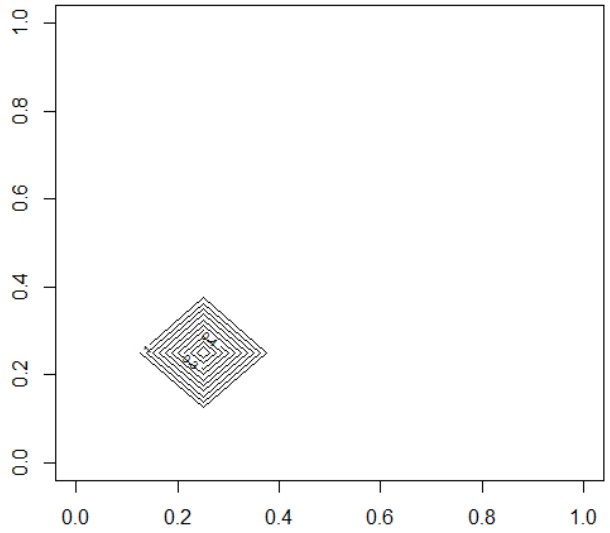
```
> mat <- matrix(1,9,9)
```

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

```
> mat[3,3] <-0
```

```
> mat
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    1    1    1    1    1    1    1    1    1
 [2,]    1    1    1    1    1    1    1    1    1
 [3,]    1    1    0    1    1    1    1    1    1
 [4,]    1    1    1    1    1    1    1    1    1
 [5,]    1    1    1    1    1    1    1    1    1
 [6,]    1    1    1    1    1    1    1    1    1
 [7,]    1    1    1    1    1    1    1    1    1
 [8,]    1    1    1    1    1    1    1    1    1
 [9,]    1    1    1    1    1    1    1    1    1
```

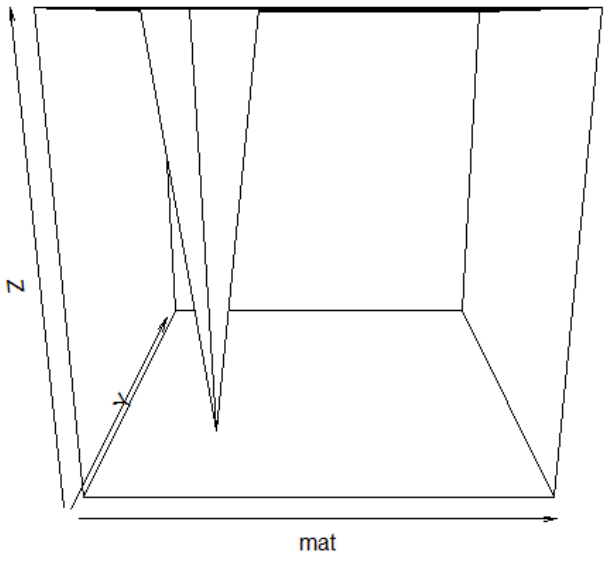
```
> contour(mat)
```



Objective: To create a 3 D perspective plot with the persp function.

Act:

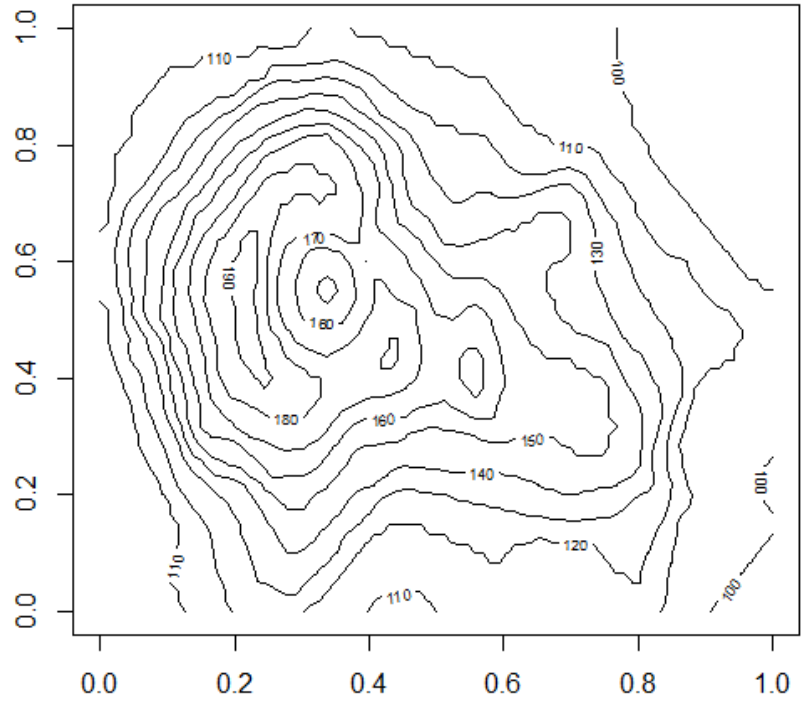
```
> persp(mat)
```



Objective: R includes some sample data sets. One of these is volcano, a 3D map of a dormant New Zealand volcano.

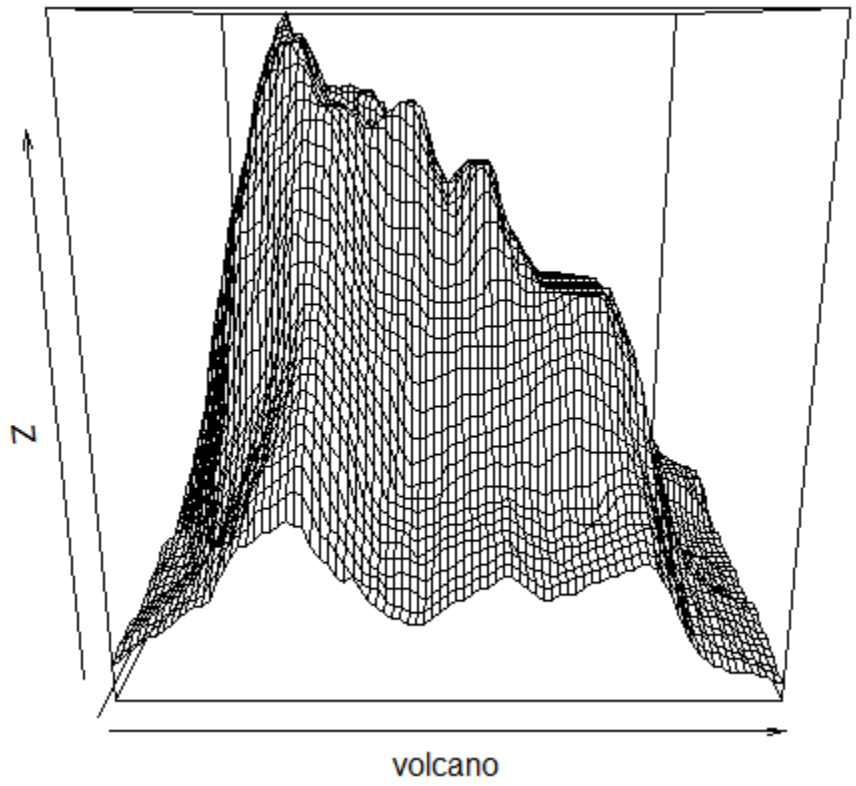
Let us create a contour map of volcano data set

```
> contour(volcano)
```



Let us create a 3 D perspective map of the sample data set, “volcano”.

```
> persp(volcano)
```

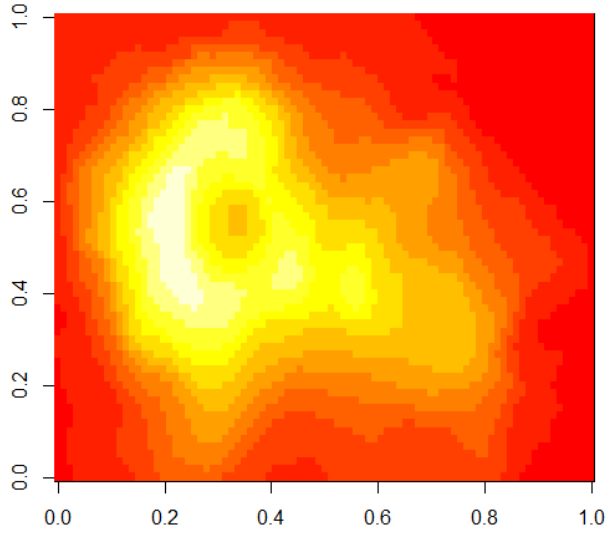


Objective:

To create a heat map of the sample data set, “volcano”.

