STATISTICS WITH R PROGRAMMING

UNIT-3

Doing Math and Simulation in R:

A. Math Function

B. Extended Example Calculating Probability

C. Cumulative Sums and Products

D. Minima and Maxima

E. Calculus,

Functions for Statistical Distribution

Sorting

Linear Algebra Operation on Vectors and Matrices

A. Extended Example: Vector cross Product

B. Extended Example: Finding Stationary Distribution of Markov Chains

Set Operation

Input /out put : Accessing the Keyboard and Monitor

Reading and writer Files

Doing Math and Simulation in R:

A. Math Function

R includes an extensive set of built-in math functions. Here is a partial list:

1. **exp(): Exponential function, base e**
   Example:
   ```
   >exp(1)
   [1] 2.718282
   >
   ```

2. **log(): Natural logarithm**
   Example:
   ```
   > log(1)
   [1] 0.6931472
   >
   ```

3. **log10(): Logarithm base 10**
   Example:
   ```
   > log10(3)
   [1] 0.4771213
   >
   ```

4. **sqrt(): Square root**
   Example:
   ```
   > sqrt(5)
   [1] 2.236068
   >
   ```

5. **abs(): Absolute value**
   Example:
   ```
   > abs(-5)
   [1] 5
   >
   ```

6. **sin(), cos(), and so on: Trig functions**
   Example:
   ```
   > sin(30)
   [1] -0.9880316
   > cos(30)
   [1] 0.1542514
   > tan(30)
   [1] -6.405331
   >
   ```

7. **min() and max(): Minimum value and maximum value within a vector**

Example:

```
> x<-c(5,2,6,8,100,-5,96)
> min(x)
[1] -5
> max(x)
[1] 100
>
```

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

Example:

```
> x<-c(5,2,6,8,100,-5,96)
> which.min(x)
[1] 6
> which.max(x)
[1] 5
>
```

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

Example: 1

```
> a
[1] 10  9  7  5
> b
[1] 11  5  2  3
> c
[1] 12  4 15  1
> d
[1]  1 14  3 12
> pmin(a,b,c,d)
[1] 1 4 2 1
> pmax(a,b,c,d)
[1] 12 14 15 12
>
```

Example: 2

```
> A<-matrix(c(10,41,5,21,4,81,31,6,91), nrow=3)
> B<-matrix(c(1,4,7,2,5,8,3,6,9), nrow=3)
> A
     [,1] [,2] [,3]
[1,]   10   21   31
[2,]   41    4    6
[3,]    5   81   91
> B
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> pmin(A,B)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    4    6
[3,]    5    8    9
> pmax(A,B)
     [,1] [,2] [,3]
[1,]   10   21   31
[2,]   41    5    6
[3,]    7   81   91
>
```

10. sum() and prod(): Sum and product of the elements of a vector

Example:

```
> x
[1]   5   2   6   8 100  -5  96
> sum(x)
[1] 212


> y<-c(2,3,4)
> y
[1] 2 3 4
> prod(y)
[1] 24
>
```

11. cumsum() and cumprod(): Cumulative sum and product of the elements of a vector

Example:

```
> x
[1]   5   2   6   8 100  -5  96
> cumsum(x)
[1]   5   7  13  21 121 116 212
 > y
[1] 2 3 4
> cumprod(y)
[1]  2  6 24
>
```

12. round(), floor(), and ceiling(): Round to the closest integer, to the closest integer below, and to the closest integer above

Example: round()

```
> a<-5.63
> round(a)
[1] 6
> b<-5.33
> round(b)
[1] 5
>
```

```
Example:  floor()
> a
[1] 5.63
> floor(a)
[1] 5
> b
[1] 5.33
> floor(b)
[1] 5
>
```

Example: ceiling()

```
> a
[1] 5.63
> ceiling(a)
[1] 6
> b
[1] 5.33
> ceiling(b)
[1] 6
>
```

13. factorial(): Factorial function evaluates the factorial of a given number

Example:

```
> factorial(5)
[1] 120
>
```

B. Extended Example: Calculating a Probability

R language provides the facility to calculate the probability. Suppose we have *n* independent events, and the *i*th event has the probability *pi* of occurring.
What is the probability of exactly one of these events occurring?
Suppose first that *n* = 3 and our 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 $pA(1 - pB)(1 - pC)$, and so on.

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)
}
```

Output

```
>a<-c(2,4,6)
>exactlyone(a)
[1] 68
```

C. Cumulative Sums and Products

As mentioned, the functions cumsum() and cumprod() return cumulative sums and products.

