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

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

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

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

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

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

... : numeric or complex or logical vectors.

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

Example1:-

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

Example2:-

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

6

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

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

... : numeric or complex or logical vectors.

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

Example1:-

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

Example 2:-

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

Extended Example: Calculating a probability :

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

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

The probability can be calculated using the prod() function. Let us assume that there are ‗n' independent events with the i^{th} event having the p_i probability of occurrence.
What is the probability of exactly one of these events occurring?

Considering an example where the value of n is 3. The events are named A, B, and C. Then we break down the computation as follows:

P(exactly one event occurs) = P(A and not B and not C) +
P(not A and B and not C) +
P(not A and not B and C)

P(A and not B and not C) would be $p_A (1 - p_B) (1 - p_C)$, and so on.

For general n, that is calculated as follows

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

(The ith term inside the sum is the probability that event i occurs and all the others do not occur.)

Here's code to compute this, with our probabilities pi contained in the vector p:

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

notp <- 1 – p :- creates a vector of all the –not occur‖ probabilities $1 - p_i$, using recycling.

The expression notp[-i] computes the product of all the elements of notp, except the ith

Cumulative Sums and Products:-

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

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

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

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

Minima and maxima:-

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

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

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

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

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

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

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

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

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

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

pmin() and pmax(): Element-wise minima and maxima of several vectors.

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

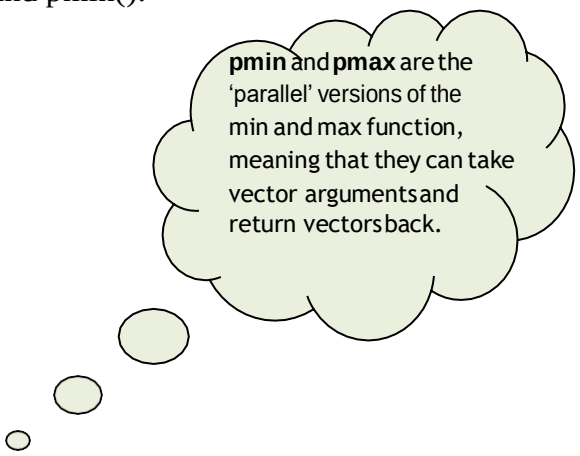
The max() and pmax() functions act analogously to min() and pmin().

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

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

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

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



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

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

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

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

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

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

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

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

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

Prefix the name as follows:

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

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

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

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

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

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

These functions also have arguments specific to the given distribution families. In our example, we use the df argument for the chi-square family, indicating the number of degrees of freedom.

Let‘s also compute the 95th percentile of the chi-square distribution with two degrees of freedom:

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

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

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

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

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

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

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

The console represents that there are two smallest values in vector x. The third smallest value being x[1], and so on. The same function order can also be used along with indexing for sorting data frames. This function can also be used to sort the characters as well as numeric values.

Another function which specifies the rank of every single element present in a vector is called rank( )

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

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

Example:- using order function on a dataframe.

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

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

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

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

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

The function computed 3·2+7·5+ 2·8 = 57.

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

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

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

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

$$x_1 + x_2 = 2$$

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

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

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

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

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

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

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

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

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

Vector Cross Product:- Let's consider the issue of vector cross products. The definition is very simple: The cross product of vectors (x_1, x_2, x_3) and (y_1, y_2, y_3) in three dimensional space is a new three-dimensional vector, as ($x_2y_3 - x_3y_2$, $-x_1y_3 + x_3y_1$, $x_1y_2 - x_2y_1$)

This can be expressed compactly as the expansion along the top row of the determinant, Here, the elements in the top row are merely placeholders.

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

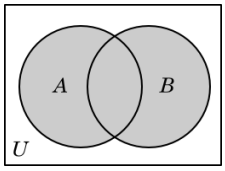
The point is that the cross product vector can be computed as a sum of subdeterminants. For instance, the first component in Equation 8.1, $x2y3 - x3y2$, is easily seen to be the determinant of the submatrix obtained by deleting the first row and first column.

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

Function to calculate cross product of vectors:

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

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



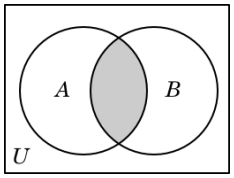
1) ***union(x,y):*** The union of two sets is defined as the set of all the elements that are members of set A, set B or both and is denoted by $A \cup B$ read as A union B.

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

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

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

= {1,2,3,4,5,a,b,c,d,e}



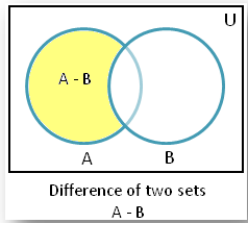
2) ***intersect(x,y):*** The intersection of any two sets A and B is the set containing of all the elements that belong to both A and B is denoted by $A \cap B$ read as A intersection B.

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

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

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

= {a,b}



Difference of two sets
A - B

3) ***setdiff(x,y):*** The set difference of any two sets A and B is the set of elements that belongs to A but not B. It is denoted by A-B and read as
_A difference B'. A-B is also denoted by A|B or A~B. It is also called the relative complement of B in A.
Eg: A={1,2,3,4,5,6} B={3,5,7,9}
A-B ={1,2,4,6}
B-A = {7,9}

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

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

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

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

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

n: n elements
r: r subset elements
...
$n_{Cr} = n!/(r! * (n-r)!)$

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

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

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

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

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

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

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

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

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

> x
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

There is one vector element for each line read, thus three elements here.

Alternatively, we can read it in one line at a time. For this, we first need to create a connection, as described next.

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

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

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

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

OUTPUT:

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

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

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

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

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

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

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

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

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

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

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

The file www will be created with these contents:

```
abc
de
f
```