Act:

```
> image(volcano)
```



Chapter 5: Factors

Creating factors

School, "XYZ", places students in groups also called as houses. Each group is assigned a unique color such as "red", "green", "blue" or "yellow". HouseColor, is a vector that stores the house colors of a group of students.

```
> HouseColor <- c('red','green','blue','yellow','red','green','blue','blue')

> types <- factor(HouseColor)

 > HouseColor
 [1] "red"    "green"  "blue"   "yellow" "red"    "green"  "blue"   "blue"

 > print(HouseColor)
 [1] "red"    "green"  "blue"   "yellow" "red"    "green"  "blue"   "blue"

> print(types)
[1] red    green  blue   yellow red    green  blue   blue
Levels: blue green red yellow
```

Levels denotes the unique values. The above has four distinct values such as "blue", "green", "red" and "yellow".

```
> as.integer(types)
[1] 3 2 1 4 3 2 1 1
```

The above output is explained as below:

1 is the number assigned to blue

2 is the number assigned to green

3 is the number assigned to red

And 4 is the number assigned to yellow.

```
> levels(types)
[1] "blue"   "green"  "red"    "yellow"
```

The vector "NoofStudents" stores the number of students in each house/group with 12 students in blue house, 14 students in green house, 12 students in red house and 13 students in yellow house.

```
> NoofStudents <- c(12,14,12,13)

> NoofStudents
[1] 12 14 12 13
```

The vector, "AverageScore" stores the average score of the students of each house/group i.e 70 is the average score for students of the blue house, 80 is the average score for students of the green house, 90 is

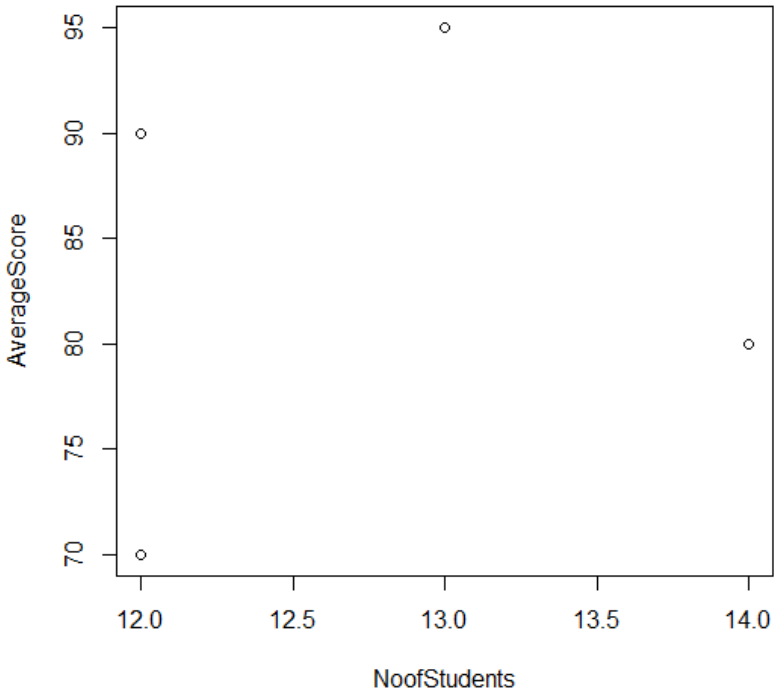
the average score for the students of the red house and 95 is the average score for the students of the yellow house.

```
> AverageScore(70,80,90,95)

> AverageScore
[1] 70 80 90 95
```

Objective: To plot the relationship between NoofStudents and AverageScore.

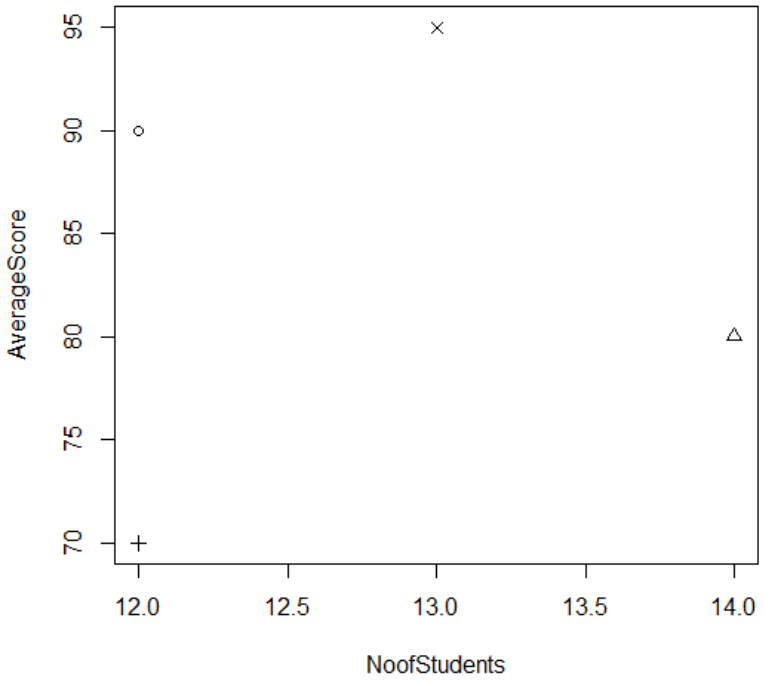
Act:

```
> plot(NoofStudents, AverageScore)
```



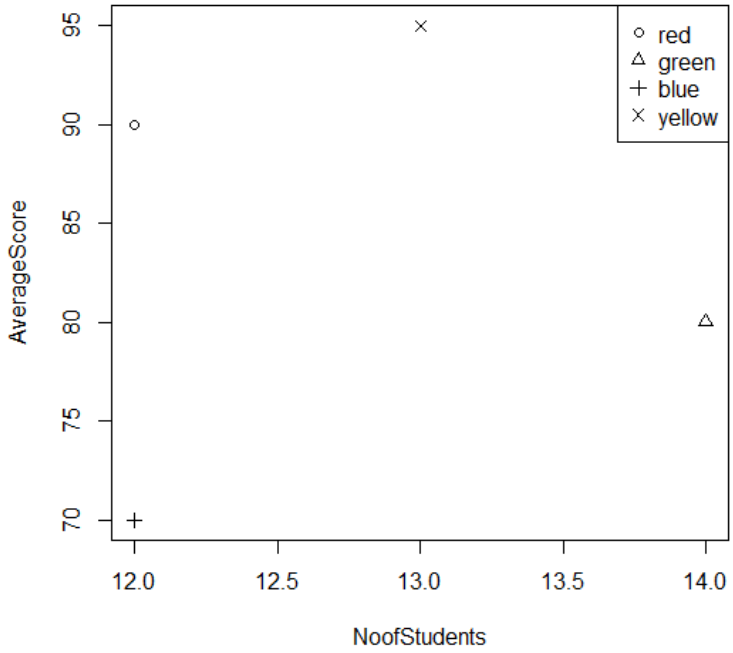

```
> plot(NoofStudents, AverageScore, pch=as.integer(types))
```

The above graph displays 4 dots. Let us improve the graph by at least using different symbols to represent each house.



To add further meaning to the graph, let us place a legend on the top right corner.

```
> legend("topright",c("red","green","blue","yellow"),pch=1:4)
```



Chapter 6: List

It is similar to C Struct.

Objective: To create a list in R.

Act:

To create a list, “emp” having three elements, “EmpName”, “EmpUnit”, “EmpSal”.

```
> emp <- list("EmpName="Alex", EmpUnit = "IT", EmpSal = 55000)
```

Outcome:

To get the elements of the list, “emp” use the below command.

```
> emp
$EmpName
[1] "Alex"

$EmpUnit
[1] "IT"

$EmpSal
[1] 55000
```

Actually, the element names, e.g. “EmpName”, “EmpUnit” and “EmpSal” are optional. We could alternatively do this as follows:

```
> EmpList <- list("Alex", "IT", 55000)

> EmpList
[[1]]
[1] "Alex"

[[2]]
[1] "IT"

[[3]]
[1] 55000
```

Note: Here the elements of EmpList are referred to as 1,2,3.

List Tags and Values

A list has elements. The elements in a list can have names which is referred to as tags and the elements can have values.

For example in the “emp” list we have three elements: EmpName, EmpUnit and EmpSal. The values are as follows. The element “EmpName” has the value “Alex”, the element “EmpUnit” has the value “IT” and the element “EmpSal” has the value 55000.

Let us look at the command to retrieve the names and values of the elements in a list.

Objective: To retrieve the names of the elements in the list "emp".

Act:

```
> names(emp)
[1] "EmpName" "EmpUnit" "EmpSal"
```

Objective: To retrieve the values of the elements in the list "emp".

