UNIT-I R PROGRAMMING

R programming introduction: R is a scripting language for statistical data manipulation, statistical analysis, graphics representation and reporting. It was inspired by, and is mostly compatible with, the statistical language S developed by AT&T. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team.

The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions. R allows integration with the procedures written in the C, C++, .Net, Python or FORTRAN languages for efficiency.

R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac.

FEATURES OF R: R is a programming language and software environment for statistical analysis, graphics representation and reporting. The following are the important features of R

- R is a well-developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.
- R has an effective data handling and storage facility.
- **It incorporates features found in object-oriented and functional programming languages**.
- R provides a suite of operators for calculations on arrays, lists, vectors and matrices.
- R provides a large, coherent and integrated collection of tools for data analysis.
- R provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.
- R provides special data types that includes the Numerical, Integer , Character, Logical, Complex
- R provides very powerful data structures it contains Vectors, Matrices, List, Arrays, Data frames, Classes.
- Because R is open source software, it's easy to get help from the user community. Also, a lot of new functions are contributed by users, many of whom are prominent statisticians.

As a conclusion, R is world's most widely used statistics programming language.

How to Run R: R operates in two modes: *interactive* and *batch mode*.

1. Interactive mode: The one typically used is interactive mode. In this mode, you type in commands, R displays results, you type in more commands, and so on.

2. Batch mode: Batch mode does not require interaction with the user. It's useful for production jobs, such as when a program must be run periodically; say once per day, because you can automate the process.

R Command Prompt: Once you have R environment setup, then it's easy to start your R command prompt by just typing the following command at your command prompt −

```
C:\Users\CSELAB>R
```

This will launch R interpreter and you will get a prompt **>** where you can start typing your program as follows −

```
>myString<-"WELCOME R PROGRAMMING"
>print(myString)
[1]"WELCOME R PROGRAMMING"
```

Here first statement defines a string variable myString, where we assign a string "Hello, World!" and then next statement print() is being used to print the value stored in variable myString.

Object orientation : can be explained by example. Consider statistical regression. When you perform a regression analysis with other statistical packages, such as SAS or SPSS, you get a mountain of output on the screen. By contrast, if you call the lm() regression function in R, the function returns an object containing all the results—the estimate coefficients, their standard errors, residuals, and so on. You then pick and choose, programmatically, which parts of that object to extract. You will see that R's approach makes programming much easier, partly because it offers a certain uniformity of access to data.

Functional Programming:

As is typical in functional programming languages, a common theme in R programming is avoidance of explicit iteration. Instead of coding loops, you exploit R's functional features, which let you express iterative behavior implicitly. This can lead to code that executes much more efficiently, and it can make a huge timing difference when running R on large data sets.

The functional programming nature of the R language offers many advantages:
• Clearer, more compact code
• Potentially much faster execution speed
• Less debugging, because the code is simpler
• Easier transition to parallel programming.

Comments: Comments are like helping text in your r program and they are ignored by the interpreter while executing your actual program. single comment is written using # in the beginning of the statement as follows −

```
# My first program in R Programming
```

R does not support multi-line comments but you can perform a trick which is something as follows −

```
if(FALSE){
"This is a demo for multi-line comments and it should be put inside either a single
    OR double quote"
}
myString<-"Hello, World!" print(myString)
```

Variables in R
Variables are used to store data, whose value can be changed according to our need. Unique name given to variable (function and objects as well) is identifier.
Rules for writing Identifiers in R

1. Identifiers can be a combination of letters, digits, period (.) and underscore (_).
2. It must start with a letter or a period. If it starts with a period, it cannot be followed by a digit.
3. Reserved words in R cannot be used as identifiers.

Valid identifiers in R
total, Sum, .fine.with.dot, this_is_acceptable, Number5
Invalid identifiers in R
tot@l, 5um, _fine, TRUE, .0ne

Best Practices

Earlier versions of R used underscore (_) as an assignment operator. So, the period (.) was used extensively in variable names having multiple words. Current versions of R support underscore as a valid identifier but it is good practice to use period as word separators. For example,
a.variable.name is preferred over "a_variable_name" or alternatively we could use camel case as aVariableName

Constants in R

Constants, as the name suggests, are entities whose value cannot be altered. Basic types of constant are numeric constants and character constants.

Numeric Constants

All numbers fall under this category. They can be of type integer, double or complex. This can be checked with the typeof() function. Numeric constants followed by $_L$ are regarded as integer and those followed by $_i$ are regarded as $_{complex}$.

>typeof(5) [1] "double"	>typeof(5L) [1] "integer"	>typeof(5i) [1] "complex"

Character Constants: Character constants can be represented using either single quotes (') or double quotes (") as delimiters.

```
 > 'example'
[1] "example"

>typeof("5")
 [1] "character"
```

Built-in Constants

Some of the built-in constants defined in R along with their values is shown below.

```
> LETTERS
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"

>letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"

>pi
 [1] 3.141593

> month.name
 [1] "January"   "February"  "March"    "April"    "May"      "June"      [7] "July"     "August"
"September" "October"   "November"  "December"
>month.abb
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

But it is not good to rely on these, as they are implemented as variables whose values can be changed.

```
>pi
[1] 3.141593

>pi<- 56
>pi [1] 56
```

Maths in R Programming: R Having Number Of Mathematical Calculations in R .It Contains Given Below

Program	Output	Program	Output
>a<-99 >print(a)	[1] 99	> z<-c(1:10)	>z [1] 1 2 3 4 5 6 7 8 9 10
> 2+2	[1] 4	> z1<-c(-5:5)	> z1 [1] -5 -4 -3 -2 -1 0 1 2 3 4 5
> 3-2	[1] 1	>seq(-5,5)	[1] -5 -4 -3 -2 -1 0 1 2 3 4 5
> 6/2	[1] 3	> seq(-5,5,by=2)	[1] -5 -3 -1 1 3 5
> 5*5	[1] 25	>seq(from=1,to=10,by=10)	[1] 1
>log(10)	[1] 2.302585	>seq(from=1,to=100,by=10)	[1] 1 11 21 31 41 51 61 71 81 91
>exp(1)	[1] 2.718282	>seq(from=0,to=100,by=10)	[1] 0 10 20 30 40 50 60 70 80 90 100
>exp(2)	[1] 7.389056	>rep("rao",5)	[1] "rao" "rao" "rao" "rao" "rao"
>exp(2)*exp(1)	[1] 20.08554	>rep(1:5,4)	[1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
>log10(6)	[1] 0.7781513	>rep("R programming",9)	[1] "R programming" "R programming" "R programming" "R programming" "R programming" "R programming" "R programming" "R programming" "R programming"
>log(16,4)	[1] 2		
>pi	[1] 3.141593		
>sin(2)	[1] 0.9092974		
>floor(9)	[1] 9		
>floor(12.5)	[1] 12		
>ceiling(13.5)	[1] 14		
>round(15.5)	[1] 16		
>abs(-1)	[1] 1		
>sqrt(100)	[1] 10		

R Session:

Let's make a simple data set (in R parlance, a *vector*) consisting of the numbers 1, 2, and 4, and name it x:

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

The standard assignment operator in R is **<-**. You can also use =, but this is discouraged, as it does not work in some special situations. Note that there are no fixed types associated with variables. Here, we've assigned a vector to x, but later we might assign something of a different type to it. The c stands for *concatenate*. Here, we are concatenating the numbers 1, 2, and 4. More precisely, we are concatenating three one-element vectors that consist of those numbers. This is because any number is also considered to be a one-element vector.

Now we can also do the following:

> q <- c(x,x,8) which sets q to (1,2,4,1,2,4,8) (yes, including the duplicates).

Now let's confirm that the data is really in x. To print the vector to the screen, simply type its name. If you type any variable name (or, more generally, any expression) while in interactive mode, R will print out the value of that variable (or expression). Programmers familiar with other languages such as Python will find this feature familiar. For our example, enter this:

```
> x
[1] 1 2 4
```

Yep, sure enough, x consists of the numbers 1, 2, and 4.

Individual elements of a vector are accessed via [ ]. Here's how we can print out the third element of x:

```
> x[3]
[1] 4
```

As in other languages, the selector (here, 3) is called the *index* or *subscript*. Those familiar with ALGOL-family languages, such as C and C++, should note that elements of R vectors are indexed starting from 1, not 0.

Subsetting is a very important operation on vectors. Here's an example: > x <- c(1,2,4)

```
> x[2:3]
[1] 2 4
> mean(x)
[1] 2.333333
> sd(x)
[1] 1.527525
```

Introduction to Functions: As in most programming languages, the heart of R programming consists of writing *functions*. A function is a group of instructions that takes inputs, uses them to compute other values, and returns a result.

As a simple introduction, let's define a function named oddcount(), whose purpose is to count the odd numbers in a vector of integers.

```
# counts the number of odd integers in x
> oddcount <- function(x) {
+ k <- 0 # assign 0 to k
+ for (n in x) {
+ if (n %% 2 == 1) k <- k+1 # %% is the modulo operator + }
+ return(k)
+ }
> oddcount(c(1,3,5))
[1] 3
> oddcount(c(1,2,3,7,9))
[1] 4
```

Arithmetic Operators: These operators are used to carry out mathematical operations like addition and multiplication. Here is a list of arithmetic operators available in R.

An example run

```
> x <-5
> y <-16
>x+y
[1]21
>x-y
[1]-11
>x*y
[1]80
>y/x
[1]3.2
>y%/%x
[1]3
>y%%x
[1]1
>y^x
[1]1048576
```

	Arithmetic Operators in R
Operator	**Description**
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponent
%%	Modulus (Remainder from division)
%/%	Integer Division (no Fractional part)

Relational Operators: Relational operators are used to compare between values. Here is a list of relational operators available in R.

Relational Operators in R			
Operator	**Description**	**==**	**Equal to**
<	**Less than**	**!=**	**Not equal to**
>	**Greater than**		
<=	**Less than or equal to**		
>=	**Greater than or equal to**		

An example run

```
> x <-5
> y <-16
```

> x<y [1] TRUE	> x>y [1] FALSE	> x<=5 [1] TRUE	> y>=20 [1] FALSE	>x !=5 [1] FALSE	>y==16 [1] TRUE

Operation on Vectors: The above mentioned operators work on vectors. The variables used above were in fact single element vectors. We can use the function c() (as in concatenate) to make vectors in R. All operations are carried out in element-wise fashion. Here is an example.

```
> x <-c(2,8,3)
> y <-c(6,4,1)
```

>x+y [1]8 12 4	> x>y [1]FALSE TRUE TRUE

When there is a mismatch in length (number of elements) of operand vectors, the elements in shorter one is recycled in a cyclic manner to match the length of the longer one. R will issue a warning if the length of the longer vector is not an integral multiple of the shorter vector.

```
> x <-c(2,1,8,3)
> y <-c(9,4)
>x+y# Element of y is recycled to 9,4,9,4
[1]115177

>x-1# Scalar 1 is recycled to 1,1,1,1
[1]1072
>x+c(1,2,3)
[1]33114 Warning message:
In x +c(1,2,3): longer object length is not a multiple of shorter object length
```

Logical Operators: Logical operators are used to carry out Boolean operations like AND, OR etc.

	Logical Operators in R
Operator	**Description**
!	Logical NOT
&	Element-wise logical AND
&&	Logical AND
|	Element-wise logical OR
||	Logical OR

Operators & and | perform element-wise operation producing result having length of the longer operand. But && and || examines only the first element of the operands resulting into a single length logical vector. Zero is considered FALSE and non-zero numbers are taken as TRUE. An example run.

```
> x <-c(TRUE,FALSE,0,6) > y <-c(FALSE,TRUE,FALSE,TRUE)
>!x
[1]FALSE  TRUETRUE FALSE
>x&y
[1] FALSE FALSEFALSE  TRUE
>x&&y
[1] FALSE
>x|y
[1]  TRUETRUE FALSE  TRUE
 > x||y
[1] TRUE
```

Assignment Operators: These operators are used to assign values to variables.

Assignment Operators in R	
Operator	Description
<-, <<-, =	Leftwards assignment
->, ->>	Rightwards assignment

The operators `<-` and `=` can be used, almost interchangeably, to assign to variable in the same environment. `<<-` is used for assigning to variables in the parent environments (more like global assignments). The rightward assignments, although available are rarely used.

```
> x <-5
>x
[1]5
> x =9
>x
[1]9
>10-> x
>x
[1]10
```

R Program to Add Two Vectors:
We can add two vectors together using the `+` operator. One thing to keep in mind while adding (or other arithmetic operations) two vectors together is the recycling rule. If the two vectors are of equal length then there is no issue. But if the lengths are different, the shorter one is recycled (repeated) until its length is equal to that of the longer one. This recycling process will give a warning if the longer vector is not an integral multiple of the shorter one.

Source Code

```
>x
[1]3 6 8
 >y
[1]2 9 0

> x + y
[1]5 15 8

> x +1# 1 is recycled to (1,1,1)
[1]4 7 9

> x +c (1, 4)# (1,4) is recycled to (1,4,1) but warning issued
[1]4 10 9 Warning message:
In x +c(1,4): longer object length is not a multiple of shorter object length
```

As we can see above the two vectors `x` and `y` are of equal length so they can be added together without difficulty. The expression `x + 1` also works fine because the single `1` is recycled into a vector of three `1`'s. Similarly, in the last example, a two element vector is recycled into a three element vector. But a warning is issued in this case as 3 is not an integral multiple of 2.

Data types in R: To make the best of the R language, you'll need a strong understanding of the basic data types and data structures and how to operate on those.

- character
- numeric (real or decimal)
- integer
- logical
- complex

Example	**Type**
"a", "swc"	character
2, 15.5	numeric
2 (Must add a `L` at end to denote integer)	integer
TRUE, FALSE	logical
1+4i	complex

```
TRUE
[1] TRUE
>class(TRUE)
[1] "logical"
>class(FALSE)
[1] "logical"
> T
[1] TRUE
> F
[1] FALSE
>class(2)
[1] "numeric"
>class(2L)
[1] "integer"
>class(2i)
[1] "complex"
>class("i love R programming")
[1] "character"
>class(2.5)
[1] "numeric"
>is.numeric(2)
[1] TRUE
>is.numeric(2.5) [1] TRUE
>is.numeric(22i)
[1] FALSE
>is.integer(22i)
[1] FALSE
>is.integer(2)
[1] FALSE
>class("integer is numeric numeric is not always integer")
[1] "character"
```

DATA STRUCTURES (or) DATA OBJECTS IN R :A **data structure** is a specialized format for organizing Data and storing data**. R Data structures are also called as data objects.**

R has many data structures. These include

- VECTOR
- MATRIX
- LIST
- ARRAYS
- DATA FRAME

R Programming Vector: Vector is a basic data structure in R. Vectors contain element of the same type. Different data types available in R are logical, integer, double, character, complex and raw. A vector's type can be checked with the typeof() function. Another important property of a vector is its length. This is the number of elements in the vector and can be checked with the function length(). Scalars are actually vectors with length equal to one.

Creating(Declaration of) a Vector: Vectors are generally created using the c() function. Since, a vector must have elements of the same type, this function will try and coerce elements to the same type, if they are different. Coercion is from lower to higher types from logical to integer to double to character.

```
> x <-c(1,5,4,9,0)
>typeof(x)
[1]"double"
>length(x)
```

```
[1]5

> x <-c(1,5.4, TRUE,"hello")
>x
[1]"1""5.4""TRUE""hello"
>typeof(x)
[1]"character"
```

If we want to create a vector of consecutive numbers, the : operator is very helpful. More complex sequences can be created using the seq() function, like defining number of points in an interval, or the step size.

```
> x <-1:7; x
[1]1234567

> y <-2:-2; y
[1]210-1-2

>seq(1,3,by=0.2)# specify step size
[1]1.01.21.41.61.82.02.22.42.62.83.0

>seq(1,5,length.out=4)# specify length of the vector
[1]1.0000002.3333333.6666675.000000
```

<u>Accessing Elements in Vector (Vector Indexing)</u>

Elements of a vector can be accessed using vector indexing. The vector used for indexing can be logical, integer or character vector.

Using integer vector as index

Vector index in R starts from 1, unlike most programming languages where index start from 0. We can use a vector of integers as index to access specific elements. We can also use negative integers to return all elements except that those specified. But we cannot mix positive and negative integers while indexing and real numbers, if used, are truncated to integers.

```
>x
[1]0 2 4 6 8 10

>x[3]# access 3rd element
[1]4

>x[c(2,4)]# access 2nd and 4th element
[1]26

>x[-1]# access all but 1st element
[1]246810

>x[c(2,-4)]# cannot mix positive and negative integers
Errorin x[c(2,-4)]: only 0's may be mixed with negative subscripts

>x[c(2.4, 3.54)]    # real numbers are truncated to integers [1] 2 4
```

Using logical vector as index

When we use a logical vector for indexing, the position where the logical vector is TRUE is returned. This useful feature helps us in filtering of vector as shown below.

```
>x[c(TRUE, FALSE, FALSE, TRUE)]
[1]-33

>x[x<0]# filtering vectors based on conditions
[1]-3-1
```

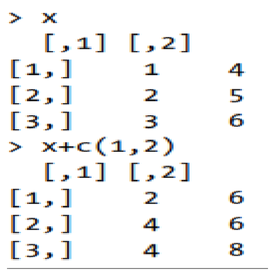
```
>x[x>0]
[1]3
```
In the above example, the expression x>0 will yield a logical vector (FALSE, FALSE, FALSE, TRUE) which is then used for indexing.

Using character vector as index

This type of indexing is useful when dealing with named vectors. We can name each elements of a vector.
```
> x <-c("first"=3,"second"=0,"third"=9)
>names(x)
[1]"first""second""third"

>x["second"] second
0

>x[c("first","third")] first third  39
```

Modifying a Vector

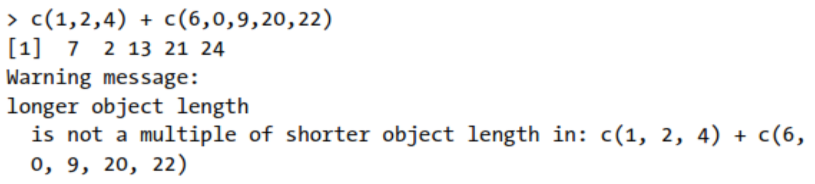
We can modify a vector using the assignment operator. We can use the techniques discussed above to access specific elements and modify them. If we want to truncate the elements, we can use reassignments.

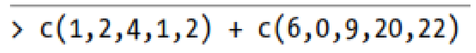

```
> x
  [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> x+c(1,2)
  [,1] [,2]
[1,]    2    6
[2,]    4    6
[3,]    4    8
```

```
>x
[1]-3-2-1012

>x[2]<-0; x        # modify 2nd element
[1]-30-1012

>x[x<0]<-5; x   # modify elements less than 0
[1]505012

> x <-x[1:4]; x      # truncate x to first 4 elements
[1]5050
```
Deleting a Vector: We can delete a vector by simply assigning a NULL to it.

```
>x
[1]-3-2-1012
> x <- NULL
>x
NULL
>x[4]
NULL
```
Recycling

When applying an operation to two vectors that requires them to be the same length, R automatically recycles, or repeats, the shorter one, until it is long enough to match the longer one.


```
> c(1,2,4) + c(6,0,9,20,22)
[1]  7  2 13 21 24
Warning message:
longer object length
  is not a multiple of shorter object length in: c(1, 2, 4) + c(6,
  0, 9, 20, 22)
```
The shorter vector was recycled, so the operation was taken as follows:


```
> c(1,2,4,1,2) + c(6,0,9,20,22)
```

Here is another example

Here, x, as a 3-by-2 matrix, is also a six-element vector, which in R is stored column by column. x is the same as c(1,2,3,4,5,6). We added a two-element vector to this six-element one, so our added vector needed to be repeated twice to make six elements. In other words we are doing x + c(1,2,1,2,1,2).

Common Vector Operations

Vector Arithmetic and Logical Operations:

In R we can perform the arithmetic and logical operations on element-wise. Means the elements of first vector will perform arithmetic operation on the first element of the second vector.

```
> x <- c(1,2,4)
> x + c(5,0,-1)
[1] 6 2 3
```

When we multiply two vectors, we will be surprised with the result. Let's see.

```
> x * c(5,0,-1)
[1]  5  0 -4
```

Here we can observe multiplication operation is done element by element.

Generating Useful Vectors with the : Operator

The colon operator ':' is used to generate the useful vector consisting of a range of numbers.

```
>x<-c(5:8)
```

Now x is a vector which has 5,6,7,8 as the elements of the vector.

Generating Vector Sequences with seq()

A generalization of : is the seq() (or *sequence*) function, which generates a sequence in arithmetic progression. For instance, whereas 3:8 yields the vector (3,4,5,6,7,8), with the elements spaced one unit apart (4 - 3=1,5- 4=1, and so on), we can make them, say, three units apart, as follows:

```
> seq(from=12,to=30,by=3)
[1] 12 15 18 21 24 27 30
```

Repeating Vector Constants with rep()

The rep() (or *repeat*) function allows us to conveniently put the same constant into long vectors. The call form is rep(x,times), which creates a vector of *times * length(x)* elements—that is, times copies of x. Here is an example:

```
> x <- rep(8,4)
> x
[1] 8 8 8 8
> rep(c(5,12,13),3)
[1]  5 12 13  5 12 13  5 12 13
> rep(1:3,2)
[1] 1 2 3 1 2 3
```

Using all() and any()

All and Any function on vector will return whether the given condition is satisfied or not

```
> x <- 1:10
> any(x > 8)
[1] TRUE
> any(x > 88)
[1] FALSE
> all(x > 88)
[1] FALSE
> all(x > 0)
[1] TRUE
```

It first evaluates x > 8, yielding this:

```
(FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,TRUE,TRUE)
```

Vectorized Operations:

Vector In, Vector Out: You can pass Vector to a function as parameter and get vector from a function as return type.
Lets take one example

```
> w <- function(x) return(x+1)
> w(u)
[1] 6 3 9
```

Here w is a function which takes x as input parameter and returns x+1 as the return type. Where x is vector and x+1 is also vector.

Vector In, Matrix Out : You can pass vector to a function as parameter and display returned vector as Matrix form.

```
> matrix(z12(x),ncol=2)
     [,1] [,2]
[1,]    1    1
[2,]    2    4
[3,]    3    9
[4,]    4   16
[5,]    5   25
[6,]    6   36
[7,]    7   49
[8,]    8   64
```

NA and NULL Values: Both NULL and NA are Reserved words

Using NA: In many of R's statistical functions, we can instruct the function to skip over any missing values, or NA's

```
> x <- c(88,NA,12,168,13)
> x
[1]  88  NA  12 168  13
> mean(x)
[1] NA
> mean(x,na.rm=T)
[1] 70.25
> x <- c(88,NULL,12,168,13)
> mean(x)
[1] 70.25
```

Here x is a vector of 5 elements and we are giving only 4 values 2nd value is missing and can be specified using NA means value is there but right now not available. If you want to perform any mathematical operations like mean(), median() you need to specify another parameter na.rm()=T. so, that mean is calculated by removing the NA values.

Using NULL: Null is used to represent the null object in R language. Null is used mainly used to build up a vector in a loop initially you need to initialize the vector to NULL and later build the vector.

```
# build up a vector of the even numbers in 1:10
> z <- NULL
> for (i in 1:10) if (i %%2 == 0) z <- c(z,i)
> z
[1]  2  4  6  8 10
```

Filtering :
Filtering allows the user to extract the vector elements that satisfy certain condition.

Generating Filtering Indices:

```
> z <- c(5,2,-3,8)
> w <- z[z*z > 8]
> w
[1] 5  -3  8
```

In this example z is a vector having 4 elements now i want to extract the elements from z where square of the number must be greater than 8 and assigned the result vector to w.let's see how r will evaluate this . z*z>8 Returns TRUE FALSE TRUE TRUE now z[TRUE FALSE TRUE TRUE] displays the elements related to that index.

Filtering with the subset() Function:

Filtering can be done with subset().subset when applied to vector it works similar to the normal filtering with indexing the main difference we can observe when there is NA as the component of the vector.