```
> x <- c(12,5,13)
> cumsum(x)
[1] 12 17 30
> cumprod(x)
[1] 12 60 780
```

In x, the sum of the first element is 12, the sum of the first two elements is 17, and the sum of the first three elements is 30.

The function cumprod() works the same way as cumsum(), but with the product instead of the sum.

D. Minima and Maxima

There is quite a difference between min() and pmin(). min() function applies on vector and the pmin() function applies on the matrix

```
> A<-matrix(c(10,41,5,21,4,81,31,6,91), nrow=3)
> B<-matrix(c(1,4,7,2,5,8,3,6,9), nrow=3)
> A
     [,1] [,2] [,3]
[1,]   10   21   31
[2,]   41    4    6
[3,]    5   81   91
> B
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> pmin(A,B)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    4    6
[3,]    5    8    9
>  pmax(A,B)
     [,1] [,2] [,3]
[1,]   10   21   31
[2,]   41    5    6
[3,]    7   81   91
>
```

E. Calculus

R also has some calculus capabilities, including symbolic differentiation and numerical integration, as you can see in the following example.

$$\frac{d}{dx}e^{x^2} = 2xe^{x^2}$$

$$\int_0^1 x^2\,dx \approx 0.3333333$$

R-code for the above equation is as follows

```
> 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
```

: Functions for Statistical Distributions (UNIT -5)

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 Common R 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()

Example:
Let's 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.

Sorting

Ordinary numerical sorting of a vector can be done with the sort() function.
Example:

```
> x <- c(13,5,12,5)
> sort(x)
[1] 5 5 12 13
> x
[1] 13 5 12 5
```

Note that x itself did not change, in keeping with R's functional language.

The order() function return the indices of the sorted values in the original vector.
Example:

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

This means that
x[2] is the smallest value in x,
x[4] is the second smallest,
x[3] is the third smallest, and so on.

rank() function returns the rank of each element of a vector.
Example:

```
> x <- c(13,5,12,5)
> rank(x)
[1] 4.0 1.5 3.0 1.5
```

This says that 13 had rank 4 in x; that is, it is the fourth smallest.
The value 5 appears twice in x, with those two being the first and second smallest, so the rank 1.5 is assigned to both. Optionally, other methods of handling ties can be specified.

Linear Algebra Operations on Vectors and Matrices

Multiplying a vector by a scalar works directly, as you saw earlier. Here's another example:

```
> y
[1] 1 3 4 10
> 2*y
[1] 2 6 8 20
```

crossprod() function compute the inner product (or dot product) of two vectors.
Example:

```
> crossprod(1:3,c(5,12,13))
[,1]
[1,] 68
```

Here 1:3 mean 1,2,3. The first element 1 is multiplied with 5, second element 2 is multiplied with 12 and so on.

```
(1 * 5)  +  (2 * 12)  + (3 * 13)  =  68
```

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

$$x1 + x2 = 2$$
$$-x1 + x2 = 4$$

Its matrix form is as follows:

Here's the R- code:

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

In that second call to solve(), the lack of a second argument signifies that we simply wish to compute the inverse of the matrix.
Here are a few other linear algebra functions:

- `t()`: Matrix transpose
- `qr()`: QR decomposition
- `chol()`: Cholesky decomposition
- `det()`: Determinant
- `eigen()`: Eigenvalues/eigenvectors
- `diag()`: Extracts the diagonal of a square matrix
- `sweep()`: Numerical analysis sweep operations

The sweep() function

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.

```
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> sweep(m,1,c(1,4,7),"+")
     [,1] [,2] [,3]
[1,]    2    3    4
[2,]    8    9   10
[3,]   14   15   16
```

The first arguments to sweep() is m which is a matrix to be modified.
Second argument is margin which specifies the row or column to be modified first.
Third argument is a vector for manipulate the matrix m.
Fourth argument is operator to be applied on matrix.

A. Extended Example: Vector Cross Product

Let's consider the vector cross products. The definition is very simple: The cross product of vectors ($x1, x2, x3$) and ($y1, y2, y3$) in three dimensional space is a new three-dimensional vector, as shown in the following Equation

$$(x2y3 - x3y2, -x1y3 + x3y1, x1y2 - x2y1)$$

This can be expressed compactly as the expansion along the top row of the determinant, as shown below

Here, the elements in the top row are missing values (NA). The point is that the cross product vector can be computed as a sum of sub determinants. For example, the first component in Equation , $x2y3 - x3y2$, is easily seen to be the determinant of the submatrix obtained by deleting the first row and first column in matrix.

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

Our need to calculate subdeterminants—that is determinants of submatrices—fits perfectly with R, which excels at specifying submatrices.

This suggests calling det() on the proper submatrices, as follows:

```
xprod <- function(x,y) {
m <- rbind(c(NA,NA,NA),x,y)
xp <- c(0,0,0)
for (i in 1:3)
xp[i] <-   -(-1)^i * det(m[2:3,-i])
return(xp)
}

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

B. Extended Example: Finding Stationary Distributions of Markov Chains

A Markov chain is a random process in which we move among various *states*, in a "memoryless" fashion, whose definition need not concern us here. The state could be the number of jobs in a queue, the number of items stored in inventory, and so on. We will assume the number of states to be finite.

As a simple example, consider a game in which we toss a coin repeatedly and win a dollar whenever we accumulate three consecutive heads.

Our state at any time *i* will the number of consecutive heads we have so far, so our state can be 0, 1, or 2. (When we get three heads in a row, our state reverts to 0.)

The central interest in Markov modeling is usually the long-run state distribution, meaning the long-run proportions of the time we are in each state. In our coin-toss game, we can use the code we'll develop here to calculate that distribution, which turns out to have us at states 0, 1, and 2 in proportions 57.1%, 28.6%, and 14.3% of the time. Note that we win our dollar if we are in state 2 and toss a head, so $0.143 \times 0.5 = 0.071$ of our tosses will result in wins.

Since R vector and matrix indices start at 1 rather than 0, it will be convenient to relabel our states here as 1, 2, and 3 rather than 0, 1, and 2. For example, state 3 now means that we currently have two consecutive heads. Let *pij* denote the *transition probability* of moving from state *i* to state *j* during a time step. In the game example, for instance, $p23 = 0.5$, reflecting the fact that with probability 1/2, we will toss a head and thus move from having one consecutive head to two. On the other hand, if we toss a tail while we are in state 2, we go to state 1, meaning 0 consecutive heads; thus $p21 = 0.5$.

We are interested in calculating the vector $\pi = (\pi 1, ..., \pi s)$, where πi is the long-run proportion of time spent at state i, over all states i. Let *P* denote the transition probability

matrix whose *i*th row, *j*th column element is *pij* . Then it can be shown that π must satisfy Equation 8.4,

$\pi = \pi P$ (8.4)
which is equivalent to Equation 8.5:

$(I - PT)\pi = 0$ (8.5)
Here *I* is the identity matrix and *PT* denotes the transpose of *P*.
Any single one of the equations in the system of Equation 8.5 is redundant. We thus eliminate one of them, by removing the last row of $I-P$ in Equation 8.5. That also means removing the last 0 in the 0 vector on the right-hand side of Equation 8.5.
But note that there is also the constraint shown in Equation 8.6.

In matrix terms, this is as follows:

where 1*n* is a vector of *n* 1s. So, in the modified version of Equation 8.5, we replace the removed row with a row of all 1s and, on the right-hand side, replace the removed 0 with a 1. We can then solve the system.
All this can be computed with R's solve() function, as follows:

```
findpi1 <- function(p) {
n <- nrow(p)
imp <- diag(n) - t(p)
imp[n,] <- rep(1,n)
rhs <- c(rep(0,n-1),1)
pivec <- solve(imp,rhs)
return(pivec)
}
```

Here are the main steps:
1. Calculate $I - PT$ in line 3. Note again that diag(), when called with a scalar argument, returns the identity matrix of the size given by that argument.
2. Replace the last row of *P* with 1 values in line 4.
3. Set up the right-hand side vector in line 5.
4. Solve for π in line 6.

Set Operations

R includes some handy set operations, including these:

- union(x,y): Union of the sets x and y
- intersect(x,y): Intersection of the sets x and y
- setdiff(x,y): Set difference between x and y, consisting of all elements of x that are not in y
- setequal(x,y): Test for equality between x and y
- c %in% y: Membership, testing whether c is an element of the set y

- choose(n,k): Number of possible subsets of size k chosen from a set of size n
- combn(): Generates combinations

Here are some simple examples of using these functions:

```
> x <- c(1,2,5)
> y <- c(5,1,8,9)
> union(x,y)
[1] 1 2 5 8 9
> intersect(x,y)
[1] 1 5
> setdiff(x,y)
[1] 2
> setdiff(y,x)
[1] 8 9
> setequal(x,y)
[1] FALSE
> setequal(x,c(1,2,5))
[1] TRUE
> 2 %in% x
[1] TRUE
> 2 %in% y
[1] FALSE
> choose(5,2)
[1] 10
```