Act:

```
> unlist(emp)
EmpName EmpUnit  EmpSal
 "Alex"    "IT" "55000"
```

If one wishes to retrieve the value of a single element in the list "emp", here is the command :

Objective: To retrieve the value of the element "EmpName" in the list "emp".

Act:

```
> unlist(emp["EmpName"])
EmpName
 "Alex"
```

Likewise one can check the value of the other elements in the list.

```
> unlist(emp["EmpUnit"])
EmpUnit
   "IT"
> unlist(emp["EmpSal"])
EmpSal
 55000
```

There is yet another way to retrieve the values of the elements in the list "emp". Here is a look at them.

Objective: To retrieve the value of the element "EmpName" in the list "emp".

Act:

```
> emp[["EmpName"]]
[1] "Alex"
```

Or

```
> emp[[1]]
[1] "Alex"
```

Add / Delete element to/from a list

Before adding an element to the list "emp", let us verify what elements exist in the list.

```
> emp
$EmpName
[1] "Alex"

$EmpUnit
[1] "IT"

$EmpSal
[1] 55000
```

Objective: To add an element with the name "EmpDesg" and value "Software Engineer" to the list, "emp".

Act:

```
> emp$EmpDesg = "Software Engineer"
```

Outcome:

```
> emp
$EmpName
[1] "Alex"

$EmpUnit
[1] "IT"

$EmpSal
[1] 55000

$EmpDesg
[1] "Software Engineer"
```

Objective: To delete an element with the name "EmpUnit" and value "IT" from the list, "emp".

Act:

```
> emp$EmpUnit <- NULL
```

Outcome:

```
> emp
$EmpName
[1] "Alex"

$EmpSal
[1] 55000

$EmpDesg
[1] "Software Engineer"
```

Size of a list

Length() function can be used to determine the number of elements present in the list.

The list “emp” has three elements as shown below:

```
> emp
$EmpName
[1] "Alex"

$EmpSal
[1] 55000

$EmpDesg
[1] "Software Engineer"
```

Objective: To determine the number of elements in the list “emp”.

Act:

```
> length(emp)
[1] 3
```

Recursive list

A recursive list means a list within a list.

Objective: To create a list within a list.

Act:

Let us begin with two lists, “emp” and “emp1”.

The elements in both the lists are as shown below:

```
> emp
$EmpName
[1] "Alex"

$EmpSal
[1] 55000

$EmpDesg
[1] "Software Engineer"

> emp1
$EmpUnit
[1] "IT"

$EmpCity
[1] "Los Angeles"
```

We would like to combine both the lists into a single list by the name “EmpList”.

```
> EmpList <- list(emp,emp1)
```

Outcome:

```
> EmpList
[[1]]
[[1]]$EmpName
[1] "Alex"

[[1]]$EmpSal
[1] 55000

[[1]]$EmpDesg
[1] "Software Engineer"


[[2]]
[[2]]$EmpUnit
[1] "IT"

[[2]]$EmpCity
[1] "Los Angeles"
```

Chapter 7: Data Frames

You can think of a data frame as something akin to a database table or an Excel spreadsheet. It has a specific number of columns, each of which is expected to contain values of a particular type. It also has an indeterminate number of rows - sets of related values for each column.

Let us assume that we have three vectors namely, “EmpNo”, “EmpName” and “ProjName” storing details such as employee nos, employee names and project names respectively.

```
> EmpNo <- c(1000,1001,1002,1003,1004)
> EmpName <- c("Jack", "Jane", "Margaritta", "Joe", "Dave")
> ProjName <- c("P01","P02","P03","P04","P05")
```

We need a data structure similar to a database table or an Excel spreadsheet that can bind all these details together. We create a data frame by the name, “Employee” to store all the three vectors together.

```
> Employee <- data.frame(EmpNo,EmpName,ProjName)
```

Let us print the content of the date frame, “Employee”.

```
> Employee
  EmpNo    EmpName ProjName
1  1000       Jack      P01
2  1001       Jane      P02
3  1002 Margaritta      P03
4  1003        Joe      P04
5  1004       Dave      P05
```

We have just created a data frame, “employee” with data neatly organized into rows and the variable names serving as column names across the top.

Data Frame Access

There are two ways to access the content of data frames.

1. **By providing the index number in double brackets.**

Example:

To access the second column, “EmpName”, we type in the following command at the R prompt.

```
> Employee[2]
     EmpName
1       Jack
2       Jane
3 Margaritta
4        Joe
5       Dave
```

Example:

To access the first and the second column, "EmpNo" and "EmpName", we type in the following command at the R prompt.

```
> Employee[1:2]
  EmpNo     EmpName
1  1000        Jack
2  1001        Jane
3  1002 Margaritta
4  1003         Joe
5  1004        Dave
```

2. By providing the column name as a string in double brackets.

```
> Employee[["EmpName"]]
[1] Jack       Jane       Margaritta Joe        Dave
Levels: Dave Jack Jane Joe Margaritta
```

Just to keep it simple (typing so many double brackets can get unwieldy at times], use the below notation with the $ (dollar) sign

```
> Employee$EmpName
[1] Jack       Jane       Margaritta Joe        Dave
Levels: Dave Jack Jane Joe Margaritta
```

Load Data frames

Let us look at how R can load data from external files.

Reading from a .csv (comma separated values file)

Example:

We have created a .csv file by the name, "item.csv" in the D:\ drive. It has the following content.

	A	B	C
1	**Itemcode**	**ItemCategory**	**ItemPrice**
2	I1001	Electornics	700
3	I1002	Desktop supplies	300
4	I1003	Office supplies	350

Let us load this file using the read.csv function.

```
> read.csv("d:/item.csv")
  Itemcode      ItemCategory ItemPrice
1    I1001       Electornics       700
2    I1002 Desktop supplies       300
3    I1003  Office supplies       350
```

Reading from a tab separated value file.

For files using a delimiter other than comma, one can use the read.table command.

Example:

We have created a tab separated file by the name, “item-tab-sep.txt” in the D:\ drive. It has the following content.

```
Itemcode        ItemQtyOnHand   ItemReorderLvl
I1001   75      25
I1002   30      25
I1003   35      25
```

Let us load this file using the read.table function.

```
> read.table("d:/item-tab-sep.txt",sep="\t")
        V1            V2            V3
1 Itemcode ItemQtyOnHand ItemReorderLvl
2    I1001            70            25
3    I1002            30            25
4    I1003            35            25
```

Notice the use of V1, V2 and V3 as column headings. It means that our specified column names, “Itemcode”, ItemCategory” and “ItemPrice” are not considered. In other words, the first line is not automatically treated as column headers.

Let us modify the syntax to have the first line treated as column headers.

```
> read.table("d:/item-tab-sep.txt",sep="\t", header=TRUE)
  Itemcode ItemQtyOnHand ItemReorderLvl
1    I1001            70            25
2    I1002            30            25
3    I1003            35            25
```

Merging data frames

Let us now attempt to merge two data frames using the merge function.

The merge function takes arguments with an x frame (item.csv) and a y frame (item-tab-sep.txt). By default, it joins the frames on columns with the same name (the two “Itemcode” columns).

```
> csvitem <- read.csv("d:/item.csv")

> tabitem <- read.table("d:/item-tab-sep.txt",sep="\t",header=TRUE)

> merge(x=csvitem, y=tabitem)
  Itemcode     ItemCategory ItemPrice ItemQtyOnHand ItemReorderLvl
1    I1001      Electornics       700            70            25
2    I1002 Desktop supplies       300            30            25
3    I1003  Office supplies       350            35            25
```

Section 2: R Data Interfaces

R Data Interfaces

This section deals with reading the data from a CSV (Comma separated value) file, an XML (extensible markup language) file and a JSON (Java Script Object Notation) document.

Read the .csv file

Objective: To read the data from a .csv file (D:\SampleSuperstore.csv) into a data frame. The data should then be grouped by Category. The column on which grouping is done is "Sales". The aggregate function to be used is "sum".

Step 1:

The data is stored in "D:\SampleSuperstore.csv". Data is available under the following columns.

Row ID, Order ID, Order Date, Ship Date, Ship Mode, Customer ID, Customer Name, Segment, Country, State, City, Postal Code, Region, Product ID, Category, Sub-Category, Product Name, Sales, Quantity, Discount, Price.