```
> x <- c(6,1:3,NA,12)
> x
[1]  6  1  2  3 NA 12
> x[x > 5]
[1]  6 NA 12
> subset(x,x > 5)
[1]  6 12
```

Observe in the adjacent example that x is a vector of 6 elements out of which one is NA if we filter with normal indexing we will get NA also to the sub-vector. But with subset function we can get only elements by filtering NA also.

The Selection Function which():

```
> z <- c(5,2,-3,8)
> which(z*z > 8)
[1] 1 3 4
```

If we want to find out the indexes where our condition satisfied instead of the values of the vector we will use which() function. Here in this example positions 1 3 4 satisfied our condition.

Testing Vector Equality:

The native approach for vector equality won't work because when you compare two vectors with == you will get a vector with Boolean values.

```
> x <- 1:3
> y <- c(1,3,4)
> x == y
[1]  TRUE FALSE FALSE
```

Instead if you want to get whether the vectors are equal or not we will use all() function with condition then we will get as follows

```
> all(x == y)
[1] FALSE
```

by using all function we will get whether the two vectors are equal or not.

We can also use identical function to check the equality between two vectors.

```
> identical(x,y)
[1] FALSE
```

Vector Element Names

We can give names for the vector elements by using the names function

```
> x <- c(1,2,4)
> names(x)
NULL
> names(x) <- c("a","b","ab")
> names(x)
[1] "a"  "b"  "ab"
> x
 a  b ab
 1  2  4
```

We can remove the names from the vector by assigning NULL values.
names(x)<-NULL.
>x
[1] 1 2 4

Matrix and Array

Matrix is a vector with two additional attributes: the number of rows and columns.

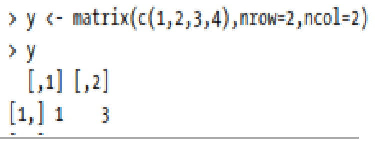
Creating Matrices: A Matrix is created using the **matrix()** function.

Dimension of the matrix can be defined by passing appropriate value for arguments nrow and ncol. Providing value for both the dimension is not necessary.

Syntax: matrix(data, nrow, ncol, byrow)

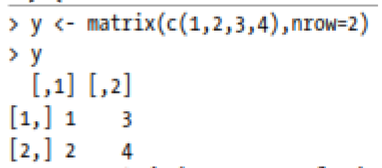
Following is the description of the parameters used −

- **data** is the input vector which becomes the data elements of the matrix.
- **nrow** is the number of rows to be created.
- **ncol** is the number of columns to be created.
- **byrow** is a logical clue. If TRUE then the input vector elements are arranged by row.



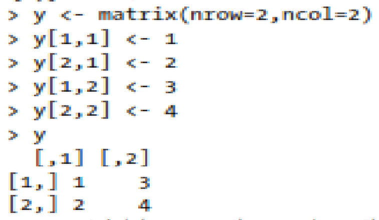
```
> y <- matrix(c(1,2,3,4),nrow=2,ncol=2)
> y
  [,1] [,2]
[1,] 1    3
```

Now y is a vector with 1,2,3,4 as elements having 2 rows and 2 columns.



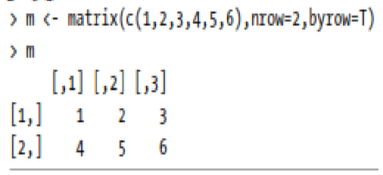
```
> y <- matrix(c(1,2,3,4),nrow=2)
> y
  [,1] [,2]
[1,] 1    3
[2,] 2    4
```

If one of the dimensions is provided, the other is inferred from length of the data.



```
> y <- matrix(nrow=2,ncol=2)
> y[1,1] <- 1
> y[2,1] <- 2
> y[1,2] <- 3
> y[2,2] <- 4
> y
  [,1] [,2]
[1,] 1    3
[2,] 2    4
```

We can also assign the elements into the matrix by using indexing. But before that we need to declare that as matrix.



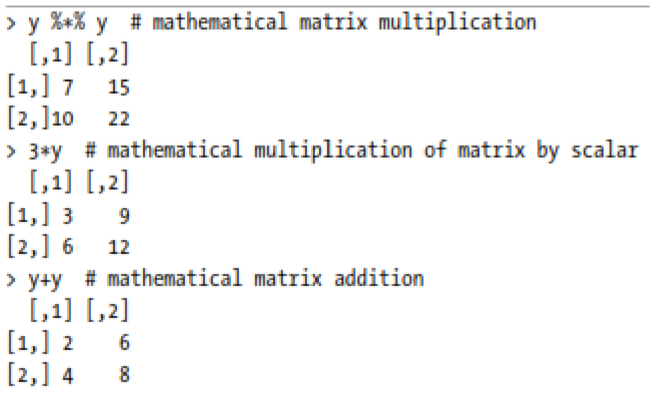
```
> m <- matrix(c(1,2,3,4,5,6),nrow=2,byrow=T)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

By default matrix is column wise. If we want to declare as the row wise we need to mention it while declaring 'byrow=T'.

<u>Matrix Operations:</u>

Performing Linear Algebra Operations on Matrices
We can perform various types of algebraic operations on matrices such as multiplication, subtraction, addition, division, vector multiplication.

Operator	operation
%*%	Matrix multiplication
*	Multiplication by scalar
+	Addition of matrix
-	Subtraction



```
> y %*% y  # mathematical matrix multiplication
  [,1] [,2]
[1,] 7   15
[2,]10   22
> 3*y  # mathematical multiplication of matrix by scalar
  [,1] [,2]
[1,] 3    9
[2,] 6   12
> y+y  # mathematical matrix addition
  [,1] [,2]
[1,] 2    6
[2,] 4    8
```

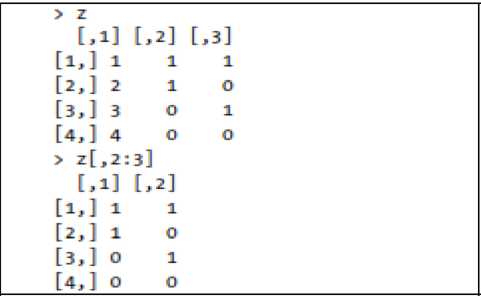
Here in 1st example we multiply Y matrix with Y. Matrix multiplication is entirely different between multiplication of matrix with scalar.
All arithmetic operations are done element wise operations.

Matrix Indexing:

Matrix indexing is nothing but accessing matrix elements by specifying the position using rows and columns.
We can access elements of a matrix using the square bracket indexing method. Elements can be accessed as var[row, column].

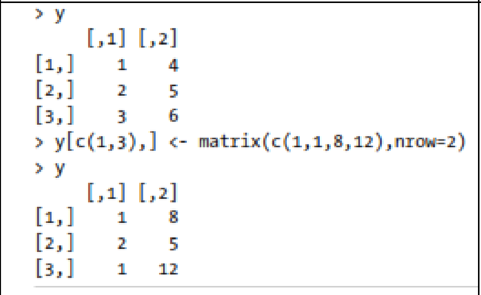
Using integer vector as index

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.



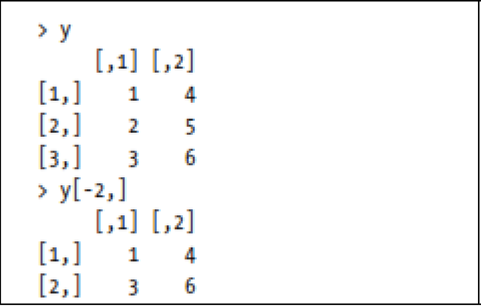
```
> z
  [,1] [,2] [,3]
[1,] 1    1    1
[2,] 2    1    0
[3,] 3    0    1
[4,] 4    0    0
> z[,2:3]
  [,1] [,2]
[1,] 1    1
[2,] 1    0
[3,] 0    1
[4,] 0    0
```

In this example if we want to access all rows and only 2,3 columns data. We can specify like this --- in the rows index we leave space because it specifies all rows and 2:3 in column index position.
We can also use vector as index means you can also specify column as vector like z[,c(2,3)].here c(2,3) is a vector.



```
> y
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> y[c(1,3),] <- matrix(c(1,1,8,12),nrow=2)
> y
     [,1] [,2]
[1,]    1    8
[2,]    2    5
[3,]    1   12
```

We can also assign values to sub matrix.

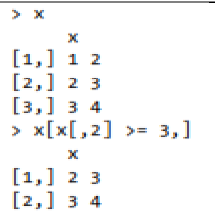


```
> y
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> y[-2,]
     [,1] [,2]
[1,]    1    4
[2,]    3    6
```

Negative subscripts used with index eliminates certain elements

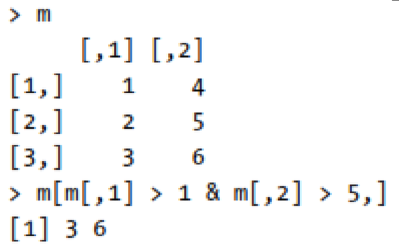
Filtering on Matrices:

Filtering on matrices can be done same as filtering of vector. The elements which satisfies some condition must be retrieved.



```
> x
     x
[1,] 1 2
[2,] 2 3
[3,] 3 4
> x[x[,2] >= 3,]
     x
[1,] 2 3
[2,] 3 4
```

Here in this example we want the data from the matrix
X[,2]>=3 means all rows having elements in 2 column greater than or equal to 3.
The above statement returns 2 & 3rd rows .
Now, x[x[,2]>=3,] means 2nd and 3rd rows all column data must be retrieved.



```
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> m[m[,1] > 1 & m[,2] > 5,]
[1] 3 6
```

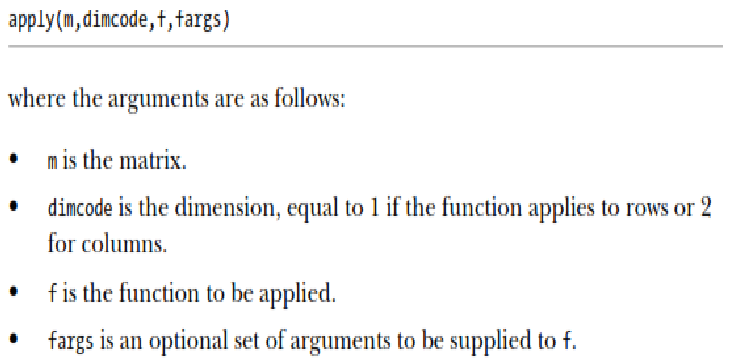
Here in this example
m[,1]>1 means all rows where 1st column >1 returns 2nd and 3rd rows all columns.
m[,2]>5 means all rows where 2nd column>5 returns 3rd row all columns.
Both above conditions & operation result in 3rd rows.
At the end m[m[,1]>1 & m[,2]>5,] returns 3rd row all columns.

Applying Functions to Matrix rows and columns:

One of the most important feature of R language is applying the user defined or predefined functions on rows or columns of a matrix.

Using apply() function:

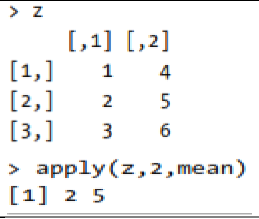
The general form of apply function to matrix is



```
apply(m,dimcode,f,fargs)
```

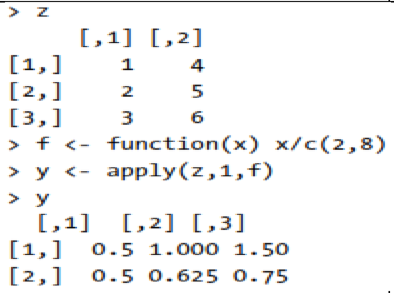
where the arguments are as follows:

- m is the matrix.
- dimcode is the dimension, equal to 1 if the function applies to rows or 2 for columns.
- f is the function to be applied.
- fargs is an optional set of arguments to be supplied to f.



```
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> apply(z,2,mean)
[1] 2 5
```

For example if we want to find mean for matrix z via column wise. We need to go for apply function.
Apply(z,2,mean)::: means apply mean function on matrix 'z' via column wise. Here we got a vector.



```
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> f <- function(x) x/c(2,8)
> y <- apply(z,1,f)
> y
   [,1]  [,2] [,3]
[1,]  0.5 1.000 1.50
[2,]  0.5 0.625 0.75
```

We can apply user defined function on the matrix also by specifying the matrix on which you want to apply, the dimensions(row wise or column wise) on which you want to apply the user defined function.

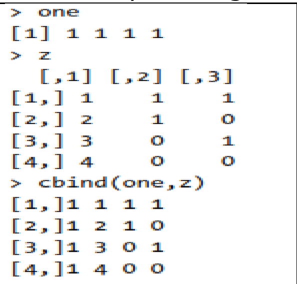
** Note: when you use the apply() on matrix the dimensions of the result matrix(2*3) may not be what you expect(3*2) because by default matrix is column wise that's why you need to transpose the result matrix by `> t(apply(z,1,f))` so that you will get exact dimensions in the resultant matrix

Adding and deleting Matrix Rows and Columns:

Technically matrices are fixed length and dimensions. However, Matrices can be modified by reassigning to get the same effect of addition and deletion of rows and columns.

Changing the size of the matrix:

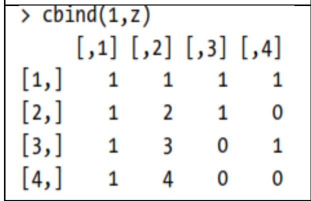
Changing the size of the matrix is done by adding rows and columns, deleting the rows and columns to the already existing matrix .This can be achieved by rbind() and cbind() functions.



```
> one
[1] 1 1 1 1
> z
  [,1] [,2] [,3]
[1,] 1    1    1
[2,] 2    1    0
[3,] 3    0    1
[4,] 4    0    0
> cbind(one,z)
[1,]1 1 1 1
[2,]1 2 1 0
[3,]1 3 0 1
[4,]1 4 0 0
```

In this example cbind() function is used to add a column to already existing matrix. At first we have a matrix z with 4*3 dimensions now we want to add one column to z to make it 4*4 dimensions. Now we have a vector 'one' which have 4 elements. Add vector 'one' to matrix 'z' using cind(one,z).

**Note: If there are less number of elements the vector you want to add means if there are only 3 elements(1,2,3) in vector 'one' R will through an warning message and the vector elements will be recycled (1,2,3,1) will be added to the matrix.

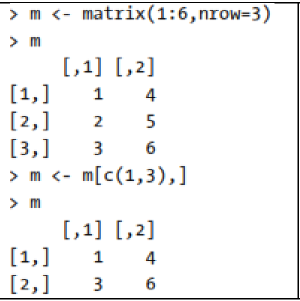


```
> cbind(1,z)
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    1    2    1    0
[3,]    1    3    0    1
[4,]    1    4    0    0
```

We can also bind a single value to matrix like this. Here we are trying to column bind value 1 with matrix. But In R' language there is no concept of scalar. All single elements will be treated as single element vectors. Now we are column binding the vector with matrix z. Vector is recycle to form the order of matrix ie., 4*1 since matrix z has 4 rows and then added the resultant vector(1,1,1,1) to the matrix z.

You can also add rows to the matrix using rbind() similar to cbind().

Deletion of rows and columns can be done by eliminating the unwanted rows and then assign to the same matrix.

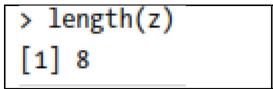


```
> m <- matrix(1:6,nrow=3)
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> m <- m[c(1,3),]
> m
     [,1] [,2]
[1,]    1    4
[2,]    3    6
```

Here matrix 'm' has 3*2 dimensions initially. Now I am reassigning by eliminating unwanted rows. Here I am extracting rows 1,3 from existing matrix 'm' and reassigning to 'm'.

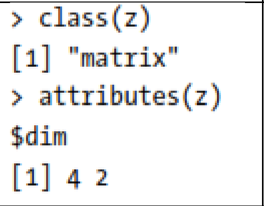
Differences between Vector and Matrices

A matrix is nothing but a vector but with two additional attributes: number of rows and columns.
Let us check the class of the matrix.



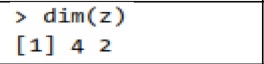
```
> length(z)
[1] 8
```

We can get the length of the matrix using length() function.



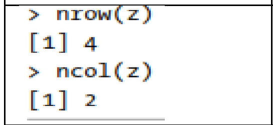
```
> class(z)
[1] "matrix"
> attributes(z)
$dim
[1] 4 2
```

In addition to the length matrix has addition attributes. To get that we use attributes() function to get the dimensions of the matrix i.e., no of rows and columns.
To know the class of the matrix we use class() function.



```
> dim(z)
[1] 4 2
```

We can get the number of rows and columns of the matrix by just dim() function. Here in z there are 4 rows and 2 columns are there.

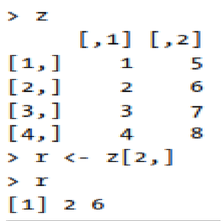


```
> nrow(z)
[1] 4
> ncol(z)
[1] 2
```

In order to get either number of rows or number of columns we have ncol() and nrow() functions.

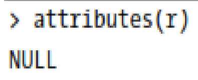
Unintended Dimension Reduction:

When ever we are extracting single row or column from the matrix R will convert that single row or column matrix into vector and display it like a vector instead of matrix. This type of missing the essence of our data in matrix is called dimension reduction.



```
> z
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
> r <- z[2,]
> r
[1] 2 6
```

Here z is a matrix and we are extracting single row from the matrix and assign to r. Now , display r it will display as vector manner which is not acceptable by the user. To confirm whether we got a vector or not check the attributes of 'r' it will display nothing. If 'r' is a matrix attributes function must return its dimension but unfortunately we got 'NULL'.

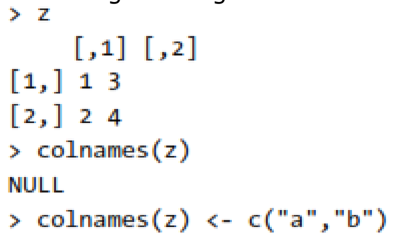


```
> attributes(r)
NULL
```

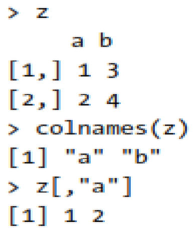
**** NOTE:** To avoid dimension reduction in matrix we have to use parameter drop='FALSE' like this `> r <- z[2,, drop=FALSE]`
Now check attributes of r it will return dimensions of matrix.

Naming the rows and columns of the matrix:

We can give assign names to the rows and columns of the matrix using rownames() and colnames().



```
> z
     [,1] [,2]
[1,] 1 3
[2,] 2 4
> colnames(z)
NULL
> colnames(z) <- c("a","b")
```



```
> z
     a b
[1,] 1 3
[2,] 2 4
> colnames(z)
[1] "a" "b"
> z[,"a"]
[1] 1 2
```

We can also get the names of the rows and columns by passing the matrix name to these functions.

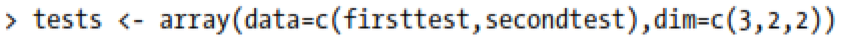
Arrays

In R language if we want to store data in layer manner additional to rows and columns there is another data structure called arrays. Arrays have third dimension in addition to rows and columns to store the data in layer manner.

Consider students and test scores. Say each test consists of two parts, so we record two scores for a student for each test. Now suppose that we have two tests, and to keep the example small, assume we have only three students. Here's the data for the first test and second test:

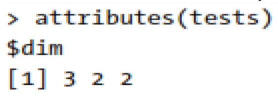
```
> firsttest
    [,1] [,2]
[1,] 46 30
[2,] 21 25
[3,] 50 48
> secondtest
    [,1] [,2]
[1,] 46 43
[2,] 41 35
[3,] 50 49
```

We can declare array using array(). Now declare array with firsttest as 1st layer, secondtest as 2nd layer as follows.



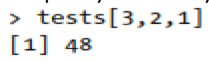
```
> tests <- array(data=c(firsttest,secondtest),dim=c(3,2,2))
```

In the dim=c(3,2,2), we are specifying two layers ,each consisting of three rows and two columns and these becomes dimensions for the array.



```
> attributes(tests)
$dim
[1] 3 2 2
```

We can access the elements of the array by using the three indexes. First is for row, second is for column and third is to specify which layer.



```
> tests[3,2,1]
[1] 48
```

Here we are accessing element form tests of 3rd row,2nd column and 1st layer i.e., 48.

If we want to print all the elements of the array at a time we can display array by calling its name.

```
> tests
, , 1

     [,1] [,2]
[1,]   46   30
[2,]   21   25
[3,]   50   48

, , 2

     [,1] [,2]
[1,]   46   43
[2,]   41   35
[3,]   50   49
```

tests array is displayed in layer format. (, ,1)represents first layer, (, ,2) represents second layer**.**

LISTS

R's list structure is similar to that of vector but with different data types. It is similar to that of C structures.

Creating Lists:

Technically speaking list is a vector. In Normal vectors elements cannot be broken down into smaller components. Whereas lists can be broken down (recursive list).

For example if we want to store data of employee, we wish to store name, salary, union member or not. This is the perfect place to use the list data structure.

We can declare list using list() function.

```
j <- list(name="Joe", salary=55000, union=T)
```

We need to give the names to the list members because we need to identify which values belongs to which one. Without names also we can declare but its not clear what is the value is for.

We can print the list by full or component.

```
> j
$name
[1] "Joe"

$salary
[1] 55000

$union
[1] TRUE
```

Or

```
> j$sal
[1] 55000
```

Since list is also vector we can declare list like vectors using following syntax.

```
> z <- vector(mode="list")
> z[["abc"]] <- 3
> z
$abc
[1] 3
```

List Operations:

List Indexing:

We can access list elements in number of ways: using component name, using index number.

```
> j$salary
[1] 55000
> j[["salary"]]
[1] 55000
```

Using component name: we can access list elements by specifying the component name with $ symbol or specifying the component name with in [[ ]] (double square bracket).

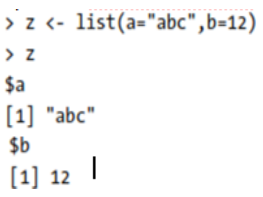
```
> j[[2]]
[1] 55000
```

Using component index: we can access list elements by specifying the index number with in the [ ](square brackets) or [[ ]] (double square brackets).if you specify with single square brackets list will display with names and double square brackets is list will display without names.(Technically speaking single brackets return a list or a sub list).

Adding and Deleting elements to list:

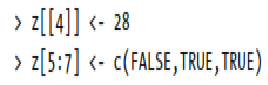
We can add elements to the list by assigning to list name followed by $ symbol followed by name of the component

Syntax is listname$comname<-value



```
> z <- list(a="abc",b=12)
> z
$a
[1] "abc"
$b
[1] 12
```

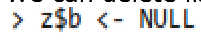
let us consider z is a list with 2 components a and b. if we want to add another component c to list z. we just need to specify like this. `> z$c <- "sailing"` here we add component c to list z with value "sailing".



```
> z[[4]] <- 28
> z[5:7] <- c(FALSE,TRUE,TRUE)
```

We can also a component from vector also. Here we added components to the list from a boolean vector.

We can delete list components by assigning NULL to that particular component.



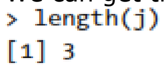
```
> z$b <- NULL
```

with this command component b will be deleted from the list.

**NOTE: upon deleting the component from list all the list elements will be moved up by 1.

Getting the size of the list:

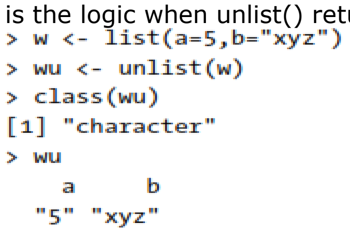
We can get the size of the list by using the length(). We will get the number of component in the list.



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

Accessing list Components and Values:

If the components of the list have names and we are trying to access the components then we will get the names of the list through indexing. Unlist() function returns the list to a vector. A list is nothing but a vector with different data types how can a vector hold different data types when an unlist() returns. Here is the logic when unlist() returns a vector R chooses character string as mode of the vector. .



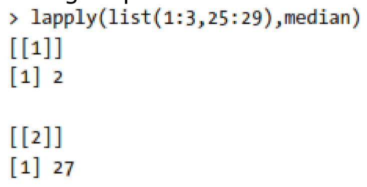
```
> w <- list(a=5,b="xyz")
> wu <- unlist(w)
> class(wu)
[1] "character"
> wu
    a     b
  "5" "xyz"
```

If you want only values you can use names(wu)<-NULL. Now wu contains only values of the list.

You can also use wu<-unname(wu) to remove the names in the vector.

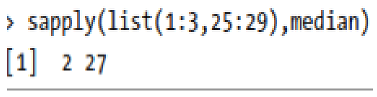
Applying Functions to lists:

Using lapply() and sapply(): These two functions will work similar to apply() in matrices. The lapply() calling a specified function on each component of the list or a vector and returning list.



```
> lapply(list(1:3,25:29),median)
[[1]]
[1] 2

[[2]]
[1] 27
```

Here lapply() function applied on the two vectors 1:3 and 25:29 and return the result as list.

