An Introduction to R

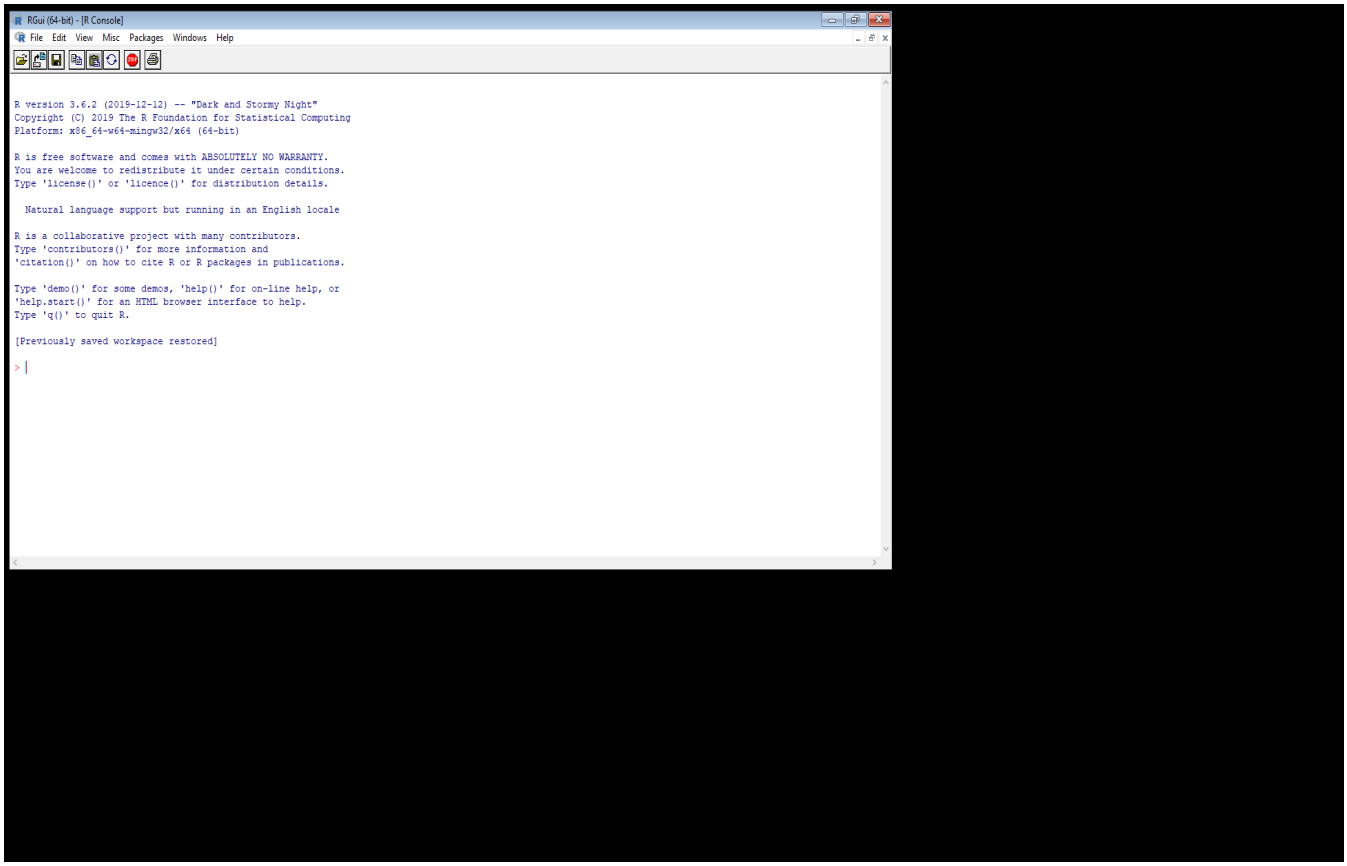
What is R?

R is a free version of a commercial statistical language/package called S-Plus. The commands in R and S-Plus are almost identical. There is a huge range of statistical commands in R, from basic summary statistics to cutting edge research applications. It is also a programming environment, so you can soon build up your own routines. Almost all practical statistical analysis is carried out on a computer, and so becoming familiar with computer packages and languages is an essential task for any statistician.