A subset of the data is shown here:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	Row ID	Order ID	Order Date	Ship Date	Ship Mode	Customer	Customer	Segment	Country	City	State	Postal Coc	Region	Product ID	Category	Sub-Categ	Product N	Sales	Quant	Discou	Profit
2	1	CA-2013-1	11/9/2013	11/12/2013	Second Cl	CG-12520	Claire Gut	Consumer	United Sta	Henderso	Kentucky	42420	South	FUR-BO-1(	Furniture	Bookcase:	Bush Som	261.96	2	0	41.91
3	2	CA-2013-1	11/9/2013	11/12/2013	Second Cl	CG-12520	Claire Gut	Consumer	United Sta	Henderso	Kentucky	42420	South	FUR-CH-1(	Furniture	Chairs	Hon Delu>	731.94	3	0	219.6
4	3	CA-2013-1	6/13/2013	6/17/2013	Second Cl	DV-13045	Darrin Van	Corporate	United Sta	Los Angel	California	90036	West	OFF-LA-1(	Office Sup	Labels	Self-Adhe	14.62	2	0	6.871
5	4	US-2012-1	10/11/2012	10/18/2012	Standard (	SO-20335	Sean O'Dc	Consumer	United Sta	Fort Laude	Florida	33311	South	FUR-TA-1(	Furniture	Tables	Bretford C	957.5775	5	0.45	-383
6	5	US-2012-1	10/11/2012	10/18/2012	Standard (	SO-20335	Sean O'Dc	Consumer	United Sta	Fort Laude	Florida	33311	South	OFF-ST-10	Office Sup	Storage	Eldon Fol(	22.368	2	0.2	2.516
7	6	CA-2011-1	6/9/2011	6/14/2011	Standard (	BH-11710	Brosina Ho	Consumer	United Sta	Los Angel	California	90032	West	FUR-FU-1(	Furniture	Furnishing	Eldon Exp	48.86	7	0	14.17
8	7	CA-2011-1	6/9/2011	6/14/2011	Standard (	BH-11710	Brosina Ho	Consumer	United Sta	Los Angel	California	90032	West	OFF-AR-1(	Office Sup	Art	Newell 32	7.28	4	0	1.966
9	8	CA-2011-1	6/9/2011	6/14/2011	Standard (	BH-11710	Brosina Ho	Consumer	United Sta	Los Angel	California	90032	West	TEC-PH-1(	Technolog	Phones	Mitel 532(	907.152	6	0.2	90.72
10	9	CA-2011-1	6/9/2011	6/14/2011	Standard (	BH-11710	Brosina Ho	Consumer	United Sta	Los Angel	California	90032	West	OFF-BI-10	Office Sup	Binders	DXL Angle	18.504	3	0.2	5.783
11	10	CA-2011-1	6/9/2011	6/14/2011	Standard (	BH-11710	Brosina Ho	Consumer	United Sta	Los Angel	California	90032	West	OFF-AP-1(	Office Sup	Appliance	Belkin F5(	114.9	5	0	34.47
12	11	CA-2011-1	6/9/2011	6/14/2011	Standard (	BH-11710	Brosina Ho	Consumer	United Sta	Los Angel	California	90032	West	FUR-TA-1(	Furniture	Tables	Chromcra	1706.184	9	0.2	85.31
13	12	CA-2011-1	6/9/2011	6/14/2011	Standard (	BH-11710	Brosina Ho	Consumer	United Sta	Los Angel	California	90032	West	TEC-PH-1(	Technolog	Phones	Konftel 25	911.424	4	0.2	68.36
14	13	CA-2014-1	4/16/2014	4/21/2014	Standard (	AA-10480	Andrew A	Consumer	United Sta	Concord	North Car	28027	South	OFF-PA-1(	Office Sup	Paper	Xerox 196	15.552	3	0.2	5.443
15	14	CA-2013-1	12/6/2013	12/11/2013	Standard (	IM-15070	Irene Mac	Consumer	United Sta	Seattle	Washingt(	98103	West	OFF-BI-10	Office Sup	Binders	Fellowes	407.976	3	0.2	132.6
16	15	US-2012-1	11/22/2012	11/26/2012	Standard (	HP-14815	Harold Pa	Home Off	United Sta	Fort Wort	Texas	76106	Central	OFF-AP-1(	Office Sup	Appliance	Holmes Re	68.81	5	0.8	-123.9
17	16	US-2012-1	11/22/2012	11/26/2012	Standard (	HP-14815	Harold Pa	Home Off	United Sta	Fort Wort	Texas	76106	Central	OFF-BI-10	Office Sup	Binders	Storex Du	2.544	3	0.8	-3.816
18	17	CA-2011-1	11/11/2011	11/18/2011	Standard (	PK-19075	Pete Kriz	Consumer	United Sta	Madison	Wisconsin	53711	Central	OFF-ST-10	Office Sup	Storage	Stur-D-Stc	665.88	6	0	13.32
19	18	CA-2011-1	5/13/2011	5/15/2011	Second Cl	AG-10270	Alejandro	Consumer	United Sta	West Jord	Utah	84084	West	OFF-ST-10	Office Sup	Storage	Fellowes	55.5	2	0	9.99
20	19	CA-2011-1	8/27/2011	9/1/2011	Second Cl	ZD-21925	Zuschuss I	Consumer	United Sta	San Franci	California	94109	West	OFF-AR-1(	Office Sup	Art	Newell 34	8.56	2	0	2.482

With the use of read.csv function, data is read from "D:\SampleSuperstore.csv" file and stored in the data frame named, "InputData".

```
> InputData <- read.csv("d:/SampleSuperstore.csv")
```

Step 2:

Data is grouped / aggregated on InputData$Sales by InputData$Category. The aggregation function used is "sum". InputData$Sales refers to the "Sales" column of the data frame, "InputData". Likewise InputData$Category refers to the "Category" column of the data frame, "InputData".

```
> GroupedInputData <- aggregate(InputData$Sales ~ InputData$Category,InputData,sum)
```

Display the aggregated data. As evident from the below display, data is available in three categories: "Furniture", "Office Supplies", and "Technology".

```
> GroupedInputData
  InputData$Category InputData$Sales
1          Furniture        156514.4
2    Office Supplies        132600.8
3         Technology        168638.0
```

Reading a JSON (Java Script Object Notation) document

Step 1:

Install rjson package.

```
> install.packages("rjson")
Installing package into 'C:/Users/seema_acharya/Documents/R/win-library/3.2'
(as 'lib' is unspecified)
trying URL 'https://cran.hafro.is/bin/windows/contrib/3.2/rjson_0.2.15.zip'
Content type 'application/zip' length 493614 bytes (482 KB)
downloaded 482 KB

package 'rjson' successfully unpacked and MD5 sums checked
```

Step 2:

Input data

Store the below data in a text file ("D:/Jsondoc.json"). Please ensure the file is saved with an extension of .json

```
{
  "EMPID":["1001","2001","3001","4001","5001","6001","7001","8001" ],
  "Name":["Ricky","Danny","Mitchelle","Ryan","Gerry","Nonita","Simon","Gallop" ],
  "Dept": ["IT","Operations","IT","HR","Finance","IT","Operations","Finance"]
}
```

A JSON document begins and ends with a curly brace ({}). A JSON document is a set of key value pairs. Each key:value pair is delimited using "," as a delimiter.

Step 3:

Read the JSON file, "d:/Jsondoc.json".

```
> output <- fromJSON(file = "d:/Jsondoc.json")
```

```
> output
$EMPID
[1] "1001" "2001" "3001" "4001" "5001" "6001" "7001" "8001"

$Name
[1] "Ricky"     "Danny"     "Mitchelle" "Ryan"      "Gerry"     "Nonita"
[7] "Simon"     "Gallop"

$Dept
[1] "IT"         "Operations" "IT"         "HR"         "Finance"
[6] "IT"         "Operations" "Finance"
```

Step 4:

Convert JSON to a Data Frame

```
> JSONDataFrame <- as.data.frame(output)
```

Display the content of the data frame, "output".

```
> JSONDataFrame
  EMPID      Name       Dept
1  1001     Ricky         IT
2  2001     Danny Operations
3  3001 Mitchelle         IT
4  4001      Ryan         HR
5  5001     Gerry    Finance
6  6001    Nonita         IT
7  7001     Simon Operations
8  8001    Gallop    Finance
```

Reading an XML file

Step 1:

Install XML package

```
> install.packages("XML")
Installing package into 'C:/Users/seema_acharya/Documents/R/win-library/3.2'
(as 'lib' is unspecified)
trying URL 'https://cran.hafro.is/bin/windows/contrib/3.2/XML_3.98-1.3.zip'
Content type 'application/zip' length 4299803 bytes (4.1 MB)
downloaded 4.1 MB

package 'XML' successfully unpacked and MD5 sums checked
```

Step 2:

Input data

Store the below data in a text file (XMLFile.xml in the D: drive). Please ensure the file is saved with an extension of .xml

```
<RECORDS>
  <EMPLOYEE>
    <EMPID>1001</EMPID>
    <EMPNAME>Merrilyn</EMPNAME>
    <SKILLS>MongoDB</SKILLS>
    <DEPT>Computer Science</DEPT>
  </EMPLOYEE>

  <EMPLOYEE>
    <EMPID>1002</EMPID>
    <EMPNAME>Ramya</EMPNAME>
    <SKILLS>People Management</SKILLS>
    <DEPT>Human Resources</DEPT>
  </EMPLOYEE>

  <EMPLOYEE>
    <EMPID>1003</EMPID>
    <EMPNAME>Fedora</EMPNAME>
    <SKILLS>Recruitment</SKILLS>
    <DEPT>Human Resources</DEPT>
  </EMPLOYEE>
</RECORDS>
```

Reading XML File

The xml file is read in R using the function **xmlParse()**. It is stored as a list in R.