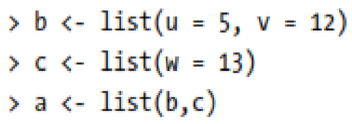


```
> sapply(list(1:3,25:29),median)
[1]  2 27
```

If we want the result in a Vector not in a list manner use sapply() function. Same median function applied on the same list using sapply() returns a vector format.

Recursive Lists:

We can declare a list with in another list this type of list is called recursive lists. List can be recursive.



```
> b <- list(u = 5, v = 12)
> c <- list(w = 13)
> a <- list(b,c)
```

Here b, c are declared as list having 2 and 1 components respectively. 'a' is also a list whose components are b & c means list with in a list. Now a is called as recursive list.

```
> a
[[1]]
[[1]]$u
[1] 5

[[1]]$v
[1] 12


[[2]]
[[2]]$w
[1] 13


> length(a)
[1] 2
```

When we display list ‘a’ we will get output like this.
We can also get the length of the list as 2 since list ‘a’ has only 2 components namely ‘b’ & ‘c’.

```
> c(list(a=1,b=2,c=list(d=5,e=9)))
$a
[1] 1

$b
[1] 2

$c
$c$d
[1] 5

$c$e
[1] 9
```

We can also declare recursive list using c()

DATA FRAMES

Data frame is like a matrix, with a two dimensional rows and columns structure. However it differ from matrix that each column may have different modes. For instance one column may contains number another may contains character strings. A data frame is a list, with the components of that list being equal-length vectors.

Creating Data Frames:
Data Frames are created by using data.frame(). While creating data frame from character vector we need to specify not to convert vector to factor. This can be accomplish by specifying stringAsFactor=FALSE.

```
> kids <- c("Jack","Jill")
> ages <- c(12,10)
> d <- data.frame(kids,ages,stringsAsFactors=FALSE)
> d  # matrix-like viewpoint
  kids ages
1 Jack   12
2 Jill   10
```

Here kids and ages are two vectors of character and numeric data types respectively. We are creating data frame from these vectors each as column in the data frame. Since kids is a character vector we need to specify stringAsFactor=FALSE.

Accessing Data Frames:
Data frame is nothing but collection of lists we can access the data frames same as we access from list like index value or component name.

```
> d[[1]]
[1] "Jack" "Jill"
> d$kids
[1] "Jack" "Jill"
```

We can access the data frame using index number or component name. kids is the component of the data frame. We can access like d$kids(same as list accessing).we can access using index number like d[[1]] to access the first column.

As we told that data frames are matrix except that each column has different data type. We can access data from data frame same like matrix accessing using row number and column number.

```
> d[,1]
[1] "Jack" "Jill"
```

Here we are getting data from first column of all rows same like accessing data from matrix.

Matrix-Like Operations:

Various matrix operations also apply to data frames. Most notably and usefully, we can do filtering to extract various subdata frames of interest.

Extracting Subdata Frames

we can extract subdata frames by rows or columns.

```
> examsquiz[2:5,]
  Exam.1 Exam.2 Quiz
2    3.3      2  3.7
3    4.0      4  4.0
4    2.3      0  3.3
5    2.3      1  3.3
> examsquiz[2:5,2]
[1] 2 4 0 1
> class(examsquiz[2:5,2])
[1] "numeric"
> examsquiz[2:5,2,drop=FALSE]
  Exam.2
2      2
3      4
4      0
5      1
> class(examsquiz[2:5,2,drop=FALSE])
[1] "data.frame"
```

Here we are extracting the subdata frames from existing data frame by specifying the row numbers and column numbers as index. Same as in matrix while we are extracting the data if the sub matrix is single row or column R will convert it into vector. to avoid missing of the essence of matrix we specify the 'drop' parameter in indexing. Same is the case with data frame. In order to not to disturb the mode of the subdata frame to vector we need to specify the 'drop=false'.

We can also filter the data from data frame based on some condition like in matrix

```
> examsquiz[examsquiz$Exam.1 >= 3.8,]
   Exam.1 Exam.2 Quiz
3       4    4.0  4.0
9       4    3.3  4.0
11      4    4.0  4.0
14      4    0.0  4.0
16      4    3.7  4.0
```

Here we are extracting (filtering) data from data frame where rows having exam.1>=3.8.

Importance of NA values in data frame:

```
> x <- c(2,NA,4)
> mean(x)
[1] NA
> mean(x,na.rm=TRUE)
[1] 3
```

x is a vector having NA value and we performed mean on vector x then result will be NA since any mathematical function on data structure having NA values will result in NA only. To avoid that we need to specify na.rm=true while applying the function.

While dealing with data frames if you have NA value in the data frame when you apply any function or filtering on the data frame it may result in NA.

subset()

```
> subset(examsquiz,Exam.1 >= 3.8)
```

with the subset function we can skip the NA values in the data frame same like that of na.rm=true.

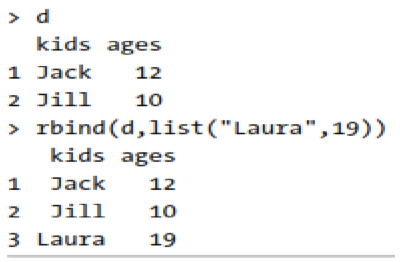
Sometimes we may need to remove all the rows having NA values to do that we need functions like complete.cases(). This function will remove all the incomplete rows in the given data frame. lets see an example.

```
> d4
     kids states
1    Jack     CA
2    <NA>     MA
3 Jillian     MA
4    John   <NA>
> complete.cases(d4)
[1]  TRUE FALSE  TRUE FALSE
> d5 <- d4[complete.cases(d4),]
> d5
     kids states
1    Jack     CA
3 Jillian     MA
```

Here d4 is a data frame which contains NA values in it. Now I want only rows which contains all the complete information. To get the records from d4 having complete information we need to apply complete.cases(d4). Give it as the row index for d4 so that it will set the row index with Boolean values.

Using rbind() and cbind() with data frames:

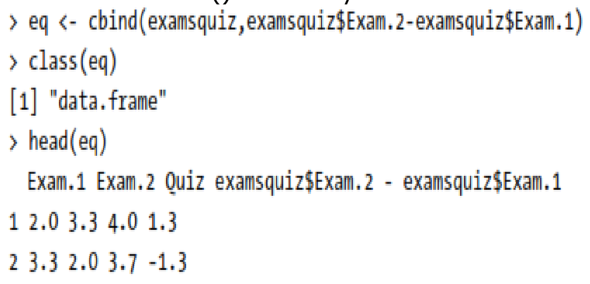
We can use rbind and cbind functions to modify the data frames. rbind is used to add new row to the existing data frame. Since data frame is collection of list we can add one more list to the data frame.



```
> d
  kids ages
1 Jack   12
2 Jill   10
> rbind(d,list("Laura",19))
   kids ages
1  Jack   12
2  Jill   10
3 Laura   19
```

d is a data frame. We are adding another list to d using rbind function.

we can use cbind() to modify the data frame



```
> eq <- cbind(examsquiz,examsquiz$Exam.2-examsquiz$Exam.1)
> class(eq)
[1] "data.frame"
> head(eq)
  Exam.1 Exam.2 Quiz examsquiz$Exam.2 - examsquiz$Exam.1
1 2.0 3.3 4.0 1.3
2 3.3 2.0 3.7 -1.3
```

Here we added a new column where the values are calculated from the existing columns without disturbing the class or mode of the data frame.

Merging Data Frames:

For example if we want apply predefined function mean on the matrix z

```
>a
[,1][,2][,3]
[1,]147
[2,]258
[3,]369

>class(a)
[1]"matrix"

>attributes(a)
$dim
[1]33

>dim(a)
[1]33
>matrix(1:9,nrow=3,ncol=3)
[,1][,2][,3]
[1,]147
[2,]258
[3,]369

># same result is obtained by providing only one dimension
>matrix(1:9,nrow=3)
[,1][,2][,3]
```

```
[1,]147
[2,]258
[3,]369
```

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,]123
[2,]456
[3,]789
```

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  147
Y  258
Z  369
```

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  147
R2  258
R3  369
```

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,]14
[2,]25
[3,]36

>rbind(c(1,2,3),c(4,5,6))
[,1][,2][,3]
[1,]123
[2,]456
```

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]123456
>class(x)
[1]"numeric"

>dim(x)<- c(2,3)
>x
[,1][,2][,3]
[1,]135
[2,]246
>class(x)
[1]"matrix"
```

Accessing Elements in Matrix

We can access elements of a matrix using the square bracket [indexing method. Elements can be accessed as var[row, column]. Here rows and columns are 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,]147
[2,]258
[3,]369

>x[c(1,2),c(2,3)]# select rows 1 & 2 and columns 2 & 3
[,1][,2]
[1,]47
[2,]58

>x[c(3,2),]# leaving column field blank will select entire columns [,1][,2][,3]
[1,]369
[2,]258

>x[,]# leaving row as well as column field blank will select entire matrix [,1][,2][,3]
[1,]147
[2,]258
[3,]369

>x[-1,]# select all rows except first
[,1][,2][,3]
[1,]258
[2,]369
```

One thing to notice here is that, if the matrix returned after indexing is a row matrix or column matrix, the result is given as a vector.

```
>x[1,]
[1]147
>class(x[1,]) [1]"integer"
```

This behavior can be avoided by using the argument drop=FALSE while indexing.

```
>x[1,,drop=FALSE]# now the result is a 1X3 matrix rather than a vector
[,1][,2][,3]
[1,]147
>class(x[1,,drop=FALSE])
[1]"matrix"
```

It is possible to index a matrix with a single vector. While indexing in such a way, it acts like a vector formed by stacking columns of the matrix one after another. The result is returned as a vector.

```
>x
[,1][,2][,3]
[1,]483
[2,]607
[3,]129

>x[1:4]
[1]4618

>x[c(3,5,7)]
[1]103
```

Using logical vector as index

Two logical vectors can be used to index a matrix. In such situation, rows and columns where the value is TRUE is returned. These indexing vectors are recycled if necessary and can be mixed with integer vectors.

```
>x
[,1][,2][,3]
[1,]483
[2,]607
[3,]129

>x[c(TRUE,FALSE,TRUE),c(TRUE,TRUE,FALSE)]
[,1][,2]
[1,]48
[2,]12

>x[c(TRUE,FALSE),c(2,3)]# the 2 element logical vector is recycled to 3 element vector
[,1][,2]
[1,]83
[2,]29
```

It is also possible to index using a single logical vector where recycling takes place if necessary.

```
>x[c(TRUE,FALSE)]
[1]41039
```

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]6879
```

```
>x[x%%2==0]# select even elements
[1]46802
```

Using character vector as index

Indexing with character vector is possible for matrix with named row or column. This can be mixed with integer or logical indexing.

```
>x
    A B C
[1,]483
[2,]607
[3,]129

>x[,"A"]
[1]461

>x[TRUE,c("A","C")]
    A C
[1,]43
[2,]67
[3,]19

>x[2:3,c("A","C")]
    A C
[1,]67
[2,]19
```

Modifying a Matrix

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

```
>x
[,1][,2][,3]
[1,]147
[2,]258
[3,]369

>x[2,2]<-10; x    # modify a single element
[,1][,2][,3]
[1,]147
[2,]2108
```

```
[3,]369

>x[x<5]<-0; x    # modify elements less than 5
[,1][,2][,3]
[1,]007
[2,]0108 [3,]069
```

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,]000
[2,]0106
[3,]789
```

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,]0071
[2,]01082
[3,]0693

>rbind(x,c(1,2,3))# add row
[,1][,2][,3]
[1,]007
[2,]0108
[3,]069
[4,]123

> x <-x[1:2,]; x    # remove last row
[,1][,2][,3]
[1,]007
[2,]0108
```

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

```
>x
[,1][,2][,3]
[1,]135
[2,]246

>dim(x)<- c(3,2); x    # change to 3X2 matrix
[,1][,2]
[1,]14
[2,]25
[3,]36

>dim(x)<- c(1,6); x    # change to 1X6 matrix
[,1][,2][,3][,4][,5][,6]
[1,]123456
```

Matrix Computations

Various mathematical operations are performed on the matrices using the R operators. The result of the operation is also a matrix.

The dimensions (number of rows and columns) should be same for the matrices involved in the operation.

Matrix Addition & Subtraction

# Create two 2x3 matrices. matrix1 <- matrix(c(3,9,-1,4,2,6),nrow=2) print(matrix1)

matrix2 <- matrix(c(5,2,0,9,3,4),nrow=2) print(matrix2)

# Add the matrices.
result<- matrix1 + matrix2 cat("Result of addition","\n") print(result)

# Subtract the matrices result<- matrix1 - matrix2 cat("Result of subtraction","\n") print(result)

When we execute the above code, it produces the following result −

```
     [,1] [,2] [,3]
[1,]    3   -1    2
[2,]    9    4    6
     [,1] [,2] [,3]
[1,]    5    0    3
[2,]    2    9    4
Result of addition
     [,1] [,2] [,3]
[1,]    8   -1    5
[2,]   11   13   10
Result of subtraction
     [,1] [,2] [,3]
[1,]   -2   -1   -1
[2,]    7   -5    2
```

Matrix Multiplication & Division

# Create two 2x3 matrices.
matrix1 <- matrix(c(3,9,-1,4,2,6),nrow=2) print(matrix1)

matrix2 <- matrix(c(5,2,0,9,3,4),nrow=2) print(matrix2)

# Multiply the matrices. result<- matrix1 * matrix2 cat("Result of multiplication","\n") print(result)

# Divide the matrices result<- matrix1 / matrix2 cat("Result of division","\n") print(result)

When we execute the above code, it produces the following result −

```
     [,1] [,2] [,3]
[1,]    3   -1    2
[2,]    9    4    6
     [,1] [,2] [,3]
```

```
[1,]    5    0    3
[2,]    2    9    4
Result of multiplication
     [,1] [,2] [,3]
[1,]   15    0    6
[2,]   18   36   24
Result of division
     [,1]      [,2]      [,3]
[1,]  0.6      -Inf 0.6666667
[2,]  4.5 0.4444444 1.5000000
```

List in R programming: List is a data structure having components of mixed data types. A vector having all elements of the same type is called atomic vector but a vector having elements of different type is called list. Following is an example of a list having three components each of different data type. We can check if it's a list with typeof() function and find its length using length().

```
>x
$a
[1]2.5

$b
[1] TRUE

$c
[1]123

>typeof(x)
[1]"list"

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

Creating a List

List can be created using the list() function.

```
> x <-list("a"=2.5,"b"=TRUE,"c"=1:3)
```

Here, we create a list x, of three components with data types double, logical and integer vector respectively. Its structure can be examined with the str() function.

```
>str(x)
List of 3
$ a:num2.5
$ b:logi TRUE
$ c:int[1:3]123
```

In this example, a, b and c are called tags which makes it easier to reference the components of the list. However, tags are optional. We can create the same list without the tags as follows. In such scenario, numeric indices are used by default.

```
> student<list(c(101:105),c("rao","dhanshika","moksha","sankar","gopi"),c(2000,5000,100
00,3000,12000))
> print(student)
[[1]]
```

```
[1] 101 102 103 104 105

[[2]]
[1] "rao"  "dhanshika" "moksha"  "sankar"    "gopi"

[[3]]
[1]  2000  5000 10000  3000 12000
> str(student)
List of 3
 $ : int [1:5] 101 102 103 104 105
 $ : chr [1:5] "rao" "dhanshika" "moksha" "sankar" ...
 $ : num [1:5] 2000 5000 10000 3000 12000
> names(student)<-c("sid","sname","ssal")
> print(student)
$sid
[1] 101 102 103 104 105

$sname
[1] "rao" "dhanshika" "moksha" "sankar" "gopi"

$ssal
[1]  2000  5000 10000  3000 12000

> str(student)
List of 3
 $ sid  : int [1:5] 101 102 103 104 105
 $ sname: chr [1:5] "rao" "dhanshika" "moksha" "sankar" ...
 $ ssal : num [1:5] 2000 5000 10000 3000 12000
```

#Update the data in list:we are updating address column ,first we update the data student list , next update column name .

```
>  student[4]<-" 'tenali','guntur','vijayawada','chennai','delhi'"
> str(student)
List of 4
 $ sid  : int [1:5] 101 102 103 104 105
 $ sname: chr [1:5] "rao" "dhanshika" "moksha" "sankar" ...
 $ ssal : num [1:5] 2000 5000 10000 3000 12000
 $      : chr " 'tenali','guntur','vijayawada','chennai','delhi'"
> names(student)<-c("sid","sname","ssal","saddress")
> str(student)
List of 4
 $ sid     : int [1:5] 101 102 103 104 105
 $ sname   : chr [1:5] "rao" "dhanshika" "moksha" "sankar" ...
 $ ssal    : num [1:5] 2000 5000 10000 3000 12000
 $ saddress: chr " 'tenali','guntur','vijayawada','chennai','delhi'"
```

> **Accessing Components in List**

Lists can be accessed in similar fashion to vectors. Integer, logical or character vectors can be used for indexing. Let us consider a list as follows.

```
> student$sname
```

```
[1] "rao"       "dhanshika" "moksha"    "sankar"    "gopi"
> student$sid
[1] 101 102 103 104 105
> student$ssal
[1]  2000  5000 10000  3000 12000
> student$saddress
[1] " 'tenali','guntur','vijayawada','chennai','delhi'"
> student(c[1:2])
Error: could not find function "student"
> student[c(1:2)]
$sid
[1] 101 102 103 104 105

$sname
[1] "rao"       "dhanshika" "moksha"    "sankar"    "gopi"
```

# index using integer vector

```
> student[c(3:2)]
$ssal
[1]  2000  5000 10000  3000 12000
$sname
[1] "rao"       "dhanshika" "moksha"    "sankar"    "gopi"
```

Delete the elememt temporary:

> student[c(-2)]   remove **sname** Display other Elements

```
$sid
[1] 101 102 103 104 105
$ssal
[1]  2000  5000 10000  3000 12000
$saddress
[1] " 'tenali','guntur','vijayawada','chennai','delhi'"
```

Remove sid from student display other elements

```
> student[c(-1)]
$sname
[1] "rao"       "dhanshika" "moksha"    "sankar"    "gopi"
$ssal
[1]  2000  5000 10000  3000 12000
$saddress
[1] " 'tenali','guntur','vijayawada','chennai','delhi'"
```

Remove saddress from student display other elements

```
> student[c(-4)]
$sid
[1] 101 102 103 104 105
$sname
[1] "rao"       "dhanshika" "moksha"    "sankar"    "gopi"
$ssal
[1]  2000  5000 10000  3000 12000
```

# index using logical vector

> student[c(T,F,F,T)]   display only true elements false elements are not display $sid

```
[1] 101 102 103 104 105
$saddress
```

```
[1] " 'tenali','guntur','vijayawada','chennai','delhi'"
```
> student[c(F,F,F,T)]
```
$saddress
[1] " 'tenali','guntur','vijayawada','chennai','delhi'"
```
# index using character vector
```
> student[c("sid","ssal")]
$sid
[1] 101 102 103 104 105
$ssal
[1]  2000  5000 10000  3000 12000
```

Deleting Component

We can delete a component by assigning NULL to it.

```
> student[2]<-NULL
> student[2]
$ssal
[1]  2000  5000 10000  3000 12000

> str(student)
List of 3
 $ sid     : int [1:5] 101 102 103 104 105
 $ ssal    : num [1:5] 2000 5000 10000 3000 12000
 $ saddress: chr " 'tenali','guntur','vijayawada','chennai','delhi'" > student[3]
$saddress
[1] " 'tenali','guntur','vijayawada','chennai','delhi'"
> student[3]<-NULL
> str(student)
List of 2
 $ sid : int [1:5] 101 102 103 104 105
 $ ssal: num [1:5] 2000 5000 10000 3000 12000
> rm(student)
> str(student)
Error in str(student) : object 'student' not found Student list removed from r source
```

Merging Lists:

You can merge many lists into one list by placing all the lists inside one list() function.

```
# Create two lists. list1 <-list(1,2,3)
list2 <-list("Sun","Mon","Tue")

# Merge the two lists.
merged.list<-c(list1,list2)

# Print the merged list. print(merged.list)
```

When we execute the above code, it produces the following result −

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

```
[[2]]
[1] 2

[[3]]
[1] 3

[[4]]
[1] "Sun"

[[5]]
[1] "Mon"

[[6]]
[1] "Tue"
```

Converting List to Vector

A list can be converted to a vector so that the elements of the vector can be used for further manipulation. All the arithmetic operations on vectors can be applied after the list is converted into vectors. To do this conversion, we use the **unlist()** function. It takes the list as input and produces a vector.

```
# Create lists. list1 <-list(1:5) print(list1)

list2 <-list(10:14) print(list2)

# Convert the lists to vectors.
v1 <-unlist(list1) v2 <-unlist(list2)
 print(v1) print(v2) # Now add the vectors result<- v1+v2 print(result)
```

When we execute the above code, it produces the following result −

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

[[1]]
[1] 10 11 12 13 14

[1] 1 2 3 4 5
[1] 10 11 12 13 14
[1] 11 13 15 17 19
```

Data Frames in R : Dataframe is collection of vectors are arranged in parllely in rows or columns

Syntax: Data.frame(v1,v2,v3,-------)

```
>y1<-c(1,2,3,4,NA)

> y2<-c(5,6,7,NA,8)

> y3<-c(9,10,NA,11,12)
> y4<-c(13,NA,14,15,16)

> y5<-c(17,18,19,20,21)

> rao<-data.frame(y1,y2,y3,y4,y5)

> print(rao)

  y1 y2 y3 y4 y5 1  1  5  9 13 17
2 2  6 10 NA 18
3 3  7 NA 14 19
4 4 NA 11 15 20
5 NA  8 12 16 21

> str(rao)

'data.frame':   5 obs. of  5 variables:

 $ y1: num  1 2 3 4 NA
 $ y2: num  5 6 7 NA 8
 $ y3: num  9 10 NA 11 12
 $ y4: num  13 NA 14 15 16
 $ y5: num  17 18 19 20 21
> s_rao<-stack(rao) > print(s_rao)    values indicator
1      1  y1
2      2  y1
3      3  y1
4      4  y1
5      NA  y1
6      5  y2
7      6  y2
8      7  y2
9      NA  y2
10     8  y2
11     9  y3
12     10  y3
13     NA  y3
14     11  y3
15     12  y3
16     13  y4
17     NA  y4
18     14  y4
19     15  y4
20     16  y4
21     17  y5
22     18  y5
23     19  y5
24     20  y5
```

```
25     21  y5
> unstack(s_rao)    y1 y2 y3 y4 y5 1  1  5  9 13 17
2  2  6 10 NA 18
3  3  7 NA 14 19
4  4 NA 11 15 20
5  NA  8 12 16 21
>cy<-cbind(rao,rao,rao,rao)

> cy
  y1 y2 y3 y4 y5 y1 y2 y3 y4 y5 y1 y2 y3 y4 y5 y1 y2 y3 y4 y5 1  1  5  9 13 17  1  5  9 13 17
1  5  9 13 17  1  5  9 13 17
2 2  6 10 NA 18  2  6 10 NA 18  2  6 10 NA 18  2  6 10 NA 18
3 3  7 NA 14 19  3  7 NA 14 19  3  7 NA 14 19  3  7 NA 14 19
4 4 NA 11 15 20  4 NA 11 15 20  4 NA 11 15 20  4 NA 11 15 20
5 NA  8 12 16 21 NA  8 12 16 21 NA  8 12 16 21 NA  8 12 16 21
> ry<-cbind(rao,rao,rao,rao)
> ry
  y1 y2 y3 y4 y5 y1 y2 y3 y4 y5 y1 y2 y3 y4 y5 y1 y2 y3 y4 y5 1  1  5  9 13 17  1  5  9 13 17
1  5  9 13 17  1  5  9 13 17
2 2  6 10 NA 18  2  6 10 NA 18  2  6 10 NA 18  2  6 10 NA 18
3 3  7 NA 14 19  3  7 NA 14 19  3  7 NA 14 19  3  7 NA 14 19
4 4 NA 11 15 20  4 NA 11 15 20  4 NA 11 15 20  4 NA 11 15 20
5 NA  8 12 16 21 NA  8 12 16 21 NA  8 12 16 21 NA  8 12 16 21
> ry<-rbind(rao,rao,rao,rao)
> ry    y1 y2 y3 y4 y5 1   1  5  9 13 17
2   2  6 10 NA 18
3   3  7 NA 14 19
4   4 NA 11 15 20
5   NA  8 12 16 21
6   1  5  9 13 17
7   2  6 10 NA 18
8   3  7 NA 14 19
9   4 NA 11 15 20
10  NA  8 12 16 21
11  1  5  9 13 17
12  2  6 10 NA 18
13  3  7 NA 14 19
14  4 NA 11 15 20
15  NA  8 12 16 21
16  1  5  9 13 17
17  2  6 10 NA 18
18  3  7 NA 14 19
19  4 NA 11 15 20
20  NA  8 12 16 21
> class(ry)
[1] "data.frame"
> fix(ry) display R editor it is also used for update the data
> ryt<-t(ry) > fix(ryt)
> class(ryt)
[1] "matrix"
> as.data.frame(ryt)->ryt
> class(ryt)
```