Starting R on Your Personal PC

If you have your own PC then R can be downloaded from CRAN (Comprehensive R Archive Network): http://www.r-project.org

As soon as you open R, then this brings you the R console:



The Very Basics Commands in R

The ">" symbol is the R prompt, waiting for you to type a command. We can familiar with the very basics R commands just by using R as a calculator.

When you type a command at the prompt (R console) and hit Enter, your computer executes the command and shows you the results. For example, if you type 1 + 1 at the R console and hit Enter, R console will display:

```
> 1+1

[1] 2
```

You'll notice that a [1] appears next to your result. R is just letting you know that this line begins with the first value in your result.

```
> 300-200+370

[1] 470
```

Let's introduce more arithmetic operators which are used in R to calculate value of a big expression.

Description	R Symbol	Example

Addition	+	> 5+16 [1] 21
Subtraction	–	> 5-16 [1] -11
Multiplication	*	> 5*16 [1] 80
Division	/	> 16/5 [1] 3.2
Exponent	^ or **	> 2^4 [1] 16
Bracket	()	> 3*(2+4) [1] 18
Modulus (Remainder from division)	%%	> 16%%5 [1] 1
Integer Division	%/%	> 16%/%5 [1] 3

Different R Functions

Suppose we want to calculate the value of the expression of $e^3 + 3\times15\div5 - log3 + |-10|$ which consists exponential, logarithm and absolute value functions. The R offers us a lots of built in functions like these. Some of the most useful R function names are listed below along with the description of how they work.

Description	R Symbol	Example
Calculate the exponential	exp	exp(10)
Calculate the absolute value	abs	abs(-10)
Calculate the logarithm of x with base y	log(x,y)	log(12,10)
Calculate the square root	sqrt	sqrt(10)
Calculate the factorial	factorial	factorial(3)
Calculate the sin of x	sin	sin(30°)
Calculate the cos of x	cos	cos(45°)
Calculate the tan of x	tan	tan(30°)
Calculate the x , x , x	asin, acos, atan	asin(0.5)

Creating vectors, matrices and array

A vector is a sequence of elements that share the same data type. These elements are known as components of a vector. The vector elements are all of the same mode (either character, numeric, or logical) i.e.the data types can be logical, integer, double, character, complex or raw.

Creating character vector

```
> y <- c('Bengali', 'English', 'Hindi')
> y
[1] "Bengali" "English" "Hindi"
> class(y)
[1] "character"
> typeof(y)
[1] "character"
> is.vector(y)
[1] TRUE
```

To determine the number of elements of your vector use the command `length`. For example,

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

The following example shows that a vector always consists homogenous elements (2 is treated as character)

```
> y <- c(2,'Bengali', 'English', 'Hindi')
> typeof(y)
[1] "character"
```

Creating numeric vector

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

Some useful techniques to create a long vector

Suppose we wish to create the following vector x

```
> x<-c(1,2,3,4,5,7,9,11,13,15,17,17,17,17,17,17)

> x

 [1]  1  2  3  4  5  7  9 11 13 15 17 17 17 17 17 17
```

You can input all the elements one by one in R console to create the above vector which may take long time (just think you have 1000 elements like this in your vector). To create this in an efficient way the following commands can be used:

```
> seq(1,5,1)  ## start from 1 and end

[1] 1 2 3 4 5


> seq(9,17,2)

[1]  9 11 13 15 17
```

"seq" command can be used to create a long arithmetic sequence. Explore how "seq" command works in above two examples.

```
> rep(17,5)

[1] 17 17 17 17 17
```

On the other hand, "rep" command replicates a specific number n (here n=5) number of times. Using "seq" and "rep" commands we can create the above vector in an elegant way like this

```
y<-c( seq(1,5,1), seq(9,17,2),rep(17,5) )

> y

 [1]  1  2  3  4  5  9 11 13 15 17 17 17 17 17 17


w2<-rep(c(1,3),5)   ## Input a vector of numbers repeatedly

> w

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