Step 1

Begin by loading the required packages.

```
> library("XML")
Warning message:
package 'XML' was built under R version 3.2.3
> library("methods")

> output <- xmlParse(file = "d:/XMLFile.xml")


> print(output)
<?xml version="1.0"?>
<RECORDS>
  <EMPLOYEE>
    <EMPID>1001</EMPID>
    <EMPNAME>Merrilyn</EMPNAME>
    <SKILLS>MongoDB</SKILLS>
    <DEPT>ComputerScience</DEPT>
  </EMPLOYEE>
  <EMPLOYEE>
    <EMPID>1002</EMPID>
    <EMPNAME>Ramya</EMPNAME>
    <SKILLS>PeopleManagement</SKILLS>
    <DEPT>HumanResources</DEPT>
  </EMPLOYEE>
  <EMPLOYEE>
    <EMPID>1003</EMPID>
    <EMPNAME>Fedora</EMPNAME>
    <SKILLS>Recruitment</SKILLS>
    <DEPT>HumanResources</DEPT>
  </EMPLOYEE>
</RECORDS>
```

Step 2

Extract the root node from the XML file

```
> rootnode <- xmlRoot(output)
```

Find the number of nodes in the root

```
> rootsize <- xmlSize(rootnode)

> rootsize
[1] 3
```

Let us display the details of the first node .

```
> print(rootnode[1])
$EMPLOYEE
<EMPLOYEE>
  <EMPID>1001</EMPID>
  <EMPNAME>Merrilyn</EMPNAME>
  <SKILLS>MongoDB</SKILLS>
  <DEPT>ComputerScience</DEPT>
</EMPLOYEE>

attr(,"class")
[1] "XMLInternalNodeList" "XMLNodeList"
```

Let us display the details of the first element of the first node.

```
> print(rootnode[[1]][[1]])
<EMPID>1001</EMPID>
```

Let us display the details of the third element of the first node.

```
> print(rootnode[[1]][[3]]
<SKILLS>MongoDB</SKILLS>
```

Next, display the details of the third element of the second node,

```
> print(rootnode[[2]][[3]])
<SKILLS>PeopleManagement</SKILLS> \
```

One can also display the value of the 2nd element of the first node.

```
> output <- xmlValue(rootnode[[1]][[2]])
> output
[1] "Merrilyn"
```

Step 3

Convert input xml file to data frame using the function, xmlToDataFrame

```
> xmldataframe <- xmlToDataFrame("d:/XMLFile.xml")
```

Display the output of the data frame

```
> xmldataframe
  EMPID  EMPNAME           SKILLS            DEPT
1  1001 Merrilyn          MongoDB ComputerScience
2  1002    Ramya PeopleManagement  HumanResources
3  1003   Fedora      Recruitment  HumanResources
```

Section 3: R Charts and Graphs

Chart forms in R

Pie chart

Step 1:

The data is stored in "D:\SampleSuperstore.csv". Data is available under the following columns.

Row ID, Order ID, Order Date, Ship Date, Ship Mode, Customer ID, Customer Name, Segment, Country, State, City, Postal Code, Region, Product ID, Category, Sub-Category, Product Name, Sales, Quantity, Discount, Price.

With the use of read.csv function, data is read from "D:\SampleSuperstore.csv" file and stored in the data frame named, "InputData".

```
> InputData <- read.csv("d:/SampleSuperstore.csv")
```

Step 2:

Data is grouped / aggregated on InputData$Sales by InputData$Category. The aggregation function used is "sum".

```
> GroupedInputData <- aggregate(InputData$Sales ~ InputData$Category,InputData,sum)
```

Display the aggregated data. As evident from the below display, data is available in three categories: "Furniture", "Office Supplies", and "Technology".

```
> GroupedInputData
  InputData$Category InputData$Sales
1          Furniture        156514.4
2    Office Supplies        132600.8
3         Technology        168638.0
```

Step 3:

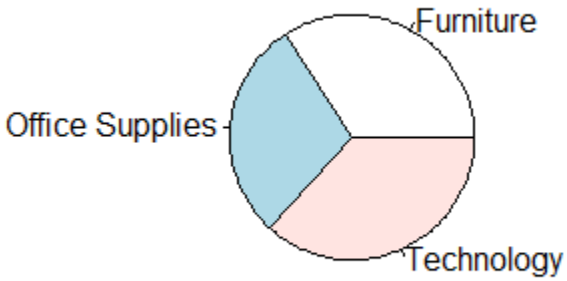
Data is converted into vector. The aggregated data, "InputData$Sales" is read into "XVector". Likewise the aggregated data, "InputData$Category" is read into vector, "LabelVector".

```
> XVector <- as.vector(GroupedInputData[['InputData$Sales']])

> LabelVector <- as.vector(GroupedInputData[['InputData$Category']])
```

Step 4:

Create the pie chart using the pie function.

```
> pie(XVector,LabelVector)
```



Bar Plot

Step 1:

The data is stored in "D:\SampleSuperstore.csv". Data is available under the following columns.

Row ID, Order ID, Order Date, Ship Date, Ship Mode, Customer ID, Customer Name, Segment, Country, State, City, Postal Code, Region, Product ID, Category, Sub-Category, Product Name, Sales, Quantity, Discount, Price.

With the use of read.csv function, data is read from "D:\SampleSuperstore.csv" file and stored in the data frame named, "InputData".

```
> InputData <- read.csv("d:/SampleSuperstore.csv")
```

Step 2:

Data is grouped / aggregated on InputData$Sales by InputData$Category. The aggregation function used is "sum".

```
> GroupedInputData <- aggregate(InputData$Sales ~ InputData$Category,InputData,sum)
```

Display the aggregated data. As evident from the below display, data is available in three categories: "Furniture", "Office Supplies", and "Technology".

```
> GroupedInputData
  InputData$Category InputData$Sales
1          Furniture        156514.4
2    Office Supplies        132600.8
3         Technology        168638.0
```

Step 3:

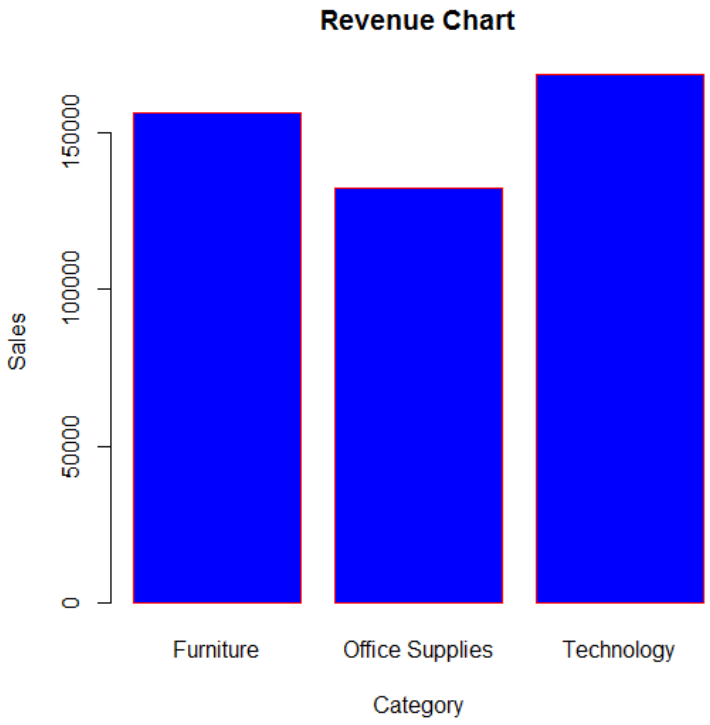
Data is converted into vector. The aggregated data, "InputData$Sales" is read into "XVector". Likewise the aggregated data, "InputData$Category" is read into vector, "LabelVector".

```
> XVector <- as.vector(GroupedInputData[['InputData$Sales']])
```

Step 4:

Create the bar chart using the barplot function.

```
> barplot(XVector, names.arg=LabelVector, xlab = "Category", ylab="Sales", col="blue", main="Revenue Chart", border="red")
```



Line chart

Step 1:

The data is stored in "D:\SampleSuperstore.csv". Data is available under the following columns.