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

```
> c32 <- combn(1:3,2)
> c32
[,1] [,2] [,3]
[1,] 1 1 2
[2,] 2 3 3
> class(c32)
[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). The function also allows you to specify a function to be called by combn() on each combination. For example, we can find the sum of the numbers in each subset, like this:

```
> combn(1:3,2,sum)
[1] 3 4 5
```

The first subset, {1,2}, has a sum of 2, and so on.

INPUT/OUTPUT : Accessing the Keyboard and Monitor

R provides several functions for accesssing the keyboard and monitor. Here, we'll look at the scan(), readline(), print(), and cat() functions.

A. Accessing the Keyboard

i. Using the scan() Function

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

```
> v <- scan("")
1: 12 5 13
4: 3 4 5
7: 8
8:
Read 7 items
>v
[1] 12 5 13 3 4 5 8
```

Note that we are prompted with the index of the next item to be input, and we signal the end of input with an empty line.

.

ii. Using the readline() Function

If you want to read in a single line from the keyboard..

```
> w <- readline()
abc de f
>w
[1] "abc de f"
```

Typically, readline() is called with its optional prompt, as follows:

```
>inits<- readline("Enter your Branch: ")
Enter your Branch: CSE
>inits
[1] "CSE"
```

iii. Using the readLines() Function

If you want to read in a multiple lines from the keyboard.

```
> x<-readLines()
hello
how
are
you

>#(Press ctrl+z to save and quit)
> x
[1] "hello" "how"   "are"   "you"
```

B. Accessing the Monitor

i. The print() function

print() function prints the data on the monitor

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

Recall that 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"
```

ii. The cat() function

cat() function prints the data on the monitor

```
>cat("abc\n")
abc
```

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.

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

If you don't want the spaces, set "sep" argument 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:

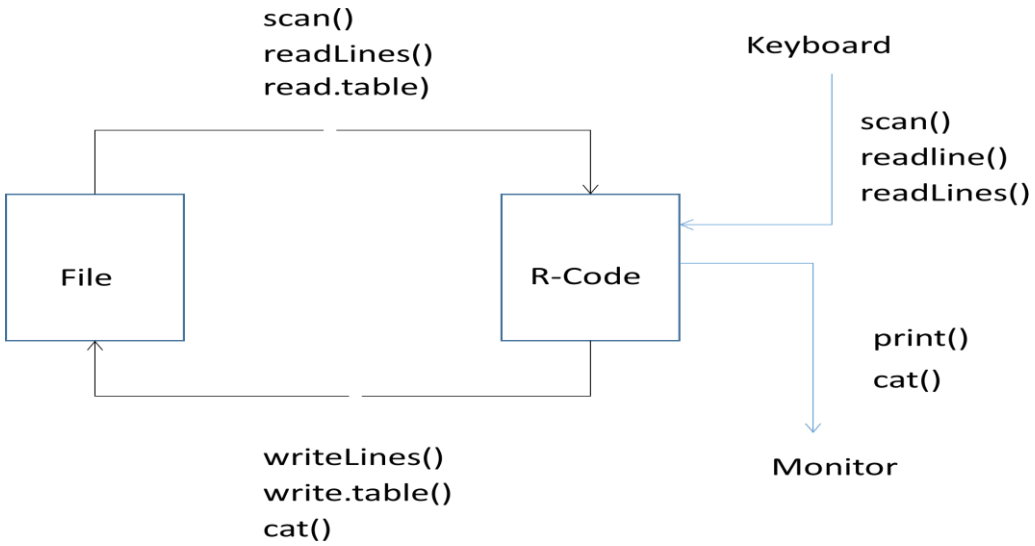
```
>cat(x,"abc","de\n",sep="\n")
1
2
3
abc
de
```

You can even set sep to be a vector of strings, like this:

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

: Reading and Writing Files

The details of functions involved in the reading and writing files



A. Reading data from a file

i. Introduction to Connections

Connection is R's term for a fundamental mechanism used in various kinds of I/O operations. Here, it will be used for file access. The connection is created by calling file(), url(), or one of several other R functions.
There are two modes in file() function. They are 1. Read mode. 2. Write mode