```
> x<-c(1,2,3,4,5,7,9,11,13,15,17,17,17,17,17,17)
> x
 [1]  1  2  3  4  5  7  9 11 13 15 17 17 17 17 17 17
```

Returning 7th element of a vector

```
> x[7]
 [1] 9
```

Returning 5 to 7th elements of a vector

```
> x[5:7]
[1] 5 7 9
```

Returning 5th and 7th elements of vector x

```
> x[c(5,7)]
[1] 5 9
> x
 [1]  1  2  3  4  5  7  9 11 13 15 17 17 17 17 17 17
```

Dropping 5th value of x

```
> y<-x[-5]
[1]  1  2  3  4  7  9 11 13 15 17 17 17 17 17 17
```

Dropping 3rd to 5th values from x

```
> x
 [1]  1  2  3  4  5  7  9 11 13 15 17 17 17 17 17 17
> y<-x[-(3:5)]
 [1]  1  2  7  9 11 13 15 17 17 17 17 17 17
```

Dropping 3rd and 5th values from x

```
> y<-x[-c(3,5)]
 [1]  1  2  4  7  9 11 13 15 17 17 17 17 17 17
```

```
x[4]<-100                    ## replace 4th element of x by 100
```

replace second, fifth and tenth elements by 100, 200 and 500 respectively

```
x[c(2,5,10)]<-c(100,200,500)
```

Showing elements from 6th to the end of a vector (first 5 elements are not shown)

```
tail(x,-5)


y<-c(x,171)
## adding a new value with existing x vector and rename it y
```

"which" command in R: Sometime we need to know the position of an element of a specific vector which satisfies a certain condition. In this case we can use "which" command to know the position of the specific element. For example,

```
> x
[1]  1  2  3  4  5  7  9 11 13 15 17 17 17 17 17 17


which(x==13)
## return the positions of element where the condition is True
> which(x==13)
[1] 9          ## 9th position value is 13
```

Sorting ascending or descending order of the elements of a specific vector

sometime you may need to sort the elements of a vector either in ascending (smallest to largest) or descending (largest to smallest) order. For example, you may want to see the scores of an exam in descending order (find the person who got the highest score). In this case, we can use "sort" command to sort the elements of a vector. For example,

```
z<-c(68.51, 66.64, 66.90, 67.84, 61.33, 63.47, 63.59, 65.90,
61.47, 69.49, 63.78, 64.53, 61.25, 63.65, 69.04)
z
[1] 68.51 66.64 66.90 67.84 61.33 63.47 63.59 65.90 61.47 69.49
63.78 64.53 61.25 63.65 69.04
```

```
> sort(z)
[1] 61.25 61.33 61.47 63.47 63.59 63.65 63.78 64.53 65.90 66.64
66.90 67.84 68.51 69.04 69.49
help(sort) ## tells how sort command works
sort(x, decreasing = FALSE, ...)
sort(z,decreasing=T)
> sort(z,decreasing=T)
[1] 69.49 69.04 68.51 67.84 66.90 66.64 65.90 64.53 63.78 63.65
63.59 63.47 61.47 61.33 61.25
```

Combining two different vectors:

Suppose we have two vectors `x` and `y` and we want to create a new vector `z` whose elements are the elements of `x` and `y`.

```
x<-c(2, 5, 8, 22, 4)
y<-c(12, 34, 25, 18, 21)
z<-c(x,y)
 [1]  2  5  8 22 4 12 34 25 18 21
```

Showing the first n elements of a vector

```
> head(z,5)                ## n = 5
[1]  2  5  8 22  4
```

Showing last n elements of a vector

```
> tail(z,5)
[1] 12 34 25 18 21
```

Showing all the elements but first n elements are deleted of a vector

```
> tail(z,-5)
[1] 12 34 25 18 21
```

Addition, Subtraction, Multiplication and Division of vectors

Suppose we have two vectors y and z which are