Row ID, Order ID, Order Date, Ship Date, Ship Mode, Customer ID, Customer Name, Segment, Country, State, City, Postal Code, Region, Product ID, Category, Sub-Category, Product Name, Sales, Quantity, Discount, Price.

With the use of read.csv function, data is read from "D:\SampleSuperstore.csv" file and stored in the data frame named, "InputData".

```
> InputData <- read.csv("d:/SampleSuperstore.csv")
```

Step 2:

Data is grouped / aggregated on InputData$Sales by InputData$Category. The aggregation function used is "sum".

```
> GroupedInputData <- aggregate(InputData$Sales ~ InputData$Category,InputData,sum)
```

Display the aggregated data. As evident from the below display, data is available in three categories: “Furniture”, “Office Supplies”, and “Technology”.

```
> GroupedInputData
  InputData$Category InputData$Sales
1          Furniture        156514.4
2    Office Supplies        132600.8
3         Technology        168638.0
```

Step 3:

Data is converted into vector. The aggregated data, “InputData$Sales” is read into “XVector”.

```
> XVector <- as.vector(GroupedInputData[['InputData$Sales']])
```

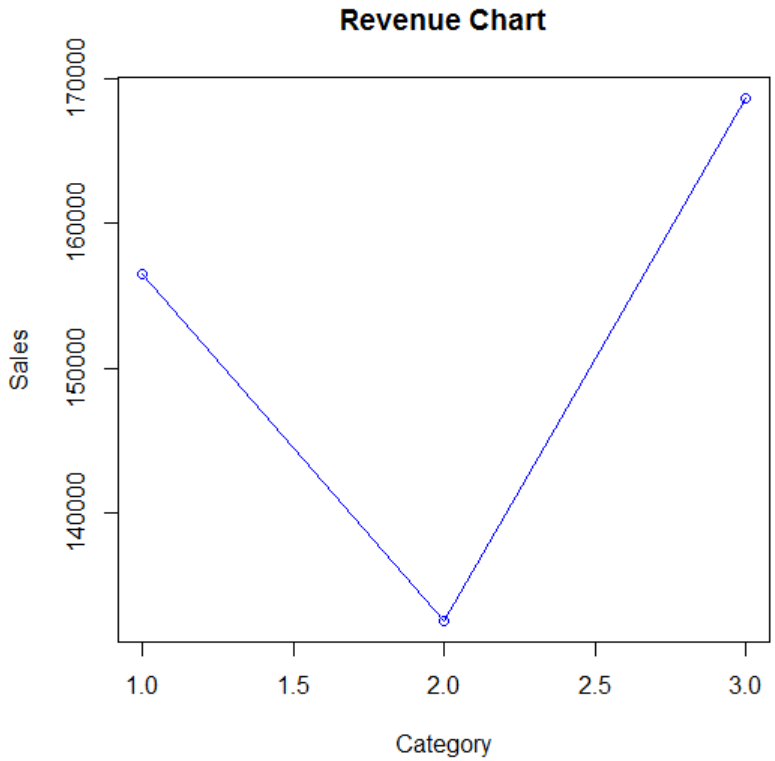
Step 4:

Create the line graph using the plot function.

plot(Xvector,type= “o”,xlab = “Category”, ylab= “Sales”, col= “blue”, main= “Revenue Chart”)

type takes the value "p" to draw only the points, "i" to draw only the lines and "o" to draw both points and lines.

```
> plot(XVector,type="o",xlab="Category", ylab="Sales", col="blue", main="Revenue Chart")
```



Scatter plot

Step 1:

The data is stored in "D:\SampleSuperstore.csv". Data is available under the following columns.

Row ID, Order ID, Order Date, Ship Date, Ship Mode, Customer ID, Customer Name, Segment, Country, State, City, Postal Code, Region, Product ID, Category, Sub-Category, Product Name, Sales, Quantity, Discount, Price.

With the use of read.csv function, data is read from "D:\SampleSuperstore.csv" file and stored in the data frame named, "InputData".

```
> InputData <- read.csv("d:/SampleSuperstore.csv")
```

Step 2:

Data is grouped / aggregated for InputData$Sales and InputData$Profit by InputData$Category. The aggregation function used is "sum".

```
> GroupedInputData <- aggregate(cbind(InputData$Sales, InputData$Profit) ~ InputData$Category,InputData,sum)
```

Display the aggregated data. As evident from the below display, data is available in three categories: "Furniture", "Office Supplies", and "Technology". V1 column has the aggregated sales, "InputData$Sales" and V2 column has the aggregated profits, "InputData$Profit".

```
> GroupedInputData
  InputData$Category       V1          V2
1          Furniture 156514.4   785.9786
2    Office Supplies 132600.8 20178.5828
3         Technology 168638.0 22243.5857
```

Step 3:

Data is converted into vector. The aggregated data, "InputData$Sales" is read into "XVector".

XVector <- as.vector(GroupedInputData[['V1']])

YVector <- as.vector(GroupedInputData[['V2']])

Shown below is the creation of the vector, "YVector".

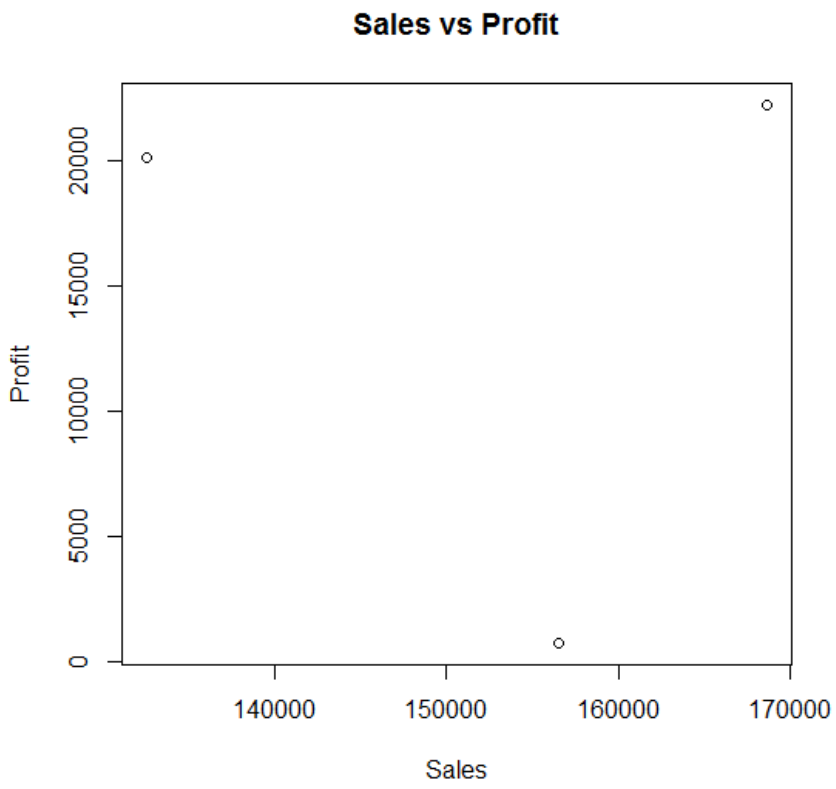
```
> YVector <- as.vector(GroupedInputData[['V2']])

> YVector
[1]   785.9786 20178.5828 22243.5857
```

Step 4:

Create the scatter plot using the plot function.

plot(x=XVector, y=YVector, xlab= "Sales", ylab= "Profit", main= "Sales vs Profit")

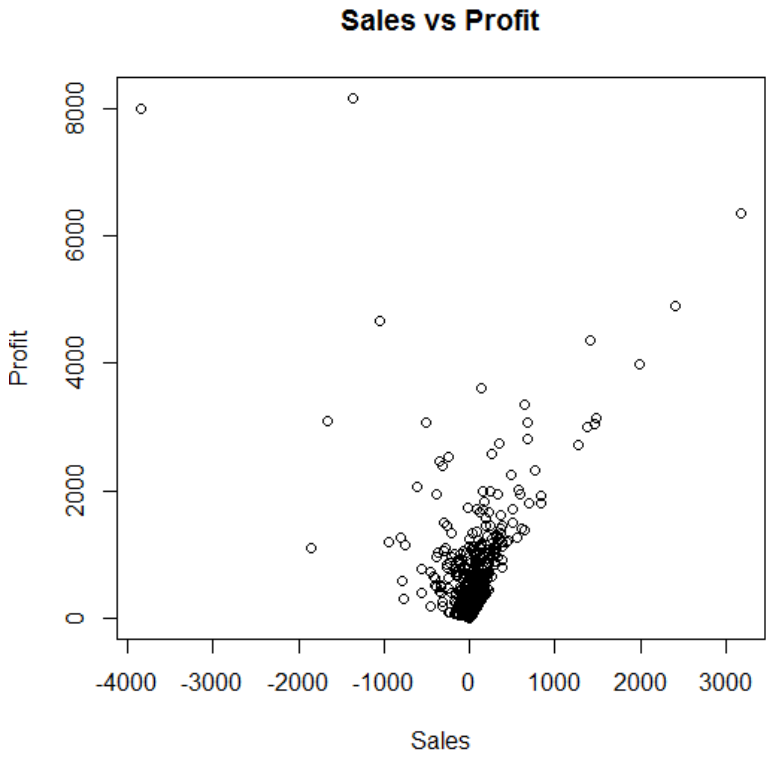
```
> plot(x=XVector, y=YVector, xlab="Sales", ylab="Profit", main="Sales vs Profit")
```



The graph above is owing to the aggregated data being plotted. We have data in three categories only, therefore you see three dots on the plot.