Connecting the file with read mode

Suppose the file z1.txt has following code

Ravi 25
Nani 22
Babu 23

```
> c <- file("z1.txt","r")
> readLines(c)
Ravi      25
Nani      22
Babu      23


>readLines(c,n=1)
[1] " Ravi     25"

>readLines(c,n=1)
[1] " Nani     22"

>readLines(c,n=1)
[1] " Babu  23"
Input/Output

>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.

We can detect EOF in our code: (Let the code file name is read.r)

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

```
> source("read.r")
[1] "Ravi 25"
[1] "Nani 22"
[1] "Babu 13"
[1] "reached the end"
```

Rewinding File

If we wish to "rewind"—to start again at the beginning of the file—we can use seek():

```
> c <- file("z1","r")
>readLines(c,n=2)
[1] "Ravi 25"
[2] "Nani 28"
```

```
>seek(con=c,where=0)
[1] 16
>readLines(c,n=1)
[1] "Ravi 25"
```

The argument where=0 in seek() means that the cursor will be placed at the beginning of the file.

ii. Using the readLines() Function

If you want to read in a multiple lines from the file.
Suppose z.txt file has the following contents

hello
how
are
you

Now we can read the data from z.txt is as follows

```
>c<-file("z.txt","r")
> x<-readLines(c)
hello
how
are
you
```

iii. Using read.table() function

The read.table() function used to read matrix or data frame from file. Suppose the file *z.txt* contains the following data.

```
name  age
Ravi  25
Nani  22
Babu  23
```

We can read the z.txt file as

```
>z<- read.table("z.txt")
>z
[1]  name   age
[2] Ravi     25
[3] Nani     22
[4] Babu     23
```

Here first line is header but it prints just like rows. We can specify header as

```
> z <- read.table("z",header=TRUE)
    name    age
[1] Ravi     25
[2] Nani     28
[3] Babu     19
```

iv. Using the scan() Function to read data from file

You can use scan() to read in a vector, whether numeric or character, from a file or the keyboard.
Suppose we have files named *z1.txt*, *z2.txt*, *z3.txt*, and *z4.txt*.

The z1.txt file contains the following:
123
4 5
6
The z2.txt file contents are as follows:
123
4.2 5
6
The z3.txt file contains this:
abc
de f
g
And finally, the z4.txt file has these contents:
abc
123 6
y

Using the scan() function to read the data

```
>scan("z1.txt")
Read 4 items
[1] 123 4 5 6
```

Here we got a vector of four integers

```
---------------------------------------
>scan("z2.txt")
Read 4 items
[1] 123.0 4.2 5.0 6.0
>scan("z3.txt")
```

```
Error in scan(file, what, nmax, sep, dec, quote, skip, nlines,
na.strings, :
```

Here scan() expected 'a real', got 'abc', since one number was non integral, the others were shown as floating-point numbers, too.

```
-------------------------------------
>scan("z3.txt",what="")
Read 4 items
[1] "abc" "de" "f" "g"
```

Here z3.txt contains the character data so scan() function need what argument. What=" " means, the scan() function can read character data.

```
-------------------------------------
>scan("z4.txt",what="")
Read 4 items
[1] "abc" "123" "6" "y"
```

Here z4.txt contains the character and numerical data so scan() function need "what" argument. What=" " means, the scan() function can read character and numerical data.

Reading Matrix through scan() function

For instance, say the file *x.txt* contains a 5-by-3 matrix, stored row-wise:

```
1 0 1
1 1 1
1 1 0
1 1 0
0 0 1
```

We can read it into a matrix this way:

```
> A<-matrix(scan("x.txt"),nrow=5, byrow=TRUE)
Read 15 items
> A
     [,1] [,2] [,3]
[1,]    1    0    1
[2,]    1    1    1
[3,]    1    1    0
[4,]    1    1    0
[5,]    0    0    1
>
```

B. Writing data to a file

i. write.table() function

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)
>d
kids ages
1 Jack 12
2 Jill 10
```

```
>write.table(d,"kds.txt")
The file kds.txt will now have these contents:
"kids" "ages"
"1" "Jack" 12
"2" "Jill" 10
```

ii. writeLines() function

The writeLines() function used to write the data to the file.
In file connection, we must specify "w" to indicate you are writing to the file.

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

The file x.txt will be created with these contents:
abc
de
f

iii. cat() function

The function cat() can used to write to a file,
Example:

```
>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

We can write multiple fields to the file as follows.

```
>cat(file="v",1,2,"xyz\n")
[1] 1 2 xyz
```