```
y<-c(11, 5, 0, 2, 6, 19, 21, 43, 54,38)
y
 [1] 11  5  0  2  6 19 21 43 54 38


z<-c(2,  5,  8, 22,  4, 12, 34, 25, 18, 21)
> z
 [1]  2  5  8 22  4 12 34 25 18 21
```

Adding 5 with z yields (`## 5 is added with every element of z)`

```
> z+5
 [1]  7 10 13 27  9 17 39 30 23 26
```

Adding `z` with `y` yields (element wise addition, both the vectors should be the same length)

```
> y+z
 [1] 13 10  8 24 10 31 55 68 72 59
```

Subtracting `z` from `y` yields (element wise subtraction)

```
  y-z
 [1]   9   0  -8 -20   2   7 -13  18  36  17
```

`> y*z (`element wise multiplication)

```
 [1]   22   25    0   44   24  228  714 1075  972  798
```

`> y/z` (element wise division)

```
  [1]  5.50000000  1.00000000  0.00000000  0.09090909  1.50000000
1.58333333 0.61764706 1.72000000 3.00000000 1.80952381
```

```
round(y/z,3)      ## taking 3 decimal number of a fraction
[1] 5.500 1.000 0.000 0.091 1.500 1.583 0.618 1.720 3.000 1.810

floor(y/z)        ## taking the nearest small integer value
[1] 5 1 0 0 1 1 0 1 3 1

ceiling(y/z)      ## taking the nearest large integer value
[1] 6 1 0 1 2 2 1 2 3 2
```

Creating Matrix

Matrix is a two dimensional data structure in R programming. Dimension can be checked directly with the dim() function. We can check if a variable is a matrix or not with the class () function.

How to create a matrix in R programming?

Matrix can be created using the matrix() function. Dimension of the matrix can be defined by passing appropriate value for arguments nrow and ncol. Providing value for both dimension is not necessary. If one of the dimension is provided, the other is inferred from length of the data.

```
> matrix(1:9, nrow = 3, ncol = 3)

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

[1,]    1    4    7

[2,]    2    5    8

[3,]    3    6    9

> # same result is obtained by providing only one dimension

> matrix(1:9, nrow = 3)

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

```
[1,]    1    4    7

[2,]    2    5    8

[3,]    3    6    9
```

We can see that the matrix is filled column-wise. This can be reversed to row-wise filling by passing `TRUE` to the argument `byrow`.

```
> matrix(1:9, nrow=3, byrow=TRUE)    # fill matrix row-wise

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

[1,]    1    2    3

[2,]    4    5    6

[3,]    7    8    9
```

In all cases, however, a matrix is stored in column-major order internally as we will see in the subsequent sections.

It is possible to name the rows and columns of matrix during creation by passing a 2 element list to the argument `dimnames`.

```
> x <- matrix(1:9, nrow = 3, dimnames = list(c("X","Y","Z"),
c("A","B","C")))



> x

  A B C

X 1 4 7

Y 2 5 8
```

```
Z 3 6 9
```

These names can be accessed or changed with two helpful functions `colnames()` and `rownames()`.

```
> colnames(x)

[1] "A" "B" "C"

> rownames(x)

[1] "X" "Y" "Z"

> # It is also possible to change names

> colnames(x) <- c("C1","C2","C3")

> rownames(x) <- c("R1","R2","R3")

> x
    C1 C2 C3

R1  1  4  7

R2  2  5  8

R3  3  6  9
```

Another way of creating a matrix is by using functions `cbind()` and `rbind()` as in column bind and row bind.

```
> cbind(c(1,2,3),c(4,5,6))

       [,1] [,2]

[1,]    1    4
```

```
[2,]    2    5

[3,]    3    6

> rbind(c(1,2,3),c(4,5,6))

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

[1,]    1    2    3

[2,]    4    5    6
```

Finally, you can also create a matrix from a vector by setting its dimension using `dim()`.

```
> x <- c(1,2,3,4,5,6)

> x

[1] 1 2 3 4 5 6

> class(x)

[1] "numeric"



> dim(x) <- c(2,3)

> x

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

[1,]    1    3    5

[2,]    2    4    6
```

```
> class(x)

[1] "matrix"
```

How to access Elements of a matrix?