```
[1] "data.frame"
> head(ryt)
1 2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 y1  1  2  3  4 NA  1  2  3  4 NA  1  2
   3  4 NA  1  2  3  4 NA y2  5  6  7 NA  8  5  6  7 NA  8  5  6  7 NA  8  5  6  7 NA  8 y3  9 10
   NA 11 12  9 10 NA 11 12  9 10 NA 11 12  9 10 NA 11 12 y4 13 NA 14 15 16 13 NA 14 15 16
   13 NA 14 15 16 13 NA 14 15 16 y5 17 18 19 20 21 17 18 19 20 21 17 18 19 20 21 17 18 19
   20 21
> head(ry)   y1 y2 y3 y4 y5 1  1  5  9 13 17
2 2  6 10 NA 18
3 3  7 NA 14 19
4 4 NA 11 15 20
5 NA  8 12 16 21
6 1  5  9 13 17
> head(ryt,10)
1   2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 y1  1  2  3  4 NA  1  2  3  4 NA  1  2
    3  4 NA  1  2  3  4 NA y2  5  6  7 NA  8  5  6  7 NA  8  5  6  7 NA  8  5  6  7 NA  8 y3  9 10
    NA 11 12  9 10 NA 11 12  9 10 NA 11 12  9 10 NA 11 12 y4 13 NA 14 15 16 13 NA 14 15
    16 13 NA 14 15 16 13 NA 14 15 16 y5 17 18 19 20 21 17 18 19 20 21 17 18 19 20 21 17
    18 19 20 21
> ry    y1 y2 y3 y4 y5 1   1  5  9 13 17
2   2  6 10 NA 18
3   3  7 NA 14 19
4   4 NA 11 15 20
5   NA  8 12 16 21
6   1  5  9 13 17
7   2  6 10 NA 18
8   3  7 NA 14 19
9   4 NA 11 15 20
10  NA  8 12 16 21
11  1  5  9 13 17
12  2  6 10 NA 18
13  3  7 NA 14 19
14  4 NA 11 15 20
15  NA  8 12 16 21
16  1  5  9 13 17
17  2  6 10 NA 18
18  3  7 NA 14 19
19  4 NA 11 15 20
20  NA  8 12 16 21
> head(ry,10) display first 10 elements
   y1 y2 y3 y4 y5 1   1  5  9 13 17
2  2  6 10 NA 18
3  3  7 NA 14 19
4  4 NA 11 15 20
5  NA  8 12 16 21
6  1  5  9 13 17
7  2  6 10 NA 18
8  3  7 NA 14 19
9  4 NA 11 15 20
10 NA  8 12 16 21
> tail(ry) display last 6 elements    y1 y2 y3 y4 y5 15 NA  8 12 16 21
16  1  5  9 13 17
17  2  6 10 NA 18
```

```
18  3  7 NA 14 19
19  4 NA 11 15 20
20  NA  8 12 16 21 > ry[1,]    y1 y2 y3 y4 y5
1   1  5  9 13 17
> ry[,1]
 [1]  1  2  3  4 NA  1  2  3  4 NA  1  2  3  4 NA  1  2  3  4 NA

> ry[,c(2,3)] display 2 3 elements
   y2 y3 1    5  9
2    6 10
3    7 NA
4    NA 11
5    8 12
6    5  9
7    6 10
8    7 NA
9    NA 11
10  8 12
11  5  9
12  6 10
13  7 NA
14  NA 11
15  8 12
16  5  9
17  6 10
18  7 NA
19  NA 11
20  8 12
> ry[c(2:20),] display all elements from 2 to 20 elements
   y1 y2 y3 y4 y5 2    2  6 10 NA 18
3    3  7 NA 14 19
4    4 NA 11 15 20
5    NA  8 12 16 21
6    1  5  9 13 17
7    2  6 10 NA 18
8    3  7 NA 14 19
9    4 NA 11 15 20
10  NA  8 12 16 21
11  1  5  9 13 17
12  2  6 10 NA 18
13  3  7 NA 14 19
14  4 NA 11 15 20
15  NA  8 12 16 21
16  1  5  9 13 17
17  2  6 10 NA 18
18  3  7 NA 14 19
19  4 NA 11 15 20
20  NA  8 12 16 21
> ry$y1
 [1]  1  2  3  4 NA  1  2  3  4 NA  1  2  3  4 NA  1  2  3  4 NA > ry$y2
 [1]  5  6  7 NA  8  5  6  7 NA  8  5  6  7 NA  8  5  6  7 NA  8
> ry$y3
 [1]  9 10 NA 11 12  9 10 NA 11 12  9 10 NA 11 12  9 10 NA 11 12 > ry$y4
 [1] 13 NA 14 15 16 13 NA 14 15 16 13 NA 14 15 16 13 NA 14 15 16
> ry[,c("y1","y2")]    y1 y2 1   1  5
2   2  6
3   3  7
4   4 NA
5   NA  8
6   1  5
7   2  6
8   3  7
9   4 NA
10  NA  8
11  1  5
12  2  6
13  3  7
14  4 NA
15  NA  8
16  1  5
17  2  6
18  3  7
19  4 NA
20  NA  8 > ry[!is.na(ry$y1),]    y1 y2 y3 y4 y5 1   1  5  9 13 17
2  2  6 10 NA 18
3  3  7 NA 14 19
4  4 NA 11 15 20
6  1  5  9 13 17
7  2  6 10 NA 18
8  3  7 NA 14 19
9  4 NA 11 15 20
11  1  5  9 13 17
12  2  6 10 NA 18
13  3  7 NA 14 19
14  4 NA 11 15 20
16  1  5  9 13 17
17  2  6 10 NA 18
18  3  7 NA 14 19
19  4 NA 11 15 20 > is.na(ry$a) logical(0)
Warning message:
In is.na(ry$a) : is.na() applied to non-(list or vector) of type 'NULL'
> is.na(ry$y1)
 [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
[13] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE
> is.na(ry$y2)
 [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
[13] FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE >
```

Data frame is a two dimensional data structure in R. It is a special case of a list which has each component of equal length. Each component form the column and contents of the component form the rows. We can check if a variable is a data frame or not using the class() function.

```
>x
 SN AgeName
1121John
2215Dora
```

```
>typeof(x)# data frame is a special case of  list
[1]"list"

>class(x)
[1]"data.frame"
```

In this example, x can be considered as a list of 3 components with each component having a two element vector. Some useful functions to know more about a data frame are given below.

```
>names(x)
[1]"SN""Age""Name"

>ncol(x)
[1]3

>nrow(x)
[1]2

>length(x)# returns length of the list, same as ncol() [1]3
```

Creating a Data Frame

We can create a data frame using the data.frame() function. For example, the above shown data frame can be created as follows.

```
> x <-data.frame("SN"=1:2,"Age"=c(21,15),"Name"=c("John","Dora"))
>str(x)# structure of x
'data.frame':2 obs.of  3 variables:
$ SN  :int12
$ Age:num2115
$ Name:Factor w/2 levels "Dora","John":21
```

Notice above that the third column, Name is of type factor, instead of a character vector. By default, data.frame() function converts character vector into factor. To suppress this behavior, we can pass the argument stringsAsFactors=FALSE.

```
> x <-
data.frame("SN"=1:2,"Age"=c(21,15),"Name"=c("John","Dora"),stringsAsFactors=F ALSE)

>str(x)# now the third column is a character vector
'data.frame':2 obs.of  3 variables:
$ SN  :int12
$ Age:num2115
$ Name:chr"John""Dora"
```

Many data input functions of R like, read.table(), read.csv(), read.delim(), read.fwf() also read data into a data frame.

Accessing Components in Data Frame

Components of data frame can be accessed like a list or like a matrix.

Accessing like a list

We can use either [, [[or$ operator to access columns of data frame.

```
>x["Name"]
Name
1John
2Dora

>x$Name
[1]"John""Dora"

>x[["Name"]]
[1]"John""Dora"

>x[[3]]
[1]"John""Dora"
```

Accessing with [[or$ is similar. However, it differs for [in that, indexing with [will return us a data frame but the other two will reduce it into a vector.

Accessing like a matrix

Data frames can be accessed like a matrix by providing index for row and column. To illustrate this, we use datasets already available in R. Datasets that are available can be listed with the command library(help = "datasets"). We will use the trees dataset which contains Girth, Height and Volume for Black Cherry Trees. A data frame can be examined using functions like str() and head().

```
>str(trees)
'data.frame':31 obs. of 3 variables:
$ Girth:num8.38.68.810.510.710.8111111.111.2...
$ Height:num70656372818366758075...
$ Volume:num10.310.310.216.418.819.715.618.222.619.9...

>head(trees,n=3)
GirthHeightVolume
18.37010.3 28.66510.3
38.86310.2
```

We can see that trees is a data frame with 31 rows and 3 columns. We also display the first 3 rows of the data frame. Now we proceed to access the data frame like a matrix.

```
>trees[2:3,]# select 2nd and 3rd row
GirthHeightVolume
28.66510.3
38.86310.2

>trees[trees$Height>82,]# selects rows with Height greater than 82
GirthHeightVolume
610.88319.7
1712.98533.8
1813.38627.4
3120.68777.0

>trees[10:12,2]
```

```
[1]757976
```

We can see in the last case that the returned type is a vector since we extracted data from a single column. This behavior can be avoided by passing the argument `drop=FALSE` as follows.

```
>trees[10:12,2, drop=FALSE]
Height
1075
1179
1276
```

Modifying a Data Frame

Data frames can be modified like we modified matrices through reassignment.

```
>x
 SN AgeName
1121John
2215Dora

>x[1,"Age"]<-20; x
 SN AgeName
1120John
2215Dora
```

Adding Components

Rows can be added to a data frame using the `rbind()` function.

```
>rbind(x,list(1,16,"Paul"))
 SN AgeName
1120John
2215Dora
3116Paul
```

Similarly, we can add columns using `cbind()`.

```
>cbind(x,State=c("NY","FL"))
 SN AgeNameState
1120John    NY
2215Dora    FL
```

Since data frames are implemented as list, we can also add new columns through simple list-like assignments.

```
>x
 SN AgeName
1120John
2215Dora

>x$State<- c("NY","FL"); x
 SN AgeNameState
1120John    NY
2215Dora    FL
```

Deleting Component

Data frame columns can be deleted by assigning NULL to it.

```
>x$State<- NULL
>x
 SN AgeName
1120John
2215Dora
```

Similarly, rows can be deleted through reassignments.

```
> x <-x[-1,]
>x
 SN AgeName 2215Dora
```

Following are the characteristics of a data frame.

- The column names should be non-empty.
- The row names should be unique.
- The data stored in a data frame can be of numeric, factor or character type.
- Each column should contain same number of data items. **Model 2: Create Data Frame**

Following are the characteristics of a data frame.

- The column names should be non-empty.
- The row names should be unique.
- The data stored in a data frame can be of numeric, factor or character type.
- Each column should contain same number of data items.

```
# Create the data frame. emp.data<-data.frame( emp_id= c (1:5),
emp_name=c("Rick","Dan","Michelle","Ryan","Gary"), salary=
c(623.3,515.2,611.0,729.0,843.25),
 start_date=as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-11",
"2015-03-27")), stringsAsFactors= FALSE
)
# Print the data frame.
print(emp.data)
```

When we execute the above code, it produces the following result −

```
emp_idemp_name     salary     start_date 1     1     Rick       623.30     2012-01-01
2        2     Dan         515.20     2013-09-23
3        3     Michelle    611.00     2014-11-15
4        4     Ryan        729.00     2014-05-11
5        5     Gary        843.25     2015-03-27
```

Get the Structure of the Data Frame

The structure of the data frame can be seen by using **str()** function.

```
# Get the structure of the data frame. str(emp.data)
```

Summary of Data in Data Frame

```
# Print the summary. print(summary(emp.data))
```

Extract Data from Data Frame

Extract specific column from a data frame using column name.

```
# Create the data frame. emp.data<-data.frame( emp_id= c (1:5),
emp_name=c("Rick","Dan","Michelle","Ryan","Gary"), salary=
c(623.3,515.2,611.0,729.0,843.25),

start_date=as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-11",
"2015-03-27")), stringsAsFactors= FALSE
)
# Extract Specific columns. result<-data.frame(emp.data$emp_name,emp.data$salary)
print(result)
```

When we execute the above code, it produces the following result −

```
emp.data.emp_nameemp.data.salary
1                     Rick        623.30
2                     Dan         515.20
3                     Michelle        611.00
4                     Ryan         729.00
5                     Gary         843.25
```

Extract the first two rows and then all columns

```
# Create the data frame. emp.data<-data.frame( emp_id= c (1:5),
emp_name=c("Rick","Dan","Michelle","Ryan","Gary"), salary=
c(623.3,515.2,611.0,729.0,843.25),
 start_date=as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-11",
"2015-03-27")), stringsAsFactors= FALSE )
# Extract first two rows. result<-emp.data[1:2,] print(result)
```

When we execute the above code, it produces the following result −

```
emp_idemp_name   salary    start_date 1     1     Rick      623.3     2012-01-01
2     2     Dan       515.2     2013-09-23
```

Extract 3rd and 5th row with 2nd and 4th column

```
# Create the data frame. emp.data<-data.frame( emp_id= c (1:5),
emp_name=c("Rick","Dan","Michelle","Ryan","Gary"), salary=
c(623.3,515.2,611.0,729.0,843.25),

        start_date=as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-0511",
"2015-03-27")), stringsAsFactors= FALSE
)

# Extract 3rd and 5th row with 2nd and 4th column. result<-emp.data[c(3,5),c(2,4)]
print(result)
```

When we execute the above code, it produces the following result −

```
emp_namestart_date
3 Michelle 2014-11-15
5     Gary 2015-03-27
```

Expand Data Frame

A data frame can be expanded by adding columns and rows.

Add Column

Just add the column vector using a new column name.

```
# Create the data frame. emp.data<-data.frame( emp_id= c (1:5),
emp_name=c("Rick","Dan","Michelle","Ryan","Gary"), salary=
c(623.3,515.2,611.0,729.0,843.25),

start_date=as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-11",
"2015-03-27")), stringsAsFactors= FALSE
)

# Add the "dept" coulmn.
emp.data$dept<-c("IT","Operations","IT","HR","Finance") v <-emp.data
print(v)
```

When we execute the above code, it produces the following result −

```
emp_idemp_name    salary    start_date           dept
1        1    Rick        623.30    2012-01-01       IT
2        2    Dan          515.20    2013-09-23       Operations
3        3    Michelle    611.00    2014-11-15       IT
4        4    Ryan         729.00    2014-05-11       HR
5        5    Gary         843.25    2015-03-27       Finance
```

Dataframe have some Useful functions:

- head() - see first 6 rows
- tail() - see last 6 rows
- dim() - see dimensions
- nrow() - number of rows
- ncol() - number of columns
- str() - structure of each column
- names() - will list the names attribute for a data frame (or any object really), which gives the column names.

Arrays in R programming: Arrays are the R data objects which can store data in more than two dimensions (or) matrices .For example − If we create an array of dimension (2, 3, 4) then it creates 4 rectangular matrices each with 2 rows and 3 columns. Arrays can store only data type.

An array is created using the **array()** function. It takes vectors as input and uses the values in the **dim** parameter to create an array.

Example

The following example creates an array of two 3x3 matrices each with 3 rows and 3 columns.

```
# Create two vectors of different lengths.
vector1 <-c(5,9,3)
vector2 <-c(10,11,12,13,14,15)

# Take these vectors as input to the array.
result<- array(c(vector1,vector2),dim = c(3,3,2)) print(result)
```

When we execute the above code, it produces the following result −

```
, , 1

    [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15

, , 2

    [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15
```

Naming Columns and Rows

We can give names to the rows, columns and matrices in the array by using the **dimnames** parameter.

```
# Create two vectors of different lengths. vector1 <-c(5,9,3) vector2 <-c(10,11,12,13,14,15)
column.names<-c("COL1","COL2","COL3") row.names<-c("ROW1","ROW2","ROW3")
matrix.names<-c("Matrix1","Matrix2")

# Take these vectors as input to the array. result <- array(c(vector1,vector2),dim =
c(3,3,2),dimnames= list(column.names,row.names, matrix.names)) print(result)
```

When we execute the above code, it produces the following result −

```
, , Matrix1

    ROW1 ROW2 ROW3
COL1    5   10   13
COL2    9   11   14
COL3    3   12   15

, , Matrix2
```

```
     ROW1 ROW2 ROW3
COL1    5   10   13
COL2    9   11   14
COL3    3   12   15
```

Accessing Array Elements

```
# Create two vectors of different lengths.
vector1 <-c(5,9,3) vector2 <-c(10,11,12,13,14,15) column.names<-c("COL1","COL2","COL3")
row.names<-c("ROW1","ROW2","ROW3") matrix.names<-c("Matrix1","Matrix2")

# Take these vectors as input to the array. result<- array(c(vector1,vector2),dim =
c(3,3,2),dimnames= list(column.names, row.names,matrix.names))

# Print the third row of the second matrix of the array. print(result[3,,2])

# Print the element in the 1st row and 3rd column of the 1st matrix.
print(result[1,3,1])

# Print the 2nd Matrix.
print(result[,,2])
```

When we execute the above code, it produces the following result −

```
ROW1 ROW2 ROW3
   3   12   15
[1] 13
     ROW1 ROW2 ROW3
COL1    5   10   13
COL2    9   11   14
COL3    3   12   15
```

Manipulating Array Elements

As array is made up matrices in multiple dimensions, the operations on elements of array are carried out by accessing elements of the matrices.

```
# Create two vectors of different lengths. vector1 <-c(5,9,3)
vector2 <-c(10,11,12,13,14,15)

# Take these vectors as input to the array.
array1 <- array(c(vector1,vector2),dim = c(3,3,2))

# Create two vectors of different lengths. vector3 <-c(9,1,0) vector4 <-c(6,0,11,3,14,1,2,6,9)
array2 <- array(c(vector1,vector2),dim = c(3,3,2))

# create matrices from these arrays.
matrix1 <-array1[,,2] matrix2 <-array2[,,2]

# Add the matrices.
result<- matrix1+matrix2 print(result)
```

When we execute the above code, it produces the following result −

```
     [,1] [,2] [,3]
[1,]   10   20   26
[2,]   18   22   28
[3,]    6   24   30
```

Calculations Across Array Elements

We can do calculations across the elements in an array using the **apply()** function. **Syntax**

apply(x, margin, fun)

Following is the description of the parameters used −

- **x** is an array.
- **margin** is the name of the data set used.
- **fun** is the function to be applied across the elements of the array.

Example
We use the apply() function below to calculate the sum of the elements in the rows of an array across all the matrices.
# Create two vectors of different lengths. vector1 <-c(5,9,3)
vector2 <-c(10,11,12,13,14,15)

# Take these vectors as input to the array. new.array<- array(c(vector1,vector2),dim = c(3,3,2)) print(new.array)

# Use apply to calculate the sum of the rows across all the matrices. result<- apply(new.array, c(1), sum)
print(result)
When we execute the above code, it produces the following result −

```
, , 1

     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15

, , 2

     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15

[1] 56 68 60
```

Factors in R : Factors are the data objects which are used to categorize the data and store it as levels. They can store both strings and integers. They are useful in the columns which have a limited number of unique values. Like "Male, "Female" and True, False etc. They are useful in data analysis for statistical modeling.

☐ Factors are created using the **factor ()** function by taking a vector as input.

Variables in factors: Variables mainly four categories 1.**Categorical Variable**:
Gender:male,female

Party:democratic,republication

Bloodgroup: A,B ,AB,O

2.**Ordinal Variables**: low,medium,high,

Slow,fast,faster

3**.Interval Variables:$**10 to $20
$20 to 30

$30 t0 40

4.Numerical Variables:1St,2nd 3rd

```
>gender<-c("single","married","married","single")
[1]single  marriedmarried single
Levels: married  single
```

Here, we can see that factor x has four elements and two levels. We can check if a variable is a factor or not using class() function. Similarly, levels of a factor can be checked using the levels() function.

```
>class(x)
[1]"factor"

>levels(x)
[1]"married""single"
```

Creating a Factor:We can create a factor using the function factor(). Levels of a factor are inferred from the data if not provided.

```
tshirt<-c("m","l","s","s","l","l","m","m")
>tshirt_fact<-factor(tshirt)
>printtshirt_fact)
Error: unexpected ')' in "printtshirt_fact)"
>print(tshirt_fact)
[1] m l s s l l m m
Levels: l m s
>str(tshirt_fact)
 Factor w/ 3 levels "l","m","s": 2 1 3 3 1 1 2 2
> gender<-c("male","female","single","widowd","male","female")
>print(gender)
[1] "male"   "female" "single" "widowd" "male"   "female"
>fact_gender<-factor(gender)
>print(fact_gender)
[1] male   female single widowd male   female
Levels: female male single widowd
>str(fact_gender)
 Factor w/ 4 levels "female","male",..: 2 1 3 4 2 1
```

>We can see from the above example that levels may be predefined even if not used.

Factors are closely related with vectors. In fact, factors are stored as integer vectors. This is clearly seen from its structure.

```
> x <- factor(c("single","married","married","single"))
>str(x)
Factor w/2 levels "married","single":2112
```

We see that levels are stored in a character vector and the individual elements are actually stored as indices.

Factors are also created when we read non-numerical columns into a data frame. By default, data.frame() function converts character vector into factor. To suppress this behavior, we have to pass the argument stringsAsFactors=FALSE.

Accessing Components in Factor

Accessing components of a factor is very much similar to that of vectors.

```
>x
[1]single  marriedmarried single
Levels: married single

>x[3]# access 3rd element
[1]married
Levels: married single

>  x[c(2,4)]# access 2nd and 4th element
[1]married single
Levels: married single

>x[-1]# access all but 1st element
[1]marriedmarried single
Levels: married single

>x[c(TRUE, FALSE, FALSE, TRUE)]# using logical vector
[1]singlesingle
Levels: married single
```

Modifying a Factor: Components of a factor can be modified using simple assignments. However, we cannot choose values outside of its predefined levels.

```
>x
[1]single  marriedmarried single
Levels: single married divorced

>x[2]<-"divorced"# modify second element;  x
[1]single   divorced married  single
Levels: single married divorced

>x[3]<-"widowed"# cannot assign values outside levels Warning message:
In`[<-.factor`(`*tmp*`,3, value ="widowed"):
invalid factor level, NA generated
>x
[1]single   divorced <NA>     single
Levels: single married divorced
```

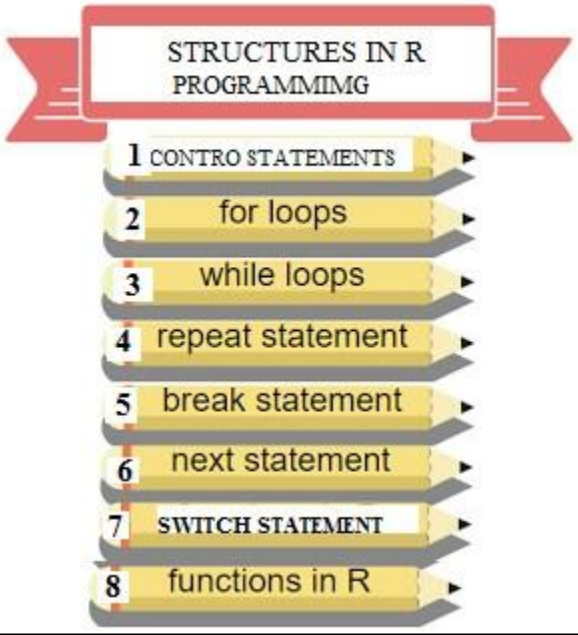
A workaround to this is to add the value to the level first.