Let us now try to plot the scatter plot using disaggregated data.

```
> plot(x=InputData$Profit, y=InputData$Sales, xlab="Sales", ylab="Profit", main="Sales vs Profit")
```



Histogram

Step 1:

The data is stored in "D:\TestScore.csv". Data is available under the following columns.

State, Subject, Score, Year

A subset of the data is presented below:

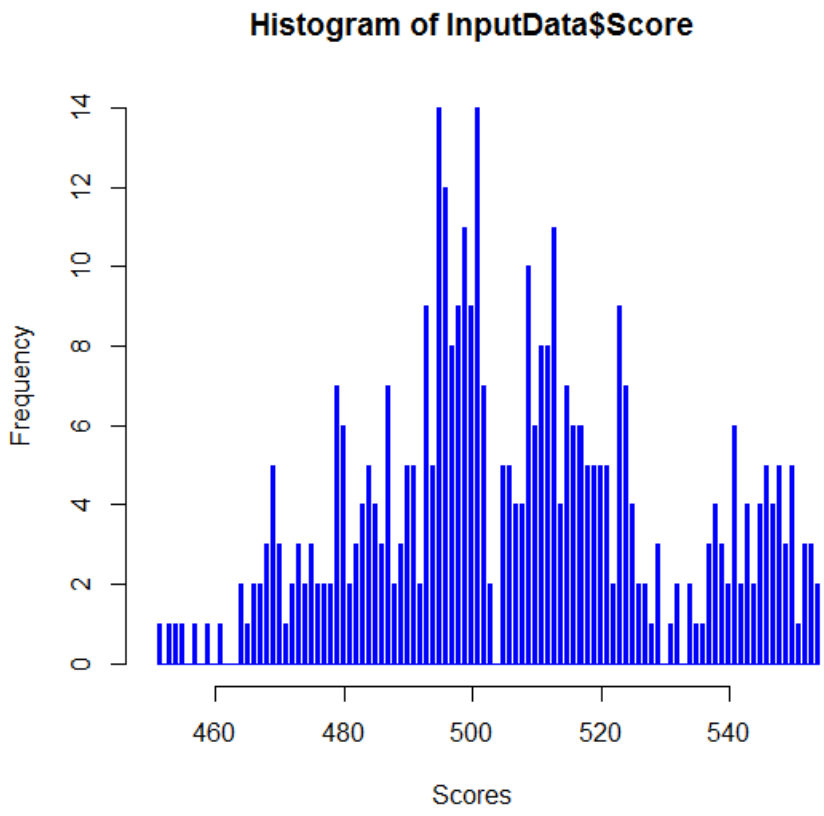
	A	B	C	D
1	**State**	**Score**	**Subject**	**Year**
2	Alabama	565	Critical reading	2005-2006
3	Alabama	561	Mathematics	2005-2006
4	Alabama	565	Writing	2005-2006
5	Alabama	557	Critical reading	2008-2009
6	Alabama	552	Mathematics	2008-2009
7	Alabama	549	Writing	2008-2009
8	Alabama	556	Critical reading	2010-2011
9	Alabama	550	Mathematics	2010-2011
10	Alabama	544	Writing	2010-2011
11	Alabama	546	Critical reading	2011-2012
12	Alabama	541	Mathematics	2011-2012
13	Alabama	536	Writing	2011-2012
14	Alaska	517	Critical reading	2005-2006
15	Alaska	517	Mathematics	2005-2006
16	Alaska	493	Writing	2005-2006
17	Alaska	520	Critical reading	2008-2009
18	Alaska	516	Mathematics	2008-2009
19	Alaska	492	Writing	2008-2009
20	Alaska	518	Critical reading	2010-2011
21	Alaska	515	Mathematics	2010-2011
22	Alaska	491	Writing	2010-2011

With the use of read.csv function, data is read from “D:\TestScore.csv” file and stored in the data frame named, “InputData”.

Step 2:

Create a histogram using the hist function.

```
> hist(InputData$Score, xlab="Scores", col="blue", border="blue", breaks = 500, xlim=c(450,550))
```



Boxplot

Step 1:

The data is stored in "D:\TestScore.csv". Data is available under the following columns.

State, Subject, Score, Year

A subset of the data is presented below:

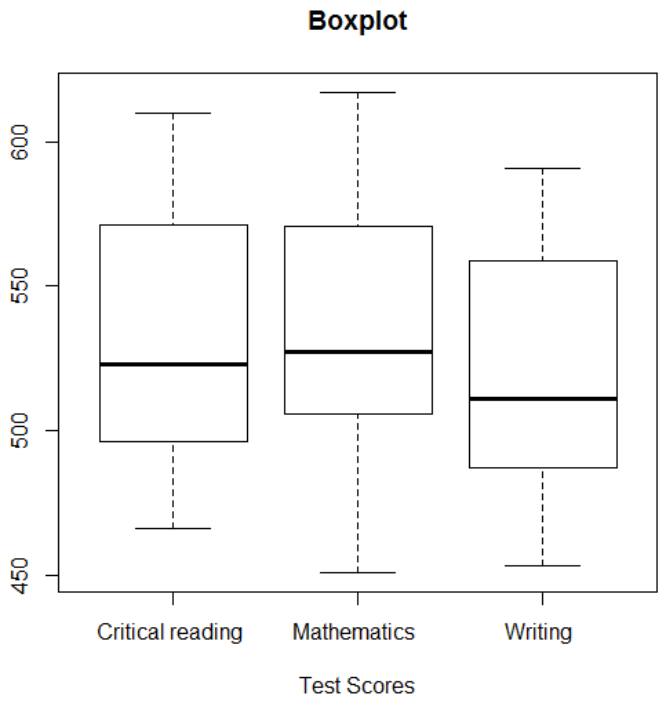
	A	B	C	D
1	**State**	**Score**	**Subject**	**Year**
2	Alabama	565	Critical reading	2005-2006
3	Alabama	561	Mathematics	2005-2006
4	Alabama	565	Writing	2005-2006
5	Alabama	557	Critical reading	2008-2009
6	Alabama	552	Mathematics	2008-2009
7	Alabama	549	Writing	2008-2009
8	Alabama	556	Critical reading	2010-2011
9	Alabama	550	Mathematics	2010-2011
10	Alabama	544	Writing	2010-2011
11	Alabama	546	Critical reading	2011-2012
12	Alabama	541	Mathematics	2011-2012
13	Alabama	536	Writing	2011-2012
14	Alaska	517	Critical reading	2005-2006
15	Alaska	517	Mathematics	2005-2006
16	Alaska	493	Writing	2005-2006
17	Alaska	520	Critical reading	2008-2009
18	Alaska	516	Mathematics	2008-2009
19	Alaska	492	Writing	2008-2009
20	Alaska	518	Critical reading	2010-2011
21	Alaska	515	Mathematics	2010-2011
22	Alaska	491	Writing	2010-2011

With the use of read.csv function, data is read from “D:\TestScore.csv” file and stored in the data frame named, “InputData”.

Step 2:

Create a boxplot using the boxplot function. Boxplot is used to show the 5 magic numbers: minimum, maximum, median, lower quartile and upper quartile.

```
> boxplot(InputData$Score ~ InputData$Subject, data = InputData, xlab = "Test Scores", main = "Boxplot")
```


Boxplot
600
550
500
450
Critical reading
Mathematics
Writing
Test Scores

Section 4:

R Statistics

Mean:

Objective: To determine the mean of a set of numbers. To plot the numbers in a barplot and have a straight line run through the plot at the mean.