We can access elements of a matrix using the square bracket x[row, column], where x is a matrix. Here row and column can be vectors.

Using integer vector as index

We specify the row numbers and column numbers as vectors and use it for indexing. If any field inside the bracket is left blank, it selects all. We can use negative integers to specify rows or columns to be excluded.

```
> x

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

[1,]    1    4    7

[2,]    2    5    8

[3,]    3    6    9



> x[c(1,2),c(2,3)]    # select rows 1 & 2 and columns 2 & 3

      [,1] [,2]

[1,]    4    7

[2,]    5    8

> x[c(3,2),]    # leaving column field blank will select entire columns
```

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

[1,]    3    6    9

[2,]    2    5    8

> x[,]  # leaving row and column field blank will select entire matrix

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

[1,]    1    4    7

[2,]    2    5    8

[3,]    3    6    9

> x[-1,]    # select all rows except first

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

[1,]    2    5    8

[2,]    3    6    9
```

In the above example, the matrix `x` is treated as vector formed by stacking columns of the matrix one after another, i.e., `(4,6,1,8,0,2,3,7,9)`.

The indexing logical vector is also recycled and thus alternating elements are selected. This property is utilized for filtering of matrix elements as shown below.

```
> x[x>5]    # select elements greater than 5

[1] 6 8 7 9

> x[x%%2 == 0]    # select even elements
```

```
[1] 4 6 8 0 2
```

How to modify a matrix in R?

We can combine assignment operator with the above learned methods for accessing elements of a matrix to modify it.

```
> x

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

[1,]    1    4    7

[2,]    2    5    8

[3,]    3    6    9

> x[2,2] <- 10; x    # modify a single element

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

[1,]    1    4    7

[2,]    2   10    8

[3,]    3    6    9

> x[x<5] <- 0; x    # modify elements less than 5

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

[1,]    0    0    7

[2,]    0   10    8
```

```
[3,]    0    6    9
```

A common operation with matrix is to transpose it. This can be done with the function `t()`.

```
> t(x)    # transpose a matrix

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

[1,]    0    0    0

[2,]    0   10    6

[3,]    7    8    9
```

We can add row or column using `rbind()` and `cbind()` function respectively. Similarly, it can be removed through reassignment.

```
> cbind(x, c(1, 2, 3))    # add column

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

[1,]    0    0    7    1

[2,]    0   10    8    2

[3,]    0    6    9    3

> rbind(x,c(1,2,3))    # add row

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

[1,]    0    0    7

[2,]    0   10    8
```

```
[3,]    0    6    9

[4,]    1    2    3

> x <- x[1:2,]; x    # remove last row

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

[1,]    0    0    7

[2,]    0   10    8
```

Dimension of matrix can be modified as well, using the `dim()` function.

```
> x

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

[1,]    1    3    5

[2,]    2    4    6

> dim(x) <- c(3,2); x    # change to 3X2 matrix

[,1] [,2]

[1,]    1    4

[2,]    2    5

[3,]    3    6

> dim(x) <- c(1,6); x    # change to 1X6 matrix

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

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

Matrix Addition and Subtraction

```
MatA+MatB; MatA-MatB
```

Element wise multiplication (not matrix multiplication)

```
MatA*MatB; MatA/MatB
```

Matrix multiplication

```
MatA%*%matB
```

Transpose of matrix

```
t(MatA)
```

Determinant of matrix and Inverse of matrix

```
Det(MatA);solve(MatC)
```

Matrices and Data Frames:

Both represent 'rectangular' data types, meaning that they are used to store tabular data, with rows and columns. The main difference, as you'll see, is that **matrices can only contain a single class of data, while data frames can consist of many different classes of data**.