```
>levels(x)<- c(levels(x),"widowed")# add new level
>x[3]<-"widowed"
>x
```

```
[1]single   divorced widowed  single
Levels: single married divorced widowed
```

UNIT-II
STRUCTURES IN R PROGRAMMING

Structures in R programming are 8 different ways: THOSE ARE FOLLOWING WAYS



1. R FLOW CONTROL(or) Contro Statements in R

Decision making is an important part of programming. This can be achieved in R programming using the conditional if...else statement. **if statement**

Syntax of if statement

```
if(test_expression){ statement }
```

If the test_expression is TRUE, the statement gets executed. But if it's FALSE, nothing happens. Here, test_expression can be a logical or numeric vector, but only the first element is taken into consideration. In the case of numeric vector, zero is taken as FALSE, rest as TRUE. **Example of if statement**

```
x <-5 if(x >0){
print("Positive number") }
```

Output

```
[1] "Positive number"
```

if…else statement

Syntax of if...else statement

```
if(test_expression){    statement1 }else{    statement2 }
```

The else part is optional and is evaluated if test_expression is FALSE. It is important to note that else must be in the same line as the closing braces of the if statements. **Example of if...else statement**

```
x <--5 if(x >0){
print("Non-negative number")
}else{
print("Negative number") }
```

Output

```
[1] "Negative number"
```

The above conditional can also be written in a single line as follows.

```
if(x >0)print("Non-negative number")elseprint("Negative number")
```

This feature of R allows us to write construct as shown below.

```
> x <--5
> y <-if(x >0)5else6
>y
[1]6
```

Nested if...else statement

We can nest as many if...else statement as we want as follows. **Syntax of nested if...else statement**

```
if( test_expression1){    statement1
}elseif( test_expression2){    statement2
}elseif( test_expression3){    statement3 }else    statement4
```

Only one statement will get executed depending upon the test_expressions.

Example of nested if...else statement

```
x <-0 if(x <0){
print("Negative number")
}elseif(x >0){ print("Positive number")
}else print("Zero")
```

Output

```
[1] "Zero"
```

R Programming ifelse() Function

Vectors form the basic building block of R programming. Most of the functions in R take vector as input and output a resultant vector. This vectorization of code, will be much faster than applying the same function to each element of the vector individually.

Similar to this concept, there is a vector equivalent form of the if...else statement in R, the ifelse() function.

Syntax of ifelse() function

```
ifelse(test_expression,x,y)
```

Here, test_expression must be a logical vector (or an object that can be coerced to logical). The return value is a vector with the same length as test_expression. This returned vector has element from x if the corresponding value of test_expression is TRUE or from y if the corresponding value of test_expression is FALSE. This is to say, the i-th element of result will be x[i] if test_expression[i] is TRUE else it will take the value of y[i]. The vectors x and y are recycled whenever necessary.

Example of ifelse() function

```
> a =c(5,7,2,9)
>ifelse(a %%2==0,"even","odd")
[1]"odd""odd""even""odd"
```

In the above example, the test_expression is a %% 2 == 0 which will result into the vector (FALSE,FALSE,TRUE ,FALSE). Similarly, the other two vectors in the function argument gets recycled to ("even","even","even","even") and ("odd","odd","odd","odd") respectively. And hence the result is evaluated accordingly

ITERATIVE STATEMENTS or LOOPING STATEMENTS: LOOP,WHILE STATEMENTS ARE LOOPING STATEMENTS

2.R Programming for loop

A for loop is used to iterate over a vector, in R programming.

Syntax of for loop

for(valin sequence){ statement }

Here, sequence is a vector and val takes on each of its value during the loop. In each iteration, statement is evaluated.

Example of for loop

Below is an example to count the number of even numbers in a vector.

```
x <-c(2,5,3,9,8,11,6) count<-0 for(valin x){
if(val%%2==0)  count = count+1
} print(count)
```

Output

```
[1] 3
```

In the above example, the loop iterates 7 times as the vector x has 7 elements. In each iteration, val takes on the value of corresponding element of x. We have used a counter to count the number of even numbers in x. We can see that x contains 3 even numbers.

3.R Programming while loop:

In R programming, while loops are used to loop until a specific condition is met. **Syntax of while loop**

while(test_expression){ statement }

Here, test_expression is evaluated and the body of the loop is entered if the result is TRUE. The statements inside the loop are executed and the flow returns to evaluate the test_expression again. This is repeated each time until test_expression evaluates to FALSE, in which case, the loop exits. **Example of while loop**

```
i <-1  while(i <6){ print(i)    i = i+1 }
```

Output

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

In the above example, iis initialized to 1. Here, the test_expression is i < 6 which evaluates to TRUE since 1 is less than 6. So, the body of the loop is entered and iis printed and incremented. Incrementing iis important as this will eventually meet the exit condition. Failing to do so will result into an infinite loop. In the next iteration, the value of i is 2 and the loop continues. This will continue until itakes the value 6. The condition 6 < 6 will give FALSE and the loop finally exits.

4.R Programming repeat loop

A repeat loop is used to iterate over a block of code multiple number of times. There is no condition check in repeat loop to exit the loop. We must ourselves put a condition explicitly inside the body of the loop and use the break statement to exit the loop. Failing to do so will result into an infinite loop.

Syntax of repeat loop

repeat { statement }

In the statement block, we must use the break statement to exit the loop. **Example of repeat loop**

```
x <-1  repeat{ print(x)    x = x+1 if(x ==6){ break }
}
```
Output

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

In the above example, we have used a condition to check and exit the loop when x takes the value of 6. Hence, we see in our output that only values from 1 to 5 get printed.

5.R Programming break statement

In R programming, a normal looping sequence can be altered using the break or the next statement.

break statement

A break statement is used inside a loop to stop the iterations and flow the control outside of the loop. In a nested looping situation, where there is a loop inside another loop, this statement exits from the innermost loop that is being evaluated.

Syntax of break statement

Break

Flowchart of break statement Example of break statement

```
x <-1:5  for(valin x){ if(val==3){ break } print(val) }
```
Output

```
[1] 1
[1] 2
```

In this example, we iterate over the vector x, which has consecutive numbers from 1 to 5. Inside the for loop we have used a condition to break if the current value is equal to 3. As we can see from the output, the loop terminates when it encounters the break statement.

6.next statement in R

A next statement is useful when we want to skip the current iteration of a loop without terminating it. On encountering next, the R parser skips further evaluation and starts next iteration of the loop.

Syntax of next statement

next

Example of next statement

```
x <-1:5  for(valin x){ if(val==3){ next } print(val)
}
```

Output

```
[1] 1
[1] 2
[1] 4
[1] 5
```

In the above example, we use the `next` statement inside a condition to check if the value is equal to 3. If the value is equal to 3, the current evaluation stops (value is not printed) but the loop continues with the next iteration. The output reflects this situation.

7.R Programming Switch Function:

Like the `switch` statements in other programming languages, R has a similar construct in the form of `switch()` function. **Syntax of `switch()` function**

```
switch(statement, list)
```

Here, the `statement` is evaluated and based on this value, the corresponding item in the list is returned.

Example of `switch()` function

If the value evaluated is a number, that item of the list is returned.

```
>switch(2,"red","green","blue")
[1]"green"

>switch(1,"red","green","blue")
[1]"red"
```

In the above example, `"red","green","blue"` form a three item list. So, the `switch()` function returns the corresponding item to the numeric value evaluated. But if the numeric value is out of range (greater than the number of items in the list or smaller than 1), then, `NULL` is returned.

```
> x <-switch(4,"red","green","blue")
>x
NULL

> x <-switch(0,"red","green","blue")
>x
NULL
```

The result of the statement can be a string as well. In this case, the matching named item's value is returned.

```
>switch("color","color"="red","shape"="square","length"=5) [1]"red"

>switch("length","color"="red","shape"="square","length"=5) [1]5
```

Check out these examples to learn more:

8. R – Functions:

A function is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.

In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.

The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

Function Definition

An R function is created by using the keyword **function**. The basic syntax of an R function definition is as follows −

```
function_name<-function(arg_1, arg_2,...){
Function body  }
```

Function Components

The different parts of a function are −

- **Function Name** − This is the actual name of the function. It is stored in R environment as an object with this name.
- **Arguments** − An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.
- **Function Body** − The function body contains a collection of statements that defines what the function does. □ **Return Value** − The return value of a function is the last expression in the function body to be evaluated.

R has many **in-built** functions which can be directly called in the program without defining them first. We can also create and use our own functions referred as **user defined** functions.

Built-in Function

Simple examples of in-built functions are **seq()**, **mean()**, **max()**, **sum(x)** and **paste(...)** etc. They are directly called by user written programs. You can refer most widely used R functions.

```
# Create a sequence of numbers from 32 to 44. print(seq(32,44))

# Find mean of numbers from 25 to 82.
print(mean(25:82))

# Find sum of numbers frm 41 to 68.
print(sum(41:68))
```
When we execute the above code, it produces the following result −

```
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44
[1] 53.5
[1] 1526
```

User-defined Function

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function is created and used.

```
# Create a function to print squares of numbers in sequence. new.function<-function(a){ for(i
in1:a){       b <- i^2 print(b) }
}
```

Calling a Function

```
# Create a function to print squares of numbers in sequence. new.function<-function(a){ for(i
in1:a){       b <- i^2 print(b)
}
}

# Call the function new.function supplying 6 as an argument. new.function(6)
```

When we execute the above code, it produces the following result −

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```

Calling a Function without an Argument

```
# Create a function without an argument.
new.function<-function(){ for(i in1:5){ print(i^2) }
}

# Call the function without supplying an argument. new.function()
```

When we execute the above code, it produces the following result −

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

Calling a Function with Argument Values (by position and by name)

The arguments to a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments.

```
# Create a function with arguments.
new.function<-function(a,b,c){ result<- a * b + c print(result)
}

# Call the function by position of arguments.
new.function(5,3,11)

# Call the function by names of the arguments. new.function(a =11, b =5, c =3)
```

When we execute the above code, it produces the following result −

```
[1] 26
[1] 58
```

Calling a Function with Default Argument

We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non default result.

```
# Create a function with arguments. new.function<-function(a =3, b =6){ result<- a * b
print(result)
}

# Call the function without giving any argument. new.function()

# Call the function with giving new values of the argument. new.function(9,5)
```

When we execute the above code, it produces the following result −

```
[1] 18
[1] 45
```

Lazy Evaluation of Function

Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.

```
# Create a function with arguments.
new.function<-function(a, b){ print(a^2) print(a) print(b)
}

# Evaluate the function without supplying one of the arguments. new.function(6)
```

When we execute the above code, it produces the following result −

```
[1] 36
[1] 6
Error in print(b) : argument "b" is missing, with no default
```

Multiple Returns

The `return()` function can return only a single object. If we want to return multiple values in R, we can use a list (or other objects) and return it. Following is an example.

```
multi_return<- function() {
my_list<- list("color" = "red", "size" = 20, "shape" = "round")    return(my_list)  }
```

Here, we create a list `my_list` with multiple elements and return this single list.

```
> a <- multi_return()
>a$color
[1] "red"
```

```
>a$size
[1] 20

>a$shape
[1] "round"
```

***R Programming Environment and Scope**

In order to write functions in a proper way and avoid unusual errors, we need to know the concept of environment and scope in R.

R Programming Environment

Environment can be thought of as a collection of objects (functions, variables etc.). An environment is created when we first fire up the R interpreter. Any variable we define, is now in this environment. The top level environment available to us at the R command prompt is the global environment called R_GlobalEnv. Global environment can be referred to as .GlobalEnv in R codes as well. We can use the ls() function to show what variables and functions are defined in the current environment. Moreover, we can use the environment() function to get the current environment.

```
> a <- 2
> b <- 5
> f <- function(x) x<-0

>ls()
[1] "a" "b" "f"

> environment()
<environment: R_GlobalEnv>

> .GlobalEnv
<environment: R_GlobalEnv>
```

In the above example, we can see that a, b and f are in the R_GlobalEnv environment. Notice that x (in the argument of the function) is not in this global environment. When we define a function, a new environment is created. In the above example, the function f creates a new environment inside the global environment. Actually an environment has a frame, which has all the objects defined, and a pointer to the enclosing (parent) environment. Hence, x is in the frame of the new environment created by the function f. This environment will also have a pointer to R_GlobalEnv. Consider the following example for more clarification of the cascading of environments.

```
f <- function(f_x){    g <- function(g_x){        print("Inside g")        print(environment())
print(ls())
   }    g(5)    print("Inside f")    print(environment())    print(ls()) }
```

Now when we run it from the command prompt, we get.

```
> f(6)
[1] "Inside g"
<environment: 0x0000000010c2bdc8>
```

```
[1] "g_x"
[1] "Inside f"
<environment: 0x0000000010c2a870>
[1] "f_x" "g"

> environment()
<environment: R_GlobalEnv>

>ls()
[1] "f"
```

Here, we defined function g inside f and it is clear that they both have different environments with different objects within their respective frames.

R Programming Scope

Let us consider the following example.

```
outer_func<- function(){    b <- 20
inner_func<- function(){        c <- 30
   } } a <- 10
```

Here we have a function inner_func() nested within outer_func(). The variable c is local to it while, a and b are global. Now from the perspective of outer_func(), b is local to it and a is global. The variable c is completely invisible to outer_func(). If we assign a value to a variable within a function, it will be local and will not effect to any global variable even if the name matches. For example, if we have a function as below.

```
outer_func<- function(){    a <- 20
inner_func<- function(){        a <- 30        print(a)
   } inner_func()    print(a) }
```

When we call it,

```
> a <- 10

>outer_func()
[1] 30
[1] 20

> print(a)
[1] 10
```

We see that the variable a is created locally within the environment frame of both the functions and is different to that of the global environment frame.

Global variables can be read but when we try to assign to it, a new local variable is created instead. To make assignments to global variables, superassignment operator, <<-, is used. When using this operator within a function, it searches for the variable in the parent environment frame, if not found it keeps on searching the next level until it reaches the global environment. If the variable is still not found, it is created and assigned at the global level.

```
outer_func<- function(){ inner_func<- function(){        a <<- 30        print(a)
   } inner_func()    print(a) }
```

On running this function,

```
>outer_func()
[1] 30
[1] 30
> print(a)
[1] 30
```

When the statement `a <<- 30` is encountered within `inner_func()`, it looks for the variable `a` in outer_func() environment. When the search fails, it searches in `R_GlobalEnv`. Since, `a` is not defined in this global environment as well, it is created and assigned there which is now referenced and printed from within `inner_func()` as well as `outer_func()`.

R Programming Recursive Function

A function that calls itself is called a recursive function. This special programming technique can be used to solve problems by breaking them into smaller and simpler sub-problems. An example can help clarify this concept.

Let us take the example of finding the factorial of a number. Factorial of a positive integer number is defined as the product of all the integers from 1 to that number. For example, the factorial of 5 (denoted as `5!`) will be `1*2*3*4*5 = 120`. This problem of finding factorial of 5 can be broken down into a sub-problem of multiplying the factorial of 4 with 5.

5! = 5*4!

Or more generally,

n! = n*(n-1)!

Now we can continue this until we reach `0!` which is `1`. The implementation of this is provided below.

Example of Recursive Function in R

```
# Recursive function to find factorial

recursive.factorial<-function(x){ if(x ==0)return(1)
elsereturn(x *recursive.factorial(x-1)) }
```

Here, we have a function which will call itself. Something like `recursive.factorial(x)` will turn into x * recursive.factorial(x) until x becomes equal to 0. When x becomes 0, we return `1` since the factorial of `0` is `1`. This is the terminating condition and is very important. Without this the recursion will not end and continue indefinitely (in theory). Here are some sample function calls to our function.

```
>recursive.factorial(0)
[1]1

>recursive.factorial(5)
[1]120

>recursive.factorial(7)
```

```
[1]5040
```

The use of recursion, often, makes code shorter and looks clean. But it is sometimes hard to follow through the code logic. It might be hard to think of a problem in a recursive way. Recursive functions are also memory intensive, since it can result into a lot of nested function calls. This must be kept in mind when using it for solving big problems.

R Programming Infix Operator

Most of the operators that we use in R are binary operators (having two operands). Hence, they are infix operators, used between the operands. Actually, these operators do a function call in the background. For example, the expression `a+b` is actually calling the function `` `+`() `` with the arguments `a` and `b`, as `` `+`(a, b) ``. Note the back tick (`` ` ``), this is important as the function name contains special symbols. Following are some example expressions along with the actual functions that get called in the background.

```
> 5+3
[1] 8
> `+`(5,3)
[1] 8

> 5-3
[1] 2
> `-`(5,3)
[1] 2

> 5*3-1
[1] 14
> `-`(`*`(5,3),1)
[1] 14
```

It is possible to create user-defined infix operators in R. This is done by naming a function that starts and ends with `%`. Following is an example of user-defined infix operator to see if a number is exactly divisible by another.

```
`%divisible%` <- function(x,y)
{
  if (x%%y ==0) return (TRUE)    else          return (FALSE) }
```

This function can be used as infix operator `a %divisible% b` or as a function call `%divisible%`(a, b). Both are the same.

```
> 10 %divisible% 3
[1] FALSE

> 10 %divisible% 2
[1] TRUE

> `%divisible%`(10,5)
[1] TRUE
```

Things to remember while defining your own infix operators are that they must start and end with `%`. Surround it with back tick (`` ` ``) in the function definition and escape any special symbols. Following operators are predefined in R.

Pr defined infix operators in R	
%%	Remainder operator
%/%	Integer division
%*%	Matrix multiplication
%o%	Outer product
%x%	Kronecker product
%in%	Matching operator

***R Program to Make a Simple Calculator**

```
# Program make a simple calculator
# that can add, subtract, multiply
# and divide using functions
add <- function(x, y) {
   return(x + y)
}
subtract <- function(x, y) {
   return(x - y)
}
multiply <- function(x, y) {
   return(x * y)
}
divide <- function(x, y) {
   return(x / y)
}
# take input from the user
print("Select operation.") print("1.Add") print("2.Subtract") print("3.Multiply") print("4.Divide")
choice = as.integer(readline(prompt="Enter choice[1/2/3/4]: ")) num1 =
as.integer(readline(prompt="Enter first number: ")) num2 =
as.integer(readline(prompt="Enter second number: "))
operator <- switch(choice,"+","-","*","/")
result <- switch(choice, add(num1, num2), subtract(num1, num2), multiply(num1, num2),
divide(num1, num2))
print(paste(num1, operator, num2, "=", result)
[1] "Select operation."
[1] "1.Add"
[1] "2.Subtract"
[1] "3.Multiply"
[1] "4.Divide"
Enter choice[1/2/3/4]: 4
Enter first number: 20
Enter second number: 4
[1] "20 / 4 = 5"
```

In this program, we ask the user to choose the desired operation. Options 1, 2, 3 and 4 are valid.

Two numbers are taken from the user and a switch branching is used to execute a particular function. User-defined functions add(), subtract(), multiply() and divide() evaluate respective operations.

R – Strings: Any value written within a pair of single quote or double quotes in R is treated as a string. Internally R stores every string within double quotes, even when you create them with single quote.

Rules Applied in String Construction

- The quotes at the beginning and end of a string should be both double quotes or both single quote. They can not be mixed.
- Double quotes can be inserted into a string starting and ending with single quote.
- Single quote can be inserted into a string starting and ending with double quotes.
- Double quotes can not be inserted into a string starting and ending with double quotes. ☐ Single quote can not be inserted into a string starting and ending with single quote.

Examples of Valid Strings

Following examples clarify the rules about creating a string in R.

```
a       <- 'Start and end with single quote' print(a)
b       <- "Start and end with double quotes" print(b)
c       <- "single quote ' in between double quotes" print(c)
d       <- 'Double quotes " in between single quote' print(d)
```

When the above code is run we get the following output −

```
[1] "Start and end with single quote"
[1] "Start and end with double quotes"
[1] "single quote ' in between double quote" [1] "Double quote \" in between single quote"
```

Examples of Invalid Strings

```
e       <- 'Mixed quotes"  print(e)
f       <- 'Single quote ' inside single quote' print(f)
g       <- "Double quotes " inside double quotes" print(g)
```

When we run the script it fails giving below results.

```
...: unexpected INCOMPLETE_STRING

.... unexpected symbol  1: f <- 'Single quote ' inside

unexpected symbol
1: g <- "Double quotes " inside
```

String Manipulation

Concatenating Strings - paste() function

Many strings in R are combined using the **paste()** function. It can take any number of arguments to be combined together.

Syntax

The basic syntax for paste function is −

paste(..., sep = " ", collapse = NULL)

Following is the description of the parameters used −

- **...**represents any number of arguments to be combined.
- **sep** represents any separator between the arguments. It is optional.
- **collapse** is used to eliminate the space in between two strings. But not the space within two words of one string.

Example

```
a <- "Hello" b <- 'How' c <- "are you? "

print(paste(a,b,c))
 print(paste(a,b,c, sep = "-"))
 print(paste(a,b,c, sep = "", collapse = ""))
```

When we execute the above code, it produces the following result −

```
[1] "Hello How are you? "
[1] "Hello-How-are you? "
[1] "HelloHoware you? "
```

Formatting numbers & strings - format() function

Numbers and strings can be formatted to a specific style using **format()** function.

Syntax

The basic syntax for format function is −
format(x, digits, nsmall, scientific, width, justify = c("left", "right", "centre", "none"))

Following is the description of the parameters used −

- **x** is the vector input.
- **digits** is the total number of digits displayed.
- **nsmall** is the minimum number of digits to the right of the decimal point.
- **scientific** is set to TRUE to display scientific notation.
- **width** indicates the minimum width to be displayed by padding blanks in the beginning. ☐ **justify** is the display of the string to left, right or center.

Example

```
# Total number of digits displayed. Last digit rounded off. result<- format(23.123456789,
digits = 9) print(result)

# Display numbers in scientific notation.
result<- format(c(6, 13.14521), scientific = TRUE) print(result)

# The minimum number of digits to the right of the decimal point. result<- format(23.47,
nsmall = 5) print(result)
```

```
# Format treats everything as a string. result<- format(6) print(result)

# Numbers are padded with blank in the beginning for width. result<- format(13.7, width = 6)
print(result)

# Left justify strings. result<- format("Hello", width = 8, justify = "l") print(result)

# Justfy string with center.
result<- format("Hello", width = 8, justify = "c") print(result)
```

When we execute the above code, it produces the following result −

```
[1] "23.1234568"
[1] "6.000000e+00" "1.314521e+01"
[1] "23.47000"
[1] "6"
[1] "  13.7"
[1] "Hello   "
[1] " Hello  "
```

Counting number of characters in a string - nchar() function

This function counts the number of characters including spaces in a string.
Syntax

The basic syntax for nchar() function is −

```
nchar(x)
```

Following is the description of the parameters used − ☐ **x** is the vector input.

Example

```
result<- nchar("Count the number of characters") print(result)
```

When we execute the above code, it produces the following result −

```
[1] 30
```

Changing the case - toupper() &tolower() functions

These functions change the case of characters of a string.

Syntax

The basic syntax for toupper() &tolower() function is −

```
toupper(x) tolower(x)
```

Following is the description of the parameters used − ☐ **x** is the vector input.

Example

```
# Changing to Upper case.
result<- toupper("Changing To Upper") print(result)

# Changing to lower case.
result<- tolower("Changing To Lower") print(result)
```

When we execute the above code, it produces the following result −

```
[1] "CHANGING TO UPPER"
[1] "changing to lower"
```

Extracting parts of a string - substring() function

This function extracts parts of a String.
Syntax

The basic syntax for substring() function is −

```
substring(x,first,last)
```

Following is the description of the parameters used −

- **x** is the character vector input.
- **first** is the position of the first character to be extracted.
- **last** is the position of the last character to be extracted.

Example

```
# Extract characters from 5th to 7th position.
result<- substring("Extract", 5, 7)
print(result) When we execute
```

Another examples:

1.grep( )
The call grep(pattern,x) searches for a specified substring pattern in a vector x of strings. If x has *n* elements—that is, it contains *n* strings—then grep(pattern,x) will return a vector of length up to *n*. Each element of this vector will be the index in x at which a match of pattern as a substring of x[i]) was found.

Here's an example of using grep:

```
> grep("Pole",c("Equator","North Pole","South Pole"))
[1] 2 3
> grep("pole",c("Equator","North Pole","South Pole")) integer(0)
```

In the first case, the string "Pole" was found in elements 2 and 3 of the second argument, hence the output (2,3). In the second case, string "pole" was not found anywhere, so an empty vector was returned.

2.nchar(): The call nchar(x) finds the length of a string x. Here's an example:

```
> nchar("South Pole")
[1] 10
```

The string "South Pole" was found to have 10 characters. C programmers, take note: There is no NULL character terminating R strings.

Also note that the results of nchar() will be unpredictable if x is not in character mode. For instance, nchar(NA) turns out to be 2, and nchar(factor("abc")) is 1. For more consistent results on nonstring objects, use Hadley Wickham's stringr package on CRAN.

3.paste()

The call paste(...) concatenates several strings, returning the result in one long string. Here are some examples:

```
> paste("North","Pole")
[1] "North Pole"
> paste("North","Pole",sep="")
[1] "NorthPole"
> paste("North","Pole",sep=".")
[1] "North.Pole"
> paste("North","and","South","Poles")
[1] "North and South Poles"
```

4.sprintf()

The call sprintf(...) assembles a string from parts in a formatted manner. Here's a simple example:

```
> i <- 8
> s <- sprintf("the square of %d is %d",i,i^2)
> s
[1] "the square of 8 is 64"
```

The name of the function is intended to evoke *string print* for "printing" to a string rather than to the screen. Here, we are printing to the string s.

What are we printing? The function says to first print "the square of" and then print the decimal value of i. (The term *decimal* here means in the base-10 number system, not that there will be a decimal point in the result.) The result is the string "the square of 8 is 64."

5.substr()

The call substr(x,start,stop) returns the substring in the given character position range start:stop in the given string x. Here's an example:

```
> substring("Equator",3,5) [1] "uat"
```

6.strsplit(): The call strsplit(x,split) splits a string x into an R list of substrings based on another string split in x. Here's an example:

```
> strsplit("6-16-2011",split="-")
```

```
[[1]]
[1] "6"    "16" "2011"
```

regexpr()

The call regexpr(pattern,text) finds the character position of the first instance of pattern within text, as in this example:

```
> regexpr("uat","Equator")
[1] 3
```

This reports that "uat" did indeed appear in "Equator," starting at character position 3.

gregexpr() The call gregexpr(pattern,text) is the same as regexpr(), but it finds all instances of pattern. Here's an example:

```
> gregexpr("iss","Mississippi")
[[1]]
[1] 2 5
```

This finds that "iss" appears twice in "Mississippi," starting at character positions 2 and 5.

Regular Expressions

When dealing with string-manipulation functions in programming languages, the notion of *regular expressions* sometimes arises. In R, you must pay attention to this point when using the string functions grep(), grepl(), regexpr(), gregexpr(), sub(), gsub(), and strsplit().

A regular expression is a kind of wild card. It's shorthand to specify broad classes of strings. For example, the expression "[au]" refers to any string that contains either of the letters *a* or *u*. You could use it like this:

```
> grep("[au]",c("Equator","North Pole","South Pole"))
[1] 1 3
```

This reports that elements 1 and 3 of ("Equator","North Pole","South Pole")—that is, "Equator" and "South Pole"—contain either an *a* or a *u*.

A period (.) represents any single character. Here's an example of using it:

```
> grep("o.e",c("Equator","North Pole","South Pole"))
[1] 2 3
```

This searches for three-character strings in which an *o* is followed by any single character, which is in turn followed by an *e*. Here is an example of the use of two periods to represent any pair of characters:

```
> grep("N..t",c("Equator","North Pole","South Pole"))
[1] 2
```

Here, we searched for four-letter strings consisting of an *N*, followed by any pair of characters, followed by a *t*.

A period is an example of a *metacharacter*, which is a character that is not to be taken literally. For example, if a period appears in the first argument of grep(), it doesn't actually mean a period; it means any character.

But what if you want to search for a period using grep()? Here's the naive approach:

```
> grep(".",c("abc","de","f.g"))
[1] 1 2 3
```

The result should have been 3, not (1,2,3). This call failed because periods are metacharacters. You need to *escape* the metacharacter nature of the period, which is done via a backslash:

```
> grep("\\.",c("abc","de","f.g"))

[1] 3
```

UNIT-III

DOING MATH AND SIMULATIONS IN R MATH FUNCTIONS:

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

- exp(): Exponential function, base e
- log(): Natural logarithm
- log10(): Logarithm base 10
- sqrt(): Square root
- abs(): Absolute value
- sin(), cos(), and so on: Trig functions
- min() and max(): Minimum value and maximum value within a vector
- which.min() and which.max(): Index of the minimal element and maximal element of a vector
- pmin() and pmax(): Element-wise minima and maxima of several vectors
- sum() and prod(): Sum and product of the elements of a vector
- cumsum() and cumprod(): Cumulative sum and product of the elements of a vector
- round(), floor(), and ceiling(): Round to the closest integer, to the closest integer below, and to the closest integer above
- factorial(): Factorial function

Extended Example: Calculating a Probability

As our first example, we'll work through calculating a probability using the prod() function. Suppose we have *n* independent events, and the *i*th event has the probability p_i 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 $p_A(1 - p_B)(1 - p_C)$, 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 *i*th 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 p_i 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)
}
```

How does it work? Well, the assignment

```
notp <- 1 - p
```

creates a vector of all the "not occur" probabilities $1 - p_j$, using recycling. The expression notp[-i] computes the product of all the elements of notp, except the i^{th}—exactly what we need.

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.

Minima and Maxima:

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. Here's an example:

```
> z
     [,1] [,2]
[1,]    1    2
[2,]    5    3
[3,]    6    2
> min(z[,1],z[,2])
[1] 1
> pmin(z[,1],z[,2])
[1] 1 3 2
```

In the first case, min() computed the smallest value in (1,5,6,2,3,2). But the call to pmin() computed the smaller of 1 and 2, yielding 1; then the smaller of 5 and 3, which is 3; then finally the minimum of 6 and 2, giving 2. Thus, the call returned the vector (1,3,2).

You can use more than two arguments in pmin(), like this:

```
> pmin(z[1,],z[2,],z[3,])
[1] 1 2
```

The 1 in the output is the minimum of 1, 5, and 6, with a similar computation leading to the 2.

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

Function minimization/maximization can be done via nlm() and optim(). For example, let's find the smallest value of $f(x) = x^2 - \sin(x)$.

```
> 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
```

the minimum value was found to be approximately −0.23, occurring at $x = 0.45$. A NewtonRaphson method (a technique from numerical analysis for approximating roots) is used, running five iterations in this case.

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

```
> 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

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

and

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

You can find R packages for differential equations (odesolve), 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); see Appendix B.

Sorting: Ordinary numerical sorting of a vector can be done with the sort() function, as in this 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 philosophy.

If you want the indices of the sorted values in the original vector, use the order() function. Here's an 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.

You can use order(), together with indexing, to sort data frames, like this:

```
> y
    V1 V2
1   def 2
2   ab 5
3   zzzz 1
> r <- order(y$V2)
> r
[1] 3 1 2
> z <- y[r,] > z V1 V2
3 zzzz 1
1   def 2
```

```
2   ab 5
```

What happened here? We called order() on the second column of y, yielding a vector r, telling us where numbers should go if we want to sort them. The 3 in this vector tells us that x[3,2] is the smallest number in x[,2]; the 1 tells us that x[1,2] is the second smallest; and the 2 tells us that x[2,2] is the third smallest. We then use indexing to produce the frame sorted by column 2, storing it in z.

You can use order() to sort according to character variables as well as numeric ones, as follows:

```
> d kids ages
1  Jack      12
2  Jill 10 3 Billy 13
> d[order(d$kids),] kids ages
3  Billy     13
1  Jack      12
2  Jill      10
> d[order(d$ages),] kids ages
 2 Jill    10
1 Jack   12
3 Billy   13
```

A related function is rank(), which reports the rank of each element of a vector.

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

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
```

If you wish to compute the inner product (or dot product) of two vectors, use crossprod(), like this:

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

The function computed 1 · 5 + 2 · 12 + 3 · 13 = 68.

Note that the name crossprod() is a misnomer, as the function does not compute the vector cross product. We'll develop a function to compute real cross products in Section 8.4.1.

For matrix multiplication in the mathematical sense, the operator to use is %*%, not *. For instance, here we compute the matrix product:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 3 & 1 \end{pmatrix}$$

Here's the code:

```
> a
     [,1] [,2]
[1,]    1    2
[2,]    3    4
> b
     [,1] [,2]
[1,]    1   -1
[2,]    0    1
> a %*% b
     [,1] [,2]
[1,]    1    1
[2,]    3    1
```

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

$$x_1 + x_2 = 2 \quad -x_1 + x_2 = 4$$

Its matrix form is as follows:

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

Here's the code:

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

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 (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.

```
> m
     [,1] [,2]
[1,]    1    2
[2,]    7    8
> dm <- diag(m)
> dm
[1] 1 8
> diag(dm)
     [,1] [,2]
[1,]    1    0
[2,]    0    8
> 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.

```
> 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 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 (to the "+" function).

*** Set Operations in R ***

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

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
```

that you can write your own binary operations. For instance, consider coding 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, the code consists of easy calls to setdiff() and union(), as follows:

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

Let’s try it.

```
> x
```

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

[1] 2 8Here’s another example: a binary operand for determining whether one set u is a subset of another set v. A bit of thought shows that this property is equivalent to the intersection of u and v being equal to u. Hence we have another easily coded function:

```
> "%subsetof%" <- function(u,v) {
+ return(setequal(intersect(u,v),u))
+ }
> c(3,8) %subsetof% 1:10 [1] TRUE
> c(3,8) %subsetof% 5:10
[1] FALSE
```

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
```

****SIMULATION IN R PROGRAMMING****

Functions for Statistical Distributions

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.

Table 8-1: 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()

As an 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.

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
> rnorm(10)

 [1] -0.30125091 -0.23552783 -0.61879922 0.23110361 1.31541443 0.41800526
 [7] 0.39434739 1.53440933 -0.37052444 0.03481197

> rnorm(10)

 [1] -1.25067676 -1.60956084 0.29984995 -0.58450931 -1.03388333 1.40477092
 [7] -0.04511211 1.21148672 0.15405686 -0.42036729

> set.seed(18749)
> rnorm(10)

 [1] -0.54555908 1.22938955 2.58249311 1.57057115 -0.03186376 0.32983807
 [7] -0.44899542 0.56599635 -0.23902963 0.79350037
```

```
> set.seed(18749)
> rnorm(10)

 [1] -0.54555908 1.22938955 2.58249311 1.57057115 -0.03186376 0.32983807
 [7] -0.44899542 0.56599635 -0.23902963 0.79350037
```

It is often useful to be able to randomly draw from a set of numbers we have already (for example when bootstrapping). This is done using the function sample. We supply sample with a vector of numbers to draw from. Without any other arguments sample will randomly permute the vector. If in addition we supply an argument size, sample will randomly draw size numbers from our vector without replacement. There is also an argument replace that specifies whether drawing should be with replacement (by defualt without replacement). The argument prob can be used to specify a probability distribution other than uniform with which to draw the numbers.

Example

```
> sample(1:10)

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

> sample(1:10, size = 5)

[1] 5 8 6 1 10

> sample(1:10, size = 15, replace = TRUE)

 [1] 9 5 1 4 5 10 1 2 1 8 1 2 7 5 10
```

Basic functions
Being a statistical package R has plenty of built in functions for preforming basic statistical operations. Some are given below and you can probably guess many others.

Example

```
> x <- rnorm(20, mean = 10, sd = 20) > mean(x)

[1] 8.144194

> sd(x)

[1] 17.36403

> sqrt(var(x))

[1] 17.36403

> median(x)

[1] 8.448636

> quantile(x, probs = c(0.05, 0.95))
        5%       95%
-18.35355 31.73562

> summary(x)
```

```
   Min. 1st Qu. Median    Mean 3rd Qu.    Max.
-18.630 -6.728 8.449 8.144 25.220 36.880
```

There are also some basic mathemathical functions that will be useful.

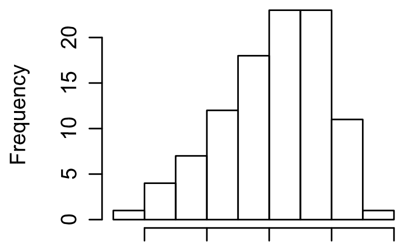
```
> y <- 1:20
> length(y)

[1] 20

> max(y)

[1] 20

> min(y)

[1] 1

> log(y)

 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851 2.3978953 2.4849066 2.5649494 2.6390573
[15] 2.7080502 2.7725887 2.8332133 2.8903718 2.9444390 2.9957323

> sum(y)

[1] 210

> prod(y)

[1] 2.432902e+18
```

Plots

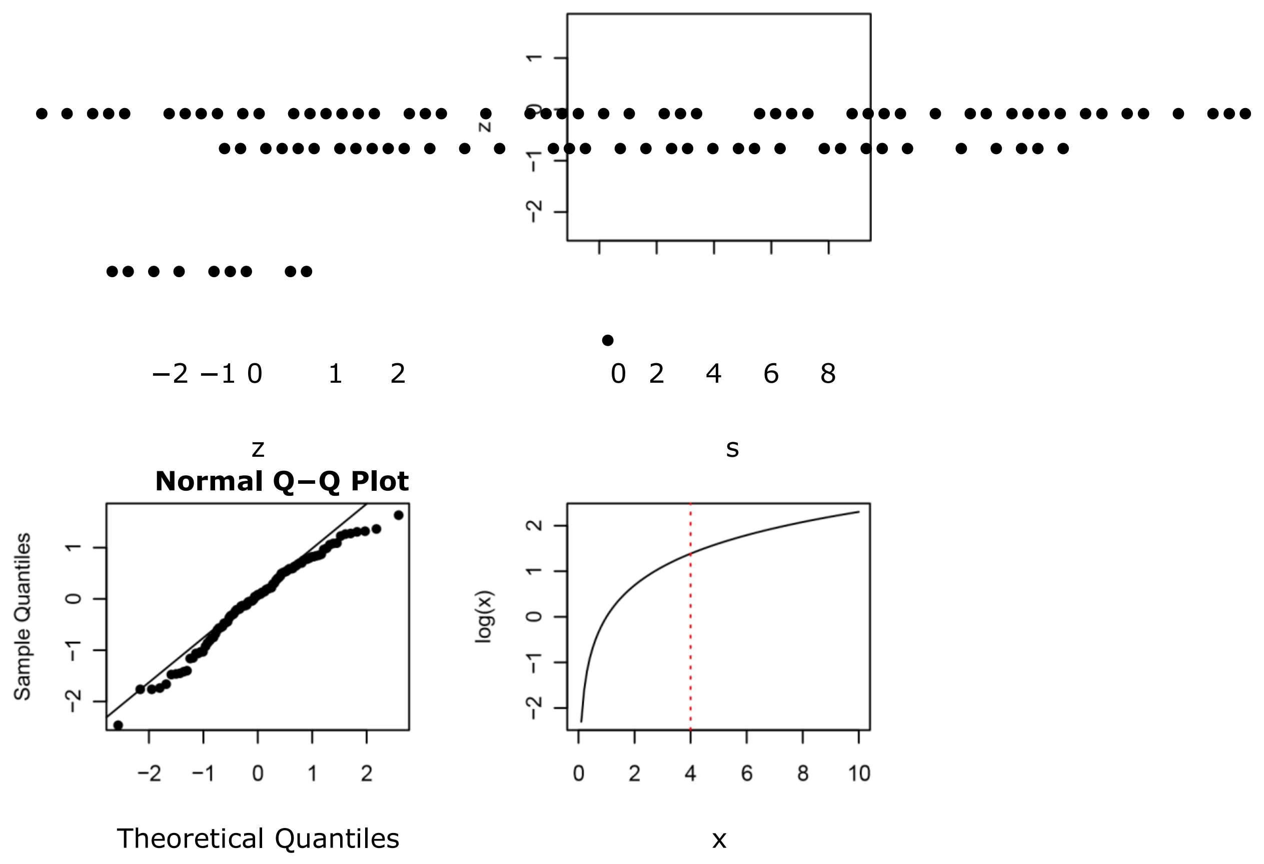
There are lots of ways to create plots in R. Functions like hist, boxplot and qqnorm produce standard statistical plots given some data. plot can be adjusted to plot a lot of different types of plot (and there are often defaults for different R objects, for example try plot(log)). There are also a number of functions that will add things to a plot: points, lines, text. The most useful one for the exercises is abline which will add a straight line to a plot.

```
> z <- rnorm(100)
> s <- 2 * z + 4 + rnorm(100)
> par(mfrow = c(2, 2))
> hist(z)
> plot(s, z)
> qqnorm(z)
> qqline(z)
> curve(log(x), from = 0, to = 10)
                                        > abline(v = 4, col = "red", lty = "dashed")
```

Histogram of z



●



Optimizing functions:
Most of the functions in R for optimization are minimizers. You generally have to rewrite your problem so that it is one of minimization. For example, to maximise a likelihood you minimize the negative likelihood. Or, to find a root of a function you could minimize its square. There are many functions that will do this: nlm, optim, optimize, nlminb and constrOptim. My favourite is nlminb because it handles both one parameter and multiparameter problems and it is easy to place constraints on the parameters. At the very least you need to give the function a place to start and a function to minimize.

Example - Maximum Likelihood

Imagine we have 20 observations from an exponential distribution with unknown parameter λ (we'll simulate this data). We want to find the maximum likelihood estimate for λ We know the density for an exponential distribution is

$$f(x|\lambda) = \lambda e^{-\lambda x} \quad x \geq 0.$$

We can write the log likelihood as,

$$l(\lambda) = \sum_{i=1}^{20} (\log \lambda - \lambda x_i)$$

$$= n\log\lambda - \lambda \sum_{i=1}^{20} x_i$$

So in R we want to minimize the negative of this function,

```
> x <- rexp(20, rate = 4)
> n <- length(x)
> nllhood = function(lambda) {
+      -1 * (n * log(lambda) - lambda * sum(x))
+ }
> fit <- nlminb(6, nllhood)
> fit

$par
[1] 3.491674

$objective
[1] -5.007625

$convergence [1] 0

$message
[1] "relative convergence (4)"

$iterations [1] 6

$evaluations function gradient
         8          9

> fit$par

[1] 3.491674 apply
```

apply is an incredibly useful function when you are making the same calculation repeatedly over the columns or rows of an object. It takes three arguments. The first the object you wish to apply the function to, the second either 1 or 2 depending on whether you are working across the rows or down the columns, and the third the function you wish to apply.

Example

```
> x <- matrix(rep(1:5, 5), ncol = 5)
> x
      [,1] [,2] [,3] [,4] [,5]
[1,]     1     1     1     1     1

[2,]     2     2     2     2     2

[3,]     3     3     3     3     3

[4,]     4     4     4     4     4

[5,]     5     5     5     5     5

> apply(x, 1, mean)

[1] 1 2 3 4 5

> apply(x, 2, mean)

[1] 3 3 3 3 3

> apply(x, 2, function(y) c(mean(y), sd(y)))
```

```
            [,1]        [,2]        [,3]        [,4]        [,5]
[1,] 3.000000 3.000000 3.000000 3.000000 3.000000 [2,] 1.581139 1.581139 1.581139
1.581139 1.581139 Also look at lapply, sapply, tapply and mapply.
```

Reading in data

```
> p.data <- read.table("http://www.stat.berkeley.edu/~wickham/poisson.txt")
> head(p.data)
  V1
1  4
2  2
3  4
4  4
5  11
6  8
```

**** INPUT/OUTPUT OPERATOINS IN R ****

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.

Using the scan() Function

You can use scan() to read in a vector, whether numeric or character, from a file or the keyboard. With a little extra work, you can even read in data to form a list.

Suppose we have files named *z1.txt*, *z2.txt*, *z3.txt*, and *z4.txt*. The *z1.txt* file contains the following:

```
>z1.txt
        123
        4 5
        6
```

The *z2.txt* file contents are as follows: >z2.txt

```
        123
        4.2 5
        6
```

The *z3.txt* file contains this: >z3.txt

```
        abc de f g
```

And finally, the *z4.txt* file has these contents: z4.txt abc 123 6 y

Let's see what we can do with these files using the scan() function.

```
> scan("z1.txt")
Read 4 items
[1] 123 4 5 6
> 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, : scan()
  expected 'a real', got 'abc'
> scan("z3.txt",what="")
Read 4 items
[1] "abc" "de" "f" "g"
> scan("z4.txt",what="")
Read 4 items
[1] "abc" "123" "6" "y"
```

In the first call, we got a vector of four integers (though the mode is numeric). The second time, since one number was nonintegral, the others were shown as floatingpoint numbers, too.

In the third case, we got an error. The scan() function has an optional argument named what, which specifies mode, defaulting to double mode. So, the nonnumeric contents of the file *z3* produced an error. But we then tried again, with what="". This assigns a character string to what, indicating that we want character mode. (We could have set what to any character string.)

```
> v <- scan("z1.txt")
```

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. As example, we can set sep to the newline character to read in each line as a string, as follows:

```
> x1 <- scan("z3.txt",what="")
Read 4 items
> x2 <- scan("z3.txt",what="",sep="\n")
Read 3 items
> x1
[1] "abc" "de" "f" "g"
> x2
[1] "abc" "de f" "g"
> x1[2]
[1] "de"
> x2[2]
[1] "de f"
```

In the first case, the strings "de" and "f" were assigned to separate elements of x1. But in the second case, we specified that elements of x2 were to be delineated by end-of-line characters, not spaces. Since "de" and "f" are on the same line, they are assigned together to x[2].

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.

If you do not wish scan() to announce the number of items it has read, include the argument quiet=TRUE.

Using the readline() Function

If you want to read in a single line from the keyboard, readline() is very handy.

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

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

```
> inits <- readline("type your initials: ") type your initials:
NM
> inits
[1] "NM"
```

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, like this:

```
> 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" > 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.

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
```

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

Now that we’ve covered the basics of I/O, let’s get to some more practical applications of reading and writing files. The following sections discuss 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

In Section 5.1.2, we discussed the use of the function read.table() to read in a data frame. As a quick review, suppose the file *z* looks like this:

```
name age John 25
Mary 28
Jim 19
```

The first line contains an optional header, specifying column names. We could read the file this way:

```
> z <- read.table("z",header=TRUE)
> z         name age
1    John   25
2    Mary   28
3    Jim     19
```

Reading Text Files

In computer literature, there is often a distinction made between *text files* and *binary files*. That distinction is somewhat misleading—every file is binary in the sense that it consists of 0s and 1s. Let’s take the term *text file* to mean a file that consists mainly of ASCII characters or coding for some other human language (such as GB for Chinese) and that uses newline characters to give humans the perception of lines. The latter aspect will turn out to be central here. Nontext files, such as JPEG images or executable program files, are generally called *binary files*.

You can use readLines() 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"
```

Extra work: imp questions

S4 Classes

Some programmers feel that S3 does not provide the safety normally associated with OOP. For example, consider our earlier employee database example, where our class "employee" had three fields: name, salary, and union. Here are some possible mishaps:

- We forget to enter the union status.
- We misspell *union* as *onion*.
- We create an object of some class other than "employee" but accidentally set its class attribute to "employee".

In each of these cases, R will not complain. The goal of S4 is to elicit a complaint and prevent such accidents.

S4 structures are considerably richer than S3 structures, but here we present just the basics. Table 9-1 shows an overview of the differences between the two classes.

Table 9-1: Basic R Operators

Operation	**S3**	**S4**
Define class	Implicit in constructor code	setClass()
Create object	Build list, set class attr	new()
Reference member variable	$	@
Implement generic f()	Define f.classname()	setMethod()
Declare generic	UseMethod()	setGeneric()

Writing S4 Classes

You define an S4 class by calling setClass(). Continuing our employee example, we could write the following:

```
> setClass("employee",
 + representation(
+    name="character",
+     salary="numeric",
+      union="logical")
+ )
[1] "employee"
```

This defines a new class, "employee", with three member variables of the specified types.