Act:

Step 1: To create a vector, "numbers".

```
> numbers <- c(1,3,5,2,8,7,9,10)
```

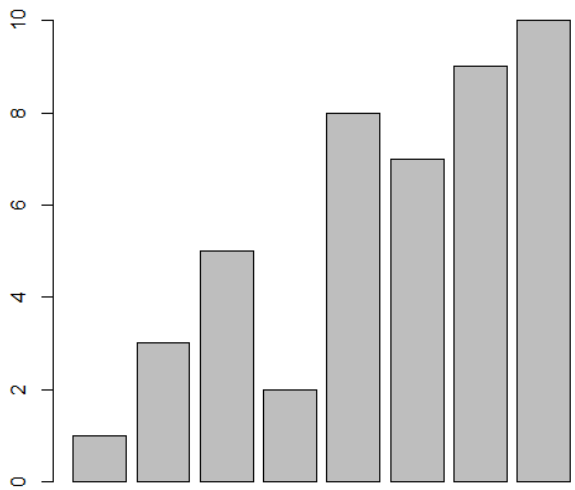
Step 2: To compute the mean value of the set of numbers contained in the vector, "numbers".

```
> mean(numbers)
[1] 5.625
```

Outcome: The mean value for the vector, "numbers" is computed as 5.625.

Step 3: To plot a bar plot using the vector, "numbers".

```
> barplot(numbers)
```

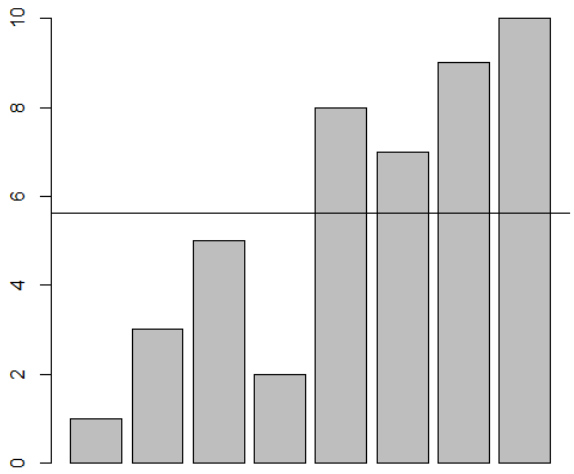


Step 4: Use the abline function to have a straight line (horizontal line) run through the bar plot at the mean value.

The abline function can take an h parameter with a value at which to draw a horizontal line, or a v parameter for a vertical line. When it's called, it updates the previous plot.

Draw a horizontal line across the plot at the mean:

```
> barplot(numbers)
> abline(h = mean(numbers))
```



Outcome: A straight line at the computed mean value (5.625) runs through the bar plot computed on the vector, "numbers".

Median:

Objective: To determine the median of a set of numbers. To plot the numbers in a bar plot and have a straight line run through the plot at the median.

Act:

Step 1: To create a vector, "numbers".

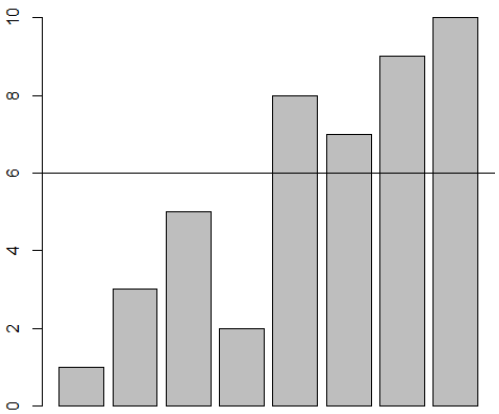
```
> numbers <- c(1,3,5,2,8,7,9,10)
```

Step 2: To compute the median value of the set of numbers contained in the vector, "numbers".

```
> median(numbers)
[1] 6
```

Step 3: To plot a bar plot using the vector, "numbers". Use the abline function to have a straight line (horizontal line) run through the bar plot at the median.

```
> barplot(numbers)
> abline(h = median(numbers))
```



Outcome: A straight line at the computed median value (6.0) runs through the bar plot computed on the vector, "numbers".

Standard Deviation:

Objective: To determine the standard deviation. To plot the numbers in a barplot and have a straight line run through the plot at the mean and another straight line run through the plot at mean + standard deviation.

Step 1: To create a vector, "numbers".

```
> numbers <- c(1,3,5,2,8,7,9,10)
```

Step 2: To compute the mean value of the set of numbers contained in the vector, "numbers".

```
> mean(numbers)
[1] 5.625
```

Step 3: To determine the standard deviation of the set of numbers held in the vector, "numbers".

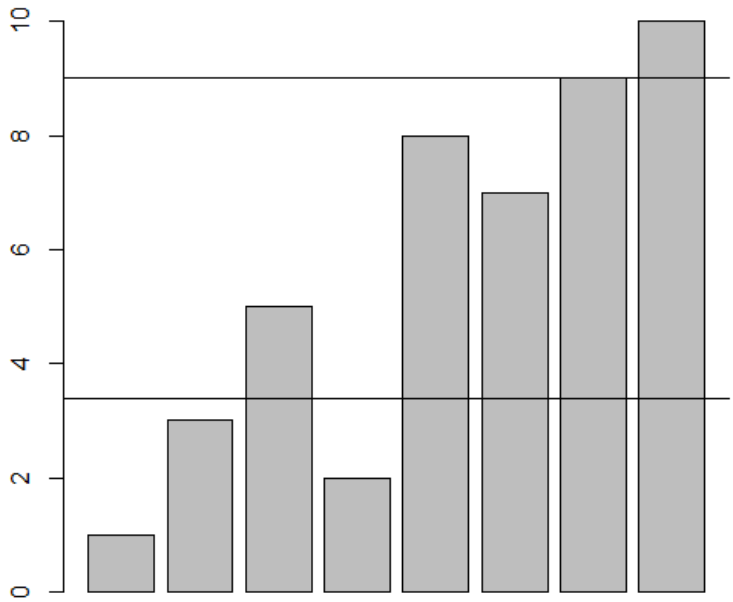
```
> deviation <- sd(numbers)
> deviation
[1] 3.377975
```

Step 4: To plot a bar plot using the vector, "numbers"..

```
> barplot(numbers)
```

Step 5: Use the abline function to have a straight line (horizontal line) run through the bar plot at the mean value (5.625) and another straight line run through the bar plot at mean value + standard deviation (5.625 + 3.377975)

```
> barplot(numbers)
> abline(h=sd(numbers))
> abline(h=sd(numbers) + mean(numbers))
```



Mode

Objective: To determine the mode of a set of numbers.

R does not have a standard built-in function to determine the mode. We will write out own, "Mode" function. This function will take the vector as input and return the mode as the output value.

Act:

Step 1:

Create a user-defined function, "Mode"

```
Mode <- function(v) {
        UniqValue <- unique(v)
        UniqValue[which.max(tabulate(match(v, UniqValue)))]
}
```

```
> Mode <- function(v) {
+ UniqValue <- unique(v)
+ UniqValue[which.max(tabulate(match(v, UniqValue)))]
+ }
```

While writing the above function, "Mode", we have used 3 other functions provided by R, namely, "unique", "tabulate" and "match".

unique function: The "unique" function will take the vector as input and returns the vector with the duplicates removed.

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

> unique(v)
[1] 2 1 3 4 5
```

match function: Takes a vector as input and returns a vector that has the positions of (first) matches of its first arguments in its second.

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

>  UniqValue <- unique(v)

> UniqValue
[1] 2 1 3 4 5

> match(v,UniqValue)
 [1] 1 2 1 3 2 1 3 4 2 5 5 3 1 3
```

tabulate function: Takes an integer valued vector as input and counts the number of times each integer occurs in it.

```
> tabulate(match(v,UniqValue))
[1] 4 3 4 1 2
```

Going by our example, "2" occurs 4 times, "1" occurs 3 times, "3" occurs 4 times, "4" occurs 1 time and "5" occurs 2 times.

Step 2:

Create a vector, "v"

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

Step 3:

Call the function, "Mode" and pass the vector, "v" to it.

```
> Output <- Mode(v)
```

Step 4:

Print out the mode value of the vector, "v".

```
> print(Output)
[1] 2
```

Let us pass a character vector, "charv" to the "Mode" function.

Step 1: Create a character vector, “charv”.

```
> charv <- c("o","it","the","it","it")
```

Step 2: Call the function, “Mode” and pass the character vector, “charv” to it.

```
> Output <- Mode(charv)
```

Step 3:

Print out the mode value of the vector, “v”.

```
> print(Output)
[1] "it"
```