Create a DataFrame in R

Let's start with a simple example, where the dataset is:

Name	Age
Jon	23
Bill	41
Maria	32
Ben	58
Tina	26

```
Name <- c("Jon", "Bill", "Maria", "Ben", "Tina")
Age <- c(23, 41, 32, 58, 26)

df <- data.frame(Name, Age)
print (df)
```

Returning a subset of a data set

```
Age <- c(22, 25, 18, 20)
Name <- c("James", "Mathew", "Olivia", "Stella")
Gender <- c("M", "M", "F", "F")
```

then the required R–code to get the following output is

```
DataFrame = data.frame(c(Age), c(Name), c(Gender))
subset(DataFrame, Gender == "M")
```

```
##   Age   Name Gender
## 1  22  James      M
## 2  25 Mathew      M
```

Reading/Importing data set from your personal computer:

Survey data are supposed to be saved in your personal PC, and your data may be saved in Excel or SPSS or R format. Whatever the format your data is saved, you can easily call (read) your data set in R.

Read `text/csv` file from your local file

Suppose you have a file named “FAO1” in your personal computer, and you want to read or import this file in R console. You need to use the following command to import your local file in R console. The first part of the following code is the location of your file where it is saved.

```
df<-read.table('C:\\Users\\Statistics\\Desktop\\R_Lecture\\FAO1.txt')
print(df)

df<-read.csv('C:\\Users\\Statistics\\Desktop\\R_Lecture\\FAO1.txt', header=FALSE)
print(df)
```

Reshape your data according to what you want to do

```
a<-as.integer(df[1:4,1])
b<-as.character(df[5:8,1])
c<-as.character(df[9:12,1])
X<-data.frame(a,b,c)
colnames(X)<-c("Age","Name", "Gender")
```

Survey data saved in notepad are eventually imported in R and converted into matrix form, X.

Read `Excel(.xlsx)` file from your local file

Before reading or importing your excel data in R, you need to save as your Excel data as `csv(MS_DOS)` format. After saved as your Excel data in `csv(MS_DOS)` format just follow the previous tricks.

```
Edata<-read.csv('C:\\Users\\Statistics\\Desktop\\R_Lecture\\Dengue_Data_New.csv', header=TRUE)
print(Edata)
```

Read `SPSS(.sav)` file from your local file

To read or import SPSS (.sav) data file from your personal computer, you need to install foreign package from R directory (installation requires internet connection). After installing the foreign package in your pc, you need to have (attach) it in your pc, and you can attach it in your pc by typing library command. Finally, you need to use the following codes to read or import SPSS (.sav) data in R.

```
rm(list=ls())
install.packages("foreign")
library(foreign)
data<-read.spss("C:/Users/Statistics/Desktop/R_Lecture/spss.sav")
attach(data)
names(data)
```

Obtain summary statistics

```
sort(y)                                  ## descending order (by
default)sort(x)
sort(y,decreasing=TRUE)   ## ascending order

round(7/3,3)   ## taking 3 decimal number of a fraction
floor(7/3)     ## taking the nearest small integer  value
ceiling(7/3)   ## taking the nearest large integer  value


mean(y)        ## Mean of a vector
median(y)      ## Median of a vector
var(y)         ## Variance of vector
sd(y)          ## Standard deviation of a vector
quantile(y)    ## The basic quantile points
quantile(y, probs=c(.10,.35)) ## Manual quantile points
summary(y)
```