Now let's create an instance of this class, for Joe, using new(), a built-in constructor function for S4 classes:

```
> joe <- new("employee",name="Joe",salary=55000,union=T)
> joe
An object of class "employee" Slot "name":
[1] "Joe"
Slot "salary": [1] 55000

Slot "union":
[1] TRUE
```

Note that the member variables are called *slots*, referenced via the @ symbol. Here's an example:

```
> joe@salary
[1] 55000
```

We can also use the slot() function, say, as another way to query Joe's salary:

```
> slot(joe,"salary")
[1] 55000
```

We can assign components similarly. Let's give Joe a raise:

```
> joe@salary <- 65000
> joe
An object of class "employee" Slot
"name": [1] "Joe" Slot "salary": [1] 65000

Slot "union":
[1] TRUE
```

Nah, he deserves a bigger raise that that:

```
> slot(joe,"salary") <- 88000
> joe
An object of class "employee" Slot
"name": [1] "Joe" Slot "salary": [1] 88000

Slot "union":
[1] TRUE
```

As noted, an advantage of using S4 is safety. To illustrate this, suppose we were to accidentally spell *salary* as *salry*, like this:

```
> joe@salry <- 48000
Error in checkSlotAssignment(object, name, value) :
  "salry" is not a slot in class "employee"
```

By contrast, in S3 there would be no error message. S3 classes are just lists, and you are allowed to add a new component (deliberately or not) at any time.

9.2.2Implementing a Generic Function on an S4 Class

To define an implementation of a generic function on an S4 class, use setMethod(). Let’s do that for our class "employee" here. We’ll implement the show() function, which is the S4 analog of S3’s generic "print".

As you know, in R, when you type the name of a variable while in interactive mode, the value of the variable is printed out:

```
> joe
An object of class "employee" Slot
"name": [1] "Joe"

Slot "salary": [1] 88000
Slot "union":
[1] TRUE
```

Since joe is an S4 object, the action here is that show() is called. In fact, we would get the same output by typing this:

```
> show(joe)
```

Let’s override that, with the following code:

```
setMethod("show", "employee", function(object) { inorout <-
   ifelse(object@union,"is","is not") cat(object@name,"has a
      salary of",object@salary,
         "and",inorout, "in the union", "\n")
   }
)
```

The first argument gives the name of the generic function for which we will define a class-specific method, and the second argument gives the class name. We then define the new function.

```
> joe
Joe has a salary of 55000 and is in the union
```

S3 Versus S4

The type of class to use is the subject of some controversy among R programmers. In essence, your view here will likely depend on your personal choice of which you value more—the convenience of S3 or the safety of S4.

John Chambers, the creator of the S language and one of the central developers of R, recommends S4 over S3 in his book *Software for Data Analysis* (Springer, 2008). He argues that S4 is needed in order to write “clear and reliable software.” On the other hand, he notes that S3 remains quite popular.

Google’s R Style Guide, which you can find at *http://google-styleguide .googlecode.com/svn/trunk/google-r-style.html*, is interesting in this regard. Google comes down squarely on the S3 side, stating “avoid S4 objects and methods when possible." (Of course, it's also interesting that Google even has an R style guide in the first place!)

Managing Your Objects

As a typical R session progresses, you tend to accumulate a large number of objects. Various tools are available to manage them. Here, we'll look at the following:

- The ls() function
- The rm() function
- The save() function
- Several functions that tell you more about the structure of an object, such as class() and mode()
- The exists() function

Listing Your Objects with the ls() Function

The ls() command will list all of your current objects. A useful named argument for this function is pattern, which enables *wildcards*. Here, you tell ls() to list only the objects whose names include a specified pattern. The following is an example.

```
> ls()

[1] "acc"        "acc05" "binomci" "cmeans" "divorg" "dv"
[7] "fit"    "g"      "genxc" "genxnt" "j"  "lo" [13] "out1" "out1.100" "out1.25"
"out1.50" "out1.75" "out2"
[19] "out2.100" "out2.25"          "out2.75" "par.set" "prpdf"
                          "out2.50"
[25] "ratbootci" "simonn"          "x"          "zout"     "zout.100"
                          "vecprod"
[31] "zout.125" "zout3" "zout5" "zout.50" "zout.75"

> ls(pattern="ut")
[1] "out1" "out1.100" "out1.25" "out1.50" "out1.75" "out2"
[7] "out2.100" "out2.25" "out2.50" "out2.75" "zout" "zout.100"
[13] "zout.125" "zout3" "zout5" "zout.50" "zout.75"
```

In the second case, we asked for a list of all objects whose names include the string "ut".

9.4.2 Removing Specific Objects with the rm() Function To remove objects you no longer need, use rm(). Here's an example:

```
> rm(a,b,x,y,z,uuu)
```

This code removes the six specified objects (a, b, and so on).

One of the named arguments of rm() is list, which makes it easier to remove multiple objects. This code assigns all of our objects to list, thus removing everything:

```
> rm(list = ls())
```

Using ls()’s pattern argument, this tool becomes even more powerful. Here’s an example:

```
>ls()
[1] "doexpt"        "notebookline" "nreps"           "numcorrectcis"
 [5] "numnotebooklines"           "observationpt" "prop"
 "numrules"
[9] "r"                 "rad"           "radius"          "rep"
[13] "s"                "s2"            "sim"              "waits"
[17] "wbar"           "x"              "y"                "z"
> ls(pattern="notebook")
[1] "notebookline" "numnotebooklines"
> rm(list=ls(pattern="notebook"))
> ls()
[1] "doexpt"      "nreps"         "numcorrectcis"
                                  "numrules"
 [5] "observationpt" "prop"      "r"              "rad"
[9] "radius"       "rep"           "s"               "s2"
[13] "sim"         "waits"         "wbar"         "x"
[17] "y"             "z"
```

Here, we found two objects whose names include the string "notebook" and then asked to remove them, which was confirmed by the second call to ls().

Saving a Collection of Objects with the save() Function

Calling save() on a collection of objects will write them to disk for later retrieval by load(). Here’s a quick example:

```
> z <- rnorm(100000)
> hz <- hist(z)
> save(hz,"hzfile")
> ls()
[1] "hz" "z"
> rm(hz)
> ls()
[1] "z"
> load("hzfile")
> ls()
[1] "hz" "z"
> plot(hz) # graph window pops up
```

Here, we generate some data and then draw a histogram of it. But we also save the output of hist() in a variable, hz. That variable is an object (of class "histogram", of course). Anticipating that we will want to reuse this object in a later R session, we use the save() function to save the object to the file *hzfile*. It can be reloaded in that future session via load(). To demonstrate this, we deliberately removed the hz object, then called load() to reload it, and then called ls() to show that it had indeed been reloaded.

I once needed to read in a very large data file, each record of which required processing. I then used save() to keep the R object version of the processed file for future R sessions.

UNIT-IV GRAPHICS IN R PROGRAMMING

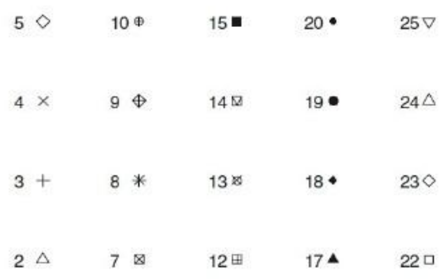
R has a very rich set of graphics facilities.

Graphics in R: A Graphics Is An Image Or Visual Representation Of An Object. Graphics Are Often
Contrasted With Text, Which Comprised Of Characters, Such As Numbers And Letters, Rather Than Image. Graphics Is An Art Of Drawing, Especially As Used In Mathematics& Engineering.
R programming have many graphic functions:
1.Plot(): The plot() function works in stages, which means you can build up a graph in stages by issuing a series of commands. plot function is the basic graphical tool plot(x,y)
2.Hist():This are used to create histograms
Another examples:
3.Linechart(): This is used for to create linecharts **4.Barchart():**This is used for to create barcharts
5.Pie(): This is used for to create piecharts.
6.Boxplot(): This is used for to create boxplots.
7.Scatterplots(): This is used for to create scatterplots.

Creating Graphs using plot() function in R: To begin, we'll look at the foundational function for creating graphs: plot(). Then we'll explore how to build a graph, from adding lines and points to attaching a legend.

The plot() Function:The plot() function forms the foundation for much of R's base graphing operations, serving as the vehicle for producing many different kinds of graphs. , plot() is a generic function, or a placeholder for a family of functions.

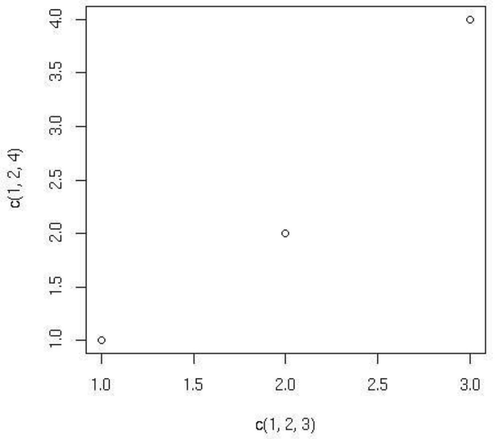
- plot function is the basic graphical tool. plot(x, y) or plot(y x)
- The 1st argument is displayed on the horizontal axis
- The 2nd argument is put on the vertical axis.
- It is conventional to plot the response variable along the vertical axis and the explanatory variable along the horizontal axis.
- The most common modifications to plot are adding a title and x- and y-labels and setting the x- and y-limits.
- dot symbols



```
> plot(c(1,2,3), c(1,2,4))
```

This will cause a window to pop up, plotting the points (1,1), (2,2), and (3,4), as shown in Figure 12-1. As you can see, this is a very plain-Jane graph.

We'll discuss adding some of the fancy bells and whistles later in the chapter.



Simple point plot

The plot() function works in stages, which means you can build up a graph in stages by issuing a series of commands. For example, as a base, we might first draw an empty graph, with only axes, like this:

```
> plot(c(-3,3), c(-1,5), type = "n", xlab="x", ylab="y")
```

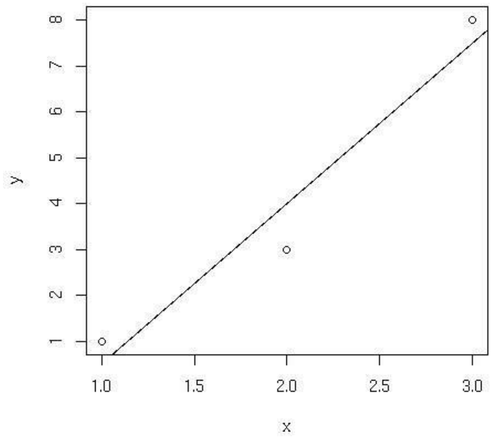
This draws axes labeled *x* and *y*. The horizontal (*x*) axis ranges from −3 to 3. The vertical (*y*) axis ranges from −1 to 5. The argument type="n" means that there is nothing in the graph itself.

Adding Lines: The abline() Function:

We now have an empty graph, ready for the next stage, which is adding a line:

```
> x <- c(1,2,3) > y <- c(1,3,8)
> plot(x,y)
> lmout <- lm(y ~ x)
> abline(lmout)
```

After the call to plot(), the graph will simply show the three points, along with the *x*- and *y*- axes with hash marks. The call to abline() then adds a line to the current graph. Now, which line is this?



Using abline()

```
> lines(c(1.5,2.5),c(3,3))
```

If you want the lines to "connect the dots," but don't want the dots themselves, include type="l" in your call to lines() or to plot(), as follows:

```
> plot(x,y,type="l")
```

You can use the lty parameter in plot() to specify the type of line, such as solid or dashed. To see the types available and their codes, enter this command:

```
> help(par)
```

- Adding a Smoothing Line by lines(). The lines( ) function adds information to a graph. It can not produce a graph on its own. Usually it follows a plot(x, y) command that produces a graph.
- A histogram plots the frequencies that data appears within certain ranges.
- A boxplot provides a graphical view of the median, quartiles, maximum, and minimum of a data set.
- Normal quantile plot. This plot is used to determine if your data is close to being normally distributed.
- qqplot() function creates a Quantile-Quantile plot for any theoretical distribution.

Starting a New Graph While Keeping the Old Ones

Each time you call plot(), directly or indirectly, the current graph window will be replaced by the new one. If you don't want that to happen, use the command for your operating system:

- On Linux systems, call X11().
- On a Mac, call macintosh().
- On Windows, call windows().

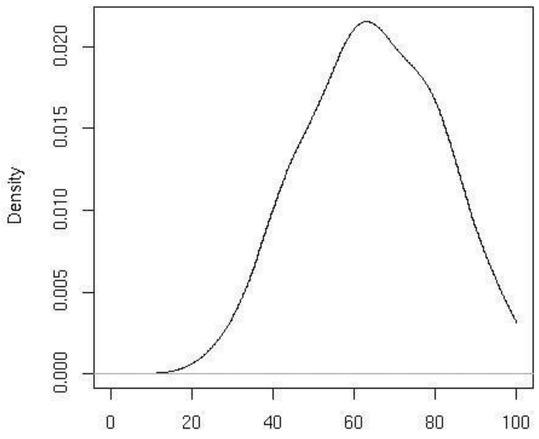
For instance, suppose you wish to plot two histograms of vectors X and Y and view them side by side. On a Linux system, you would type the following: > hist(x) > x11()
> hist(y)

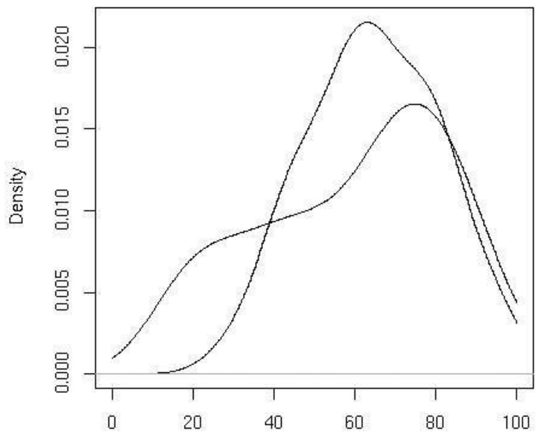
Extended Example: Two Density Estimates on the Same Graph:

Let's plot nonparametric density estimates (these are basically smoothed histograms) for two sets of examination scores in the same graph. We use the function density() to generate the estimates. Here are the commands we issue:

```
> d1 = density(testscores$Exam1,from=0,to=100)
> d2 = density(testscores$Exam2,from=0,to=100)
> plot(d1,main="",xlab="")
> lines(d2)
```

First, we compute nonparametric density estimates from the two variables, saving them in objects d1 and d2 for later use. We then call plot() to draw the curve for exam 1, at which point the plot looks like Figure 12-3. We then call lines() to add exam 2's curve to the graph, producing Figure 12-4.





Plot of first density PLOT FOR SECOND DENCITY

Adding Points: The points() Function

The points() function adds a set of (*x*,*y*) points, with labels for each, to the currently displayed graph. For instance, in our first example, suppose we entered this command:

```
points(testscores$Exam1,testscores$Exam3,pch="+")
```

The result would be to superimpose onto the current graph the points of the exam scores from that example, using plus signs (+) to mark them.

As with most of the other graphics functions, there are many options, such as point color and background color. For instance, if you want a yellow background, type this command:

```
> par(bg="yellow")
```

Now your graphs will have a yellow background, until you specify otherwise.

As with other functions, to explore the myriad of options, type this:

```
>help(par)
```

Adding a Legend: The legend() Function: The legend() function is used, not surprisingly, to add a legend to a multicurve graph. This could tell the viewer something like, "The green curve is for the men, and the red curve displays the data for the women." Type the following to see some nice examples:

R legend() function: To add **legends** to plots in **R**, the **R legend()** function can be used. A simplified format of the function is :

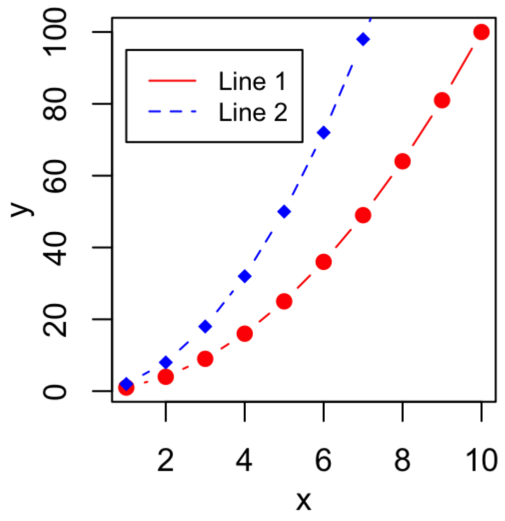
```
legend(x, y=NULL, legend, fill, col, bg)
```

- **x and y** : the x and y co-ordinates to be used to position the legend
- **legend** : the text of the legend

- **fill** : colors to use for filling the boxes beside the legend text ☐ **col** : colors of lines and points beside the legend text ☐ **bg** : the background color for the legend box.

Example :

# Generate some data x<-1:10; y1=x*x; y2=2*y1
plot(x, y1, type="b", pch=19, col="red", xlab="x", ylab="y")
# Add a line
lines(x, y2, pch=18, col="blue", type="b", lty=2)
# Add a legend
legend(1, 95, legend=c("Line 1", "Line 2"), col=c("red", "blue"), lty=1:2, cex=0.8)



To avoid repeating the above **R** code, we can create a custom plot function as follow :

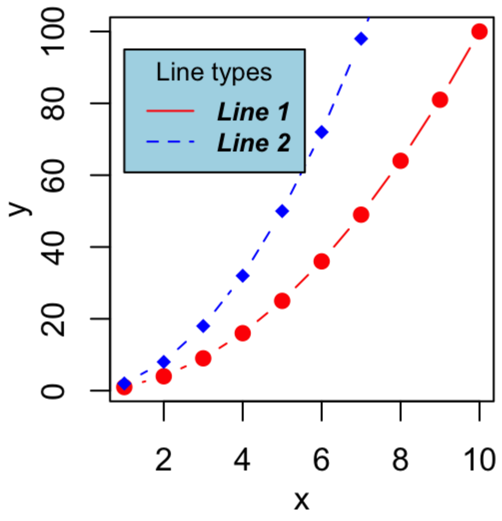
makePlot<-function(){ x<-1:10; y1=x*x; y2=2*y1
plot(x, y1, type="b", pch=19, col="red", xlab="x", ylab="y") lines(x, y2, pch=18, col="blue", type="b", lty=2) }

Title, text font and background color of the legend box

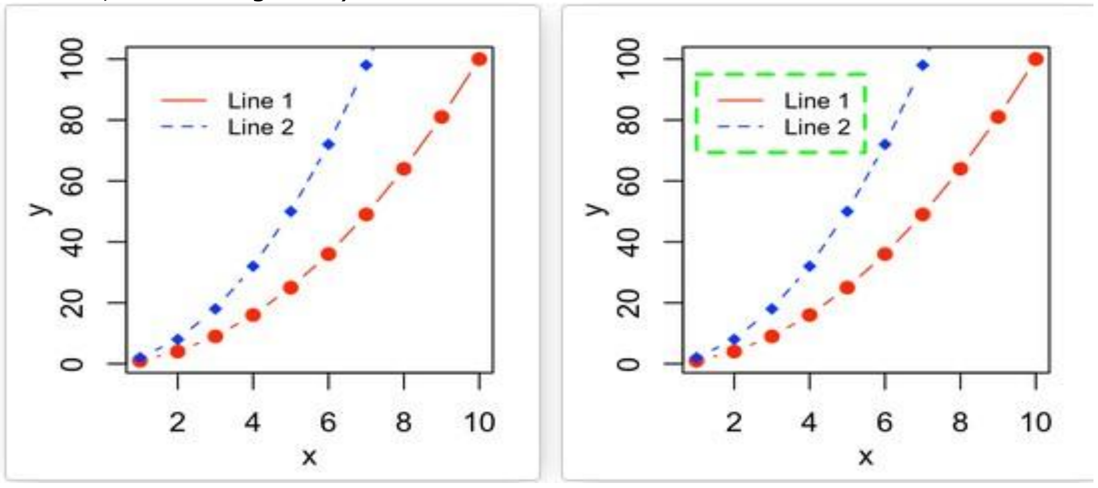
The arguments below can be used :

- **title**: The title of the legend
- **text.font**: an integer specifying the font style of the legend text; possible values are :
  - o **1**: normal o **2**: bold o **3**: italic o **4**: bold and italic
- **bg**: background color of the legend box

makePlot()
# Add a legend to the plot
legend(1, 95, legend=c("Line 1", "Line 2"), col=c("red", "blue"), lty=1:2, cex=0.8,
title="Line types", text.font=4, bg='lightblue')

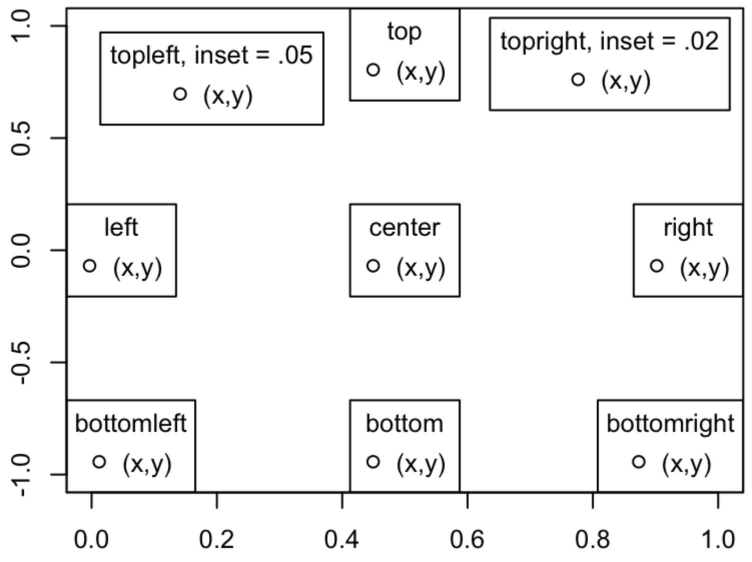


Border of the legend box: The arguments **box.lty, box.lwd and box.col** can be used to modify the line type, width and color for the legend box border, respectively.
# Remove legend border using box.lty = 0 makePlot()
legend(1, 95, legend=c("Line 1", "Line 2"), col=c("red", "blue"), lty=1:2, cex=0.8, box.lty=0) # Change the border makePlot()
legend(1, 95, legend=c("Line 1", "Line 2"), col=c("red", "blue"), lty=1:2, cex=0.8, box.lty=2, box.lwd=2, box.col="green")



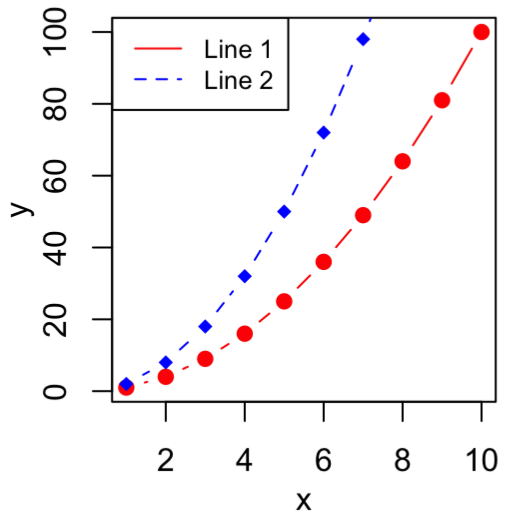
Specify legend position by keywords: The position of the legend can be specified also using the following keywords : "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right" and "center".