Different Graphs in R

R produces very nice quality graphs both for qualitative and quantitative variables and this is one of the reasons why people use it. We consider some hypothetical data in the following which have been used for producing different graphs.

```
Year<-c(2001:2020)

Rice.Ban<-c(24310,25187,26152,25600,28758,29000,28800,31200,31000,31700,33700,33820,34390,3450
0,34500,34578,32650,34909,35850,36000)

Rice.Ind<-c(32960,33411,35024,34830,34959,35300,37000,38310,36370,35500,36500,36550,36300,3556
0,36200,36858,37000,34200,33500,34900)

age<-c(37,35,38,35,34,33,32,35,36,37,30,34,35,40,39,33,39,36,37,33)

Height<-c(73.84,68.78,74.11,71.73,69.88,67.25,68.78,68.34,67.01,63.45,71.19,71.64,64.76,69.28,
69.24,67.64,72.41,63.97,69.64,67.93)

Weight<-c(241.89,162.31,212.74,220.04,206.34,152.21,183.92,167.97,175.92,156.39,186.60,213.74,
167.12,189.44,186.43,172.18,196.02,172.88,185.98,182.42)

Language<-c("Bangla","English","Mandarin","Hindi","Spanish")

People<-c(265,1132,1117,615,534)
```

```
## Scatter plot

plot(Year,Rice.Ban)   ## plot X vs Y

## Defining X-limit and Y-limit

plot(Year,Rice.Ban,xlim=c(2000,2020),ylim=c(23000,38000))

## Adding a title

plot(Year,Rice.Ban,xlim=c(2000,2020),ylim=c(23000,38000),main="Yearly      Rice
Production (1000 MT)")

## Joining points with straight lines

plot(Year,Rice.Ban,xlim=c(2000,2020),ylim=c(23000,38000),main="Yearly      Rice
Production (1000 MT)", type="l")

## Changing line colors

plot(Year,Rice.Ban,xlim=c(2000,2020),ylim=c(23000,45000),type="l",main="Yearly
Rice Production (1000 MT)", col="green")   ## Changing line colors

## Adding another plot in same graph

points(Year,Rice.Ind,xlim=c(2000,2020),type="l", lty=2, col="red")

## Adding legend

## legend position with defining (x,y) co-ordinates

legend(x=2001,y=44000,legend=c("Bangladesh","Indonesia"),col=c("green","red"),l
ty=c(1,2))

## legend position with user manually by clicking on graph

legend(locator(1),legend=c("Bangladesh","Indonesia"),col=c("green","red"),lty=c
(1,2))
```

Bar plot for Categorical or Qualitative Variable

```
## Creating data matrix ##
```

```
data1<-matrix(c(Rice.Ban[11:20]),nrow=1,byrow=T)
colnames(data1)<-2011:2020           ## Giving column names
rownames(data1)<-c("R.B")            ## Giving row names

##bar plot where X-axis are defined as columns ##

barplot(data1,xlab="Year",ylab="Production",main="Yearly Rice Production in
Bangladesh (1000 MT)")
```

Group bar plot and stacked bar plot

```
data2<-matrix(c(Rice.Ban[16:20],Rice.Ind[16:20]),nrow=2,byrow=T)
colnames(data2)<-2016:2020
rownames(data2)<-c("R.B","R.I")

## Drawing Group bars ##

barplot(data2,ylim=c(0,50000),xlab="Year",ylab="Production",main="Yearly Rice
Production comparision(1000 MT)",col=c("green","red"), beside=TRUE)

## Adding legend ##

legend(locator(1),lty=c(1,1),col=c("green","red"),legend=c("Bangladesh","Indone
sia"))
```

Drawing stacked bars (Default)##

```
barplot(data2,ylim=c(0,100000),xlab="Year",ylab="Production",main="Yearly Rice
Production comparision(1000 MT)",col=c("green","red"), beside=F)

## Adding legend ##

legend(locator(1),lty=c(1,1),col=c("green","red"),legend=c("Bangladesh","Indone
sia"))
```

Pie charts

```
pie(People,labels=Language,main="Pie chart of diffrent Language speaking
People")
```

```
## Histogram for continuous or numeric data ##

hist(Weight)                    ## Histogram with Frequency distribution ##
hist(Weight,probability=T)      ## Histogram with probabilities ##
hist(Weight,probability=T,breaks=10)## Defining number of class manually ##
```

```
par(mfrow=c(2,1))      ## Drawing 2 graphs in a single x-y coordinate ##

hist(Weight,probability=T,breaks=10)## Defining number of class manually
hist(Height,probability=T,breaks=10)## Defining number of class manually


dev.off()                            ## return to the original graph mode ##


boxplot(Height,xlab="Height",ylab="Value")## Drawing Boxplot
cor(Height,Weight) ## calculating correlation between two variable ##
```

Writing your own function

Suppose we want to calculate the sum of all elements of a given vector. Then “sum” function calculates the sum of all elements.

```
x<-c(5, 2, 7)

sum(x)



my_fun<-function(x)

         {

           total<-sum(x)

           return(total)

           }

my_fun(x)
```

…… .