The effect of using each of these keywords are shown in the figure below :



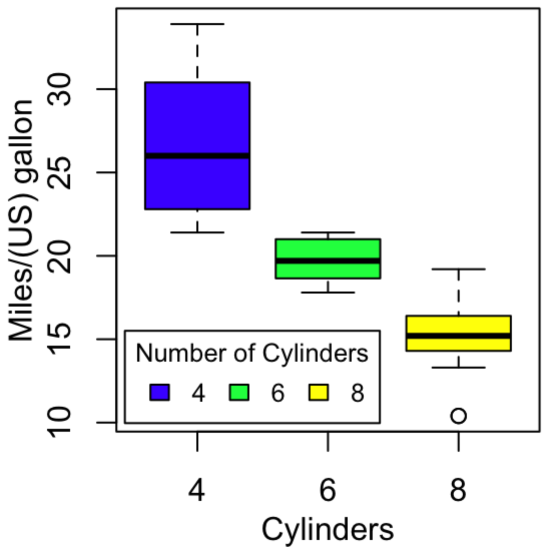
Example 1: line plot

```
#  Example 1: line plot makePlot()
legend("topleft", legend=c("Line 1", "Line 2"),       col=c("red", "blue"), lty=1:2, cex=0.8)
```



Example 2: box plot

```
attach(mtcars)
boxplot(mpg~cyl,xlab="Cylinders",ylab="Miles/(US) gallon",    col=topo.colors(3))

legend("bottomleft", inset=.02, title="Number of Cylinders",    c("4","6","8"),
fill=topo.colors(3), horiz=TRUE, cex=0.8)
```



Adding Text: The text() Function

Use the text() function to place some text anywhere in the current graph. Here's an example:

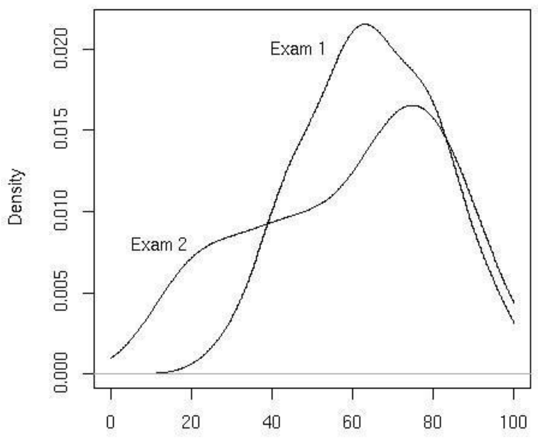
```
text(2.5,4,"abc")
```

This writes the text "abc" at the point (2.5,4) in the graph. The center of the string, in this case "b," would go at that point.

To see a more practical example, let's add some labels to the curves in our exam scores graph, as follows:

```
> text(46.7,0.02,"Exam 1")
> text(12.3,0.008,"Exam 2")
```

In order to get a certain string placed exactly where you want it, you may need to engage in some trial and error. Or you may find the locator() function to be a much quicker way to go, as detailed in the next section.



Placing text

Pinpointing Locations: The locator() Function

Placing text exactly where you wish can be tricky. You could repeatedly try different *x*- and *y*-coordinates until you find a good position, but the locator() function can save you a lot of trouble. You simply call the function and then click the mouse at the desired spot in the graph. The function returns the *x*and *y*-coordinates of your click point. Specifically, typing the following will tell R that you will click in one place in the graph:

```
locator(1)
```

Once you click, R will tell you the exact coordinates of the point you clicked. Call locator(2) to get the locations of two places, and so on. (Warn-

ing: Make sure to include the argument.) Here is a simple example:

```
> hist(c(12,5,13,25,16))
> locator(1) or locator(n=2,type="p") some result happen given below
$x
[1] 6.239237

$y
[1] 1.221038
```

This has R draw a histogram and then calls locator() with the argument 1, indicating we will click the mouse once. After the click, the function returns a list with components x and y, the *x*- and *y*-coordinates of the point where we clicked.

To use this information to place text, combine it with text():

```
> text(locator(1),"nv=75")
```

Here, text() was expecting an *x*-coordinate and a *y*-coordinate, specifying the point at which to draw the text "nv=75." The return value of locator() supplied those coordinates.

Restoring a Plot: R has no "undo" command. However, if you suspect you may need to undo your next step when building a graph, you can save it using recordPlot() and then later restore it with replayPlot().

Less formally but more conveniently, you can put all the commands you're using to build up a graph in a file and then use source(), or cut and paste with the mouse, to execute them. If you change one command, you can redo the whole graph by sourcing or copying and pasting your file.

For our current graph, for instance, we could create file named examplot.R with the following contents:

```
d1 = density(testscores$Exam1,from=0,to=100) d2 =
density(testscores$Exam2,from=0,to=100) plot(d1,main="",xlab="") lines(d2)
text(46.7,0.02,"Exam 1") text(12.3,0.008,"Exam 2")
```

If we decide that the label for exam 1 was a bit too far to the right, we can edit the file and then either do the copy-and-paste or execute the following:

```
> source("examplot.R”)
```

****Customizing Graphs****

R – Histograms: A histogram represents the frequencies of values of a variable bucketed into ranges. Histogram is similar to bar chat but the difference is it groups the values into continuous ranges. Each bar in histogram represents the height of the number of values present in that range.

R creates histogram using **hist()** function. This function takes a vector as an input and uses some more parameters to plot histograms.

Syntax: The basic syntax for creating a histogram using R is −

```
hist(v,main,xlab,xlim,ylim,breaks,col,border)
```

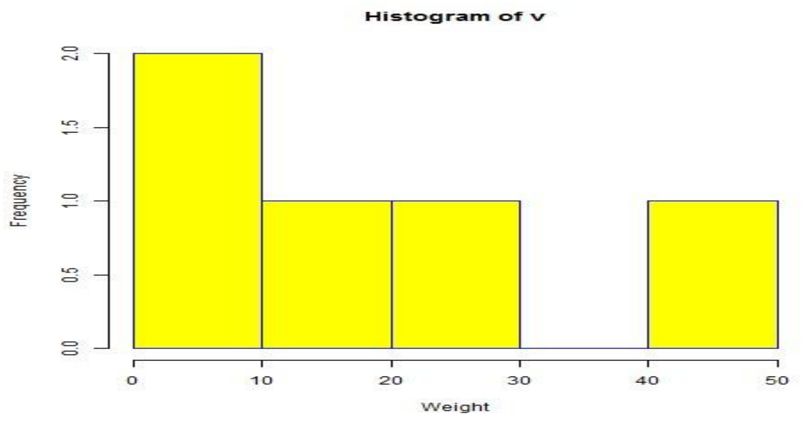
Following is the description of the parameters used − **v** is a vector containing numeric values used in histogram. **main** indicates title of the chart. **col** is used to set color of the bars. **border** is used to set border color of each bar. **xlab** is used to give description of x-axis. **xlim** is used to specify the range of values on the x-axis. **ylim** is used to specify the range of values on the y-axis. **breaks** is used to mention the width of each bar.

Example: simple histogram is created using input vector, label, col and border parameters.

The script given below will create and save the histogram in the current R working directory.

```
# Create data for the graph.
v <-  c(9,13,21,8,36,22,12,41,31,33,19) # Give the chart file a name.
png(file = "histogram.png") # Create the histogram.
hist(v,xlab = "Weight",col = "yellow",border = "blue") # Save the file.
dev.off()
```

When we execute the above code, it produces the following result −

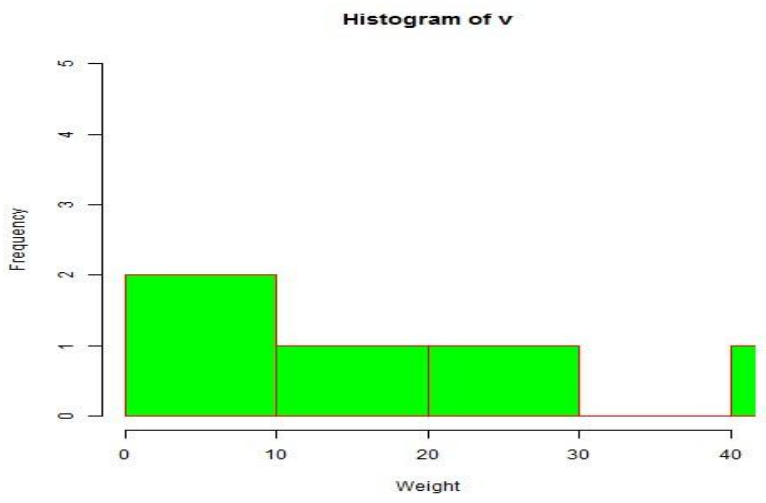


Range of X and Y values: To specify the range of values allowed in X axis and Y axis, we can use the xlim and ylim parameters.

The width of each of the bar can be decided by using breaks.

```
# Create data for the graph.
v <- c(9,13,21,8,36,22,12,41,31,33,19) # Give the chart file a name.
png(file = "histogram_lim_breaks.png") # Create the histogram.
hist(v,xlab = "Weight",col = "green",border = "red", xlim = c(0,40), ylim = c(0,5),    breaks =
5) # Save the file.
dev.off()
```

When we execute the above code, it produces the following result –



(Another parameters)Axes and Text :Many high level plotting functions (plot, hist, boxplot, etc.) allow you to include axis and text options (as well as other graphical paramters). For example

# Specify axis options within plot() plot(*x*, *y*, main="*title*", sub="*subtitle*",   xlab="*X-axis label*", ylab="*y-axix label*",   xlim=c(*xmin*, *xmax*), ylim=c(*ymin*, *ymax*))

For finer control or for modularization, you can use the functions described below.

Titles: Use the title( ) **function to add labels to a plot.**

title(main="*main title*", sub="*sub-title*",   xlab="*x-axis label*", ylab="*y-axis label*")

Many other graphical parameters (such as text size, font, rotation, and color) can also be specified in the **title( )** function.

```
# Add a red title and a blue subtitle. Make x and y
# labels 25% smaller than the default and green.
title(main="My Title", col.main="red",    sub="My Sub-title", col.sub="blue",    xlab="My X
label", ylab="My Y label",   col.lab="green", cex.lab=0.75)
```

****

R - Line Graphs:A line chart is a graph that connects a series of points by drawing line segments between them. These points are ordered in one of their coordinate (usually the x-coordinate) value. Line charts are usually used in identifying the trends in data.

The **plot()** function in R is used to create the line graph.

Syntax: The basic syntax to create a line chart in R is − plot(v,type,col,xlab,ylab)

Following is the description of the parameters used −
v is a vector containing the numeric values.

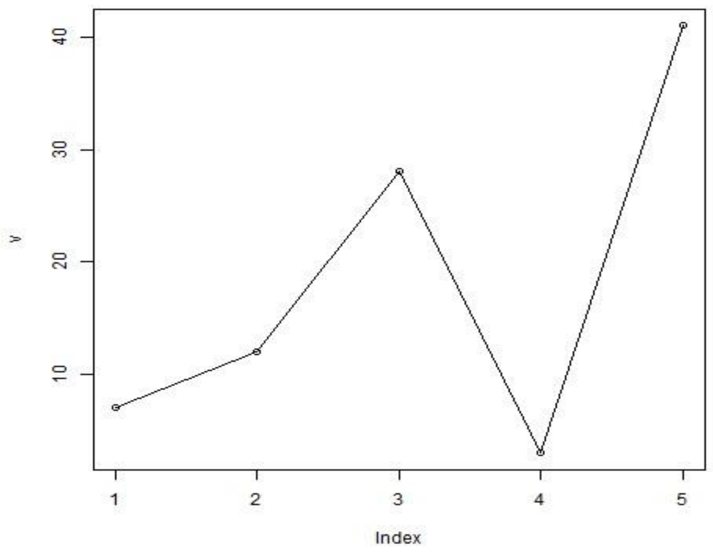
type takes the value "p" to draw only the points, "l" to draw only the lines and "o" to draw both points and lines. **xlab** is the label for x axis. **ylab** is the label for y axis. **main** is the Title of the chart. **col** is used to give colors to both the points and lines.

Example

A simple line chart is created using the input vector and the type parameter as "O". The below script will create and save a line chart in the current R working directory.

```
# Create the data for the chart.
v <- c(7,12,28,3,41)
# Give the chart file a name. png(file = "line_chart.jpg") # Plot the bar chart.
plot(v,type = "o") # Save the file.
dev.off()
```

When we execute the above code, it produces the following result −



Line Chart Title, Color and Labels

The features of the line chart can be expanded by using additional parameters. We add color to the points and lines, give a title to the chart and add labels to the axes.

Example

# Create the data for the chart.

v <- c(7,12,28,3,41)

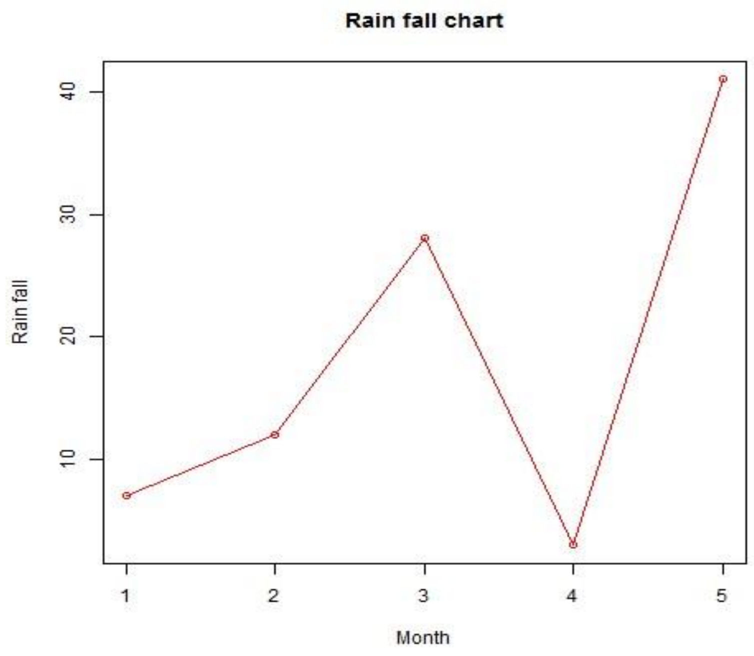
# Give the chart file a name.

png(file = "line_chart_label_colored.jpg") # Plot the bar chart.

plot(v,type = "o", col = "red", xlab = "Month", ylab = "Rain fall", main = "Rain fall chart") # Save the file.

dev.off()

When we execute the above code, it produces the following result −



Multiple Lines in a Line Chart

More than one line can be drawn on the same chart by using the **lines()**function.

After the first line is plotted, the lines() function can use an additional vector as input to draw the second line in the chart,

# Create the data for the chart.

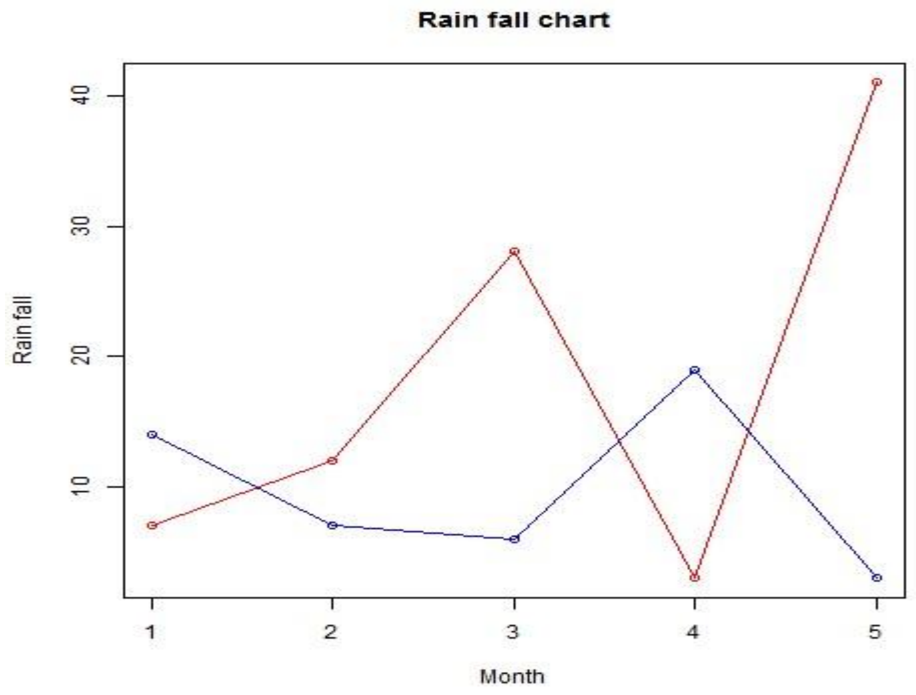
v <- c(7,12,28,3,41) t <- c(14,7,6,19,3)

# Give the chart file a name.it is not necessasary png(file = "line_chart_2_lines.jpg") # Plot the bar chart.

plot(v,type = "o",col = "red", xlab = "Month", ylab = "Rain fall",
   main = "Rain fall chart") lines(t, type = "o", col = "blue") # Save the file.

dev.off()

When we execute the above code, it produces the following result −

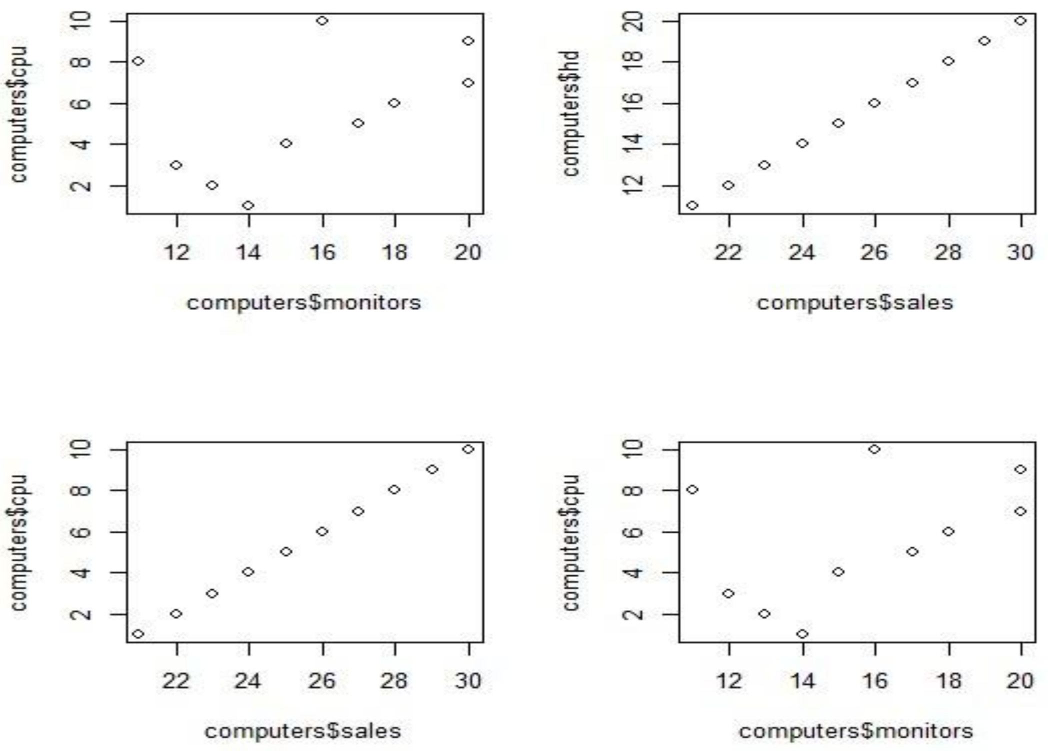


Multiple plots in one graph:

```
>print(computers)    monitors cpu hd sales inv
1       14   1 11    21   1
2       13   2 12    22   2
3       12   3 13    23   3
4       15   4 14    24   4
5       17   5 15    25   5
6       18   6 16    26   6
7       20   7 17    27   7
8       11   8 18    28   8
9       20   9 19    29   9
10      16  10 20    30  10
> str(computers)
'data.frame':   10 obs. of  5 variables:
 $ monitors: num  14 13 12 15 17 18 20 11 20 16
 $ cpu     : int  1 2 3 4 5 6 7 8 9 10
 $ hd      : int  11 12 13 14 15 16 17 18 19 20
 $ sales   : int  21 22 23 24 25 26 27 28 29 30
 $ inv     : int  1 2 3 4 5 6 7 8 9 10
```

par() this function is used for arrrange and modification of parameters: mfrow parameter:graphs are arranged in row wise.ex par(mfrow=c(2,2)) **mfcol parameter: graphs are arranged in column wise ex:par(mfcol=c(2,2))**

```
par(mfrow=c(2,2))
> plot(computers$monitors, computers$cpu)

> plot(computers$sales, computers$hd)

> plot(computers$sales, computers$cpu)

> plot(computers$monitors, computers$cpu)
```



You've seen how easy it is to build simple graphs in stages, starting with plot(). Now you can begin to enhance those graphs, using the many options R provides.

Changing Character Sizes: The cex Option

The cex (for *character expand*) function allows you to expand or shrink characters within a graph, which can be very useful. You can use it as a named parameter in various graphing functions. For instance, you may wish to draw the text "abc" at some point, say (2.5,4), in your graph but with a larger font, in order to call attention to this particular text. You could do this by typing the following:

```
text(2.5,4,"abc",cex = 1.5)
```

Changing the Range of Axes: The xlim and ylim Options:

You may wish to have the ranges on the *x*- and *y*-axes of your plot be broader or narrower than the default. This is especially useful if you will be displaying several curves in the same graph.
You can adjust the axes by specifying the xlim and/or ylim parameters in your call to plot() or points(). For example, ylim=c(0,90000) specifies a range on the *y*-axis of 0 to 90,000.

If you have several curves and do not specify xlim and/or ylim, you should draw the tallest curve first so there is room for all of them. Otherwise, R will fit the plot to the first one your draw and then cut off taller ones at the top! We took this approach earlier, when we plotted two density estimates on the same graph (Figures 12-3 and 12-4). Instead, we could have first found the highest values of the two density estimates. For d1, we find the following:

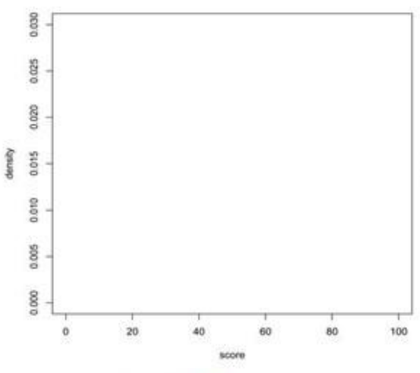
```
> d1
```

```
Call:
density.default(x = testscores$Exam1, from = 0, to = 100)
 Data: testscores$Exam1 (39 obs.); Bandwidth 'bw' = 6.967 x   y
Min. : 0         Min. :1.423e-07
1st Qu.: 25 1st
Qu.:1.629e-03
Median : 50 Median
:9.442e-03
Mean : 50      Mean :9.844e-03
3rd Qu.: 75 3rd
Qu.:1.756e-02
Max. :100 Max. :2.156e-02
```

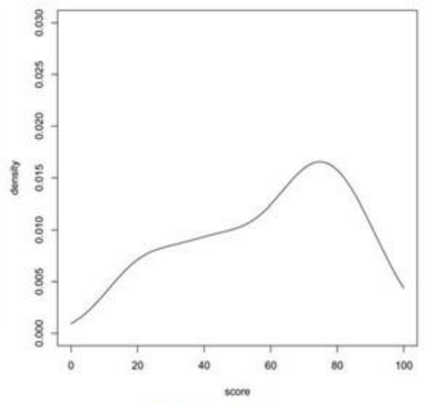
So, the largest y-value is 0.022. For d2, it was only 0.017. That means we should have plenty of room if we set ylim at 0.03. Here is how we could draw the two plots on the same picture:

```
> plot(c(0, 100), c(0, 0.03), type = "n", xlab="score", ylab="density")
> lines(d2)
> lines(d1)
```

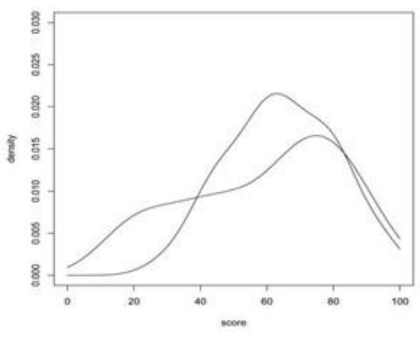
First we drew the bare-bones plot—just axes without innards, as shown in Figure 12-7. The first two arguments to plot() give xlim and ylim, so that the lower and upper limits on the Y axis will be 0 and 0.03. Calling lines() twice then fills in the graph, yielding Figures 12-8 and 12-9. (Either of the two lines() calls could come first, as we've left enough room.)



axes only



addition of d1



addition d2

Adding a Polygon: The polygon() Function:

You can use polygon() to draw arbitrary polygonal objects. For example, the following code draws the graph of the function $f(x) = 1 - e^{-x}$ and then adds a rectangle that approximates the area under the curve from $x = 1.2$ to $x = 1.4$.

```
> f <- function(x) return(1-exp(-x))
> curve(f,0,2)
> polygon(c(1.2,1.4,1.4,1.2),c(0,0,f(1.3),f(1.3)),col="gray")
```

In the call to polygon() here, the first argument is the set of *x*-coordinates for the rectangle, and the second argument specifies the *y*-coordinates. The third argument specifies that the rectangle in this case should be shaded in solid gray.

As another example, we could use the density argument to fill the rectangle with striping. This call specifies 10 lines per inch:

```
> polygon(c(1.2,1.4,1.4,1.2),c(0,0,f(1.3),f(1.3)),density=10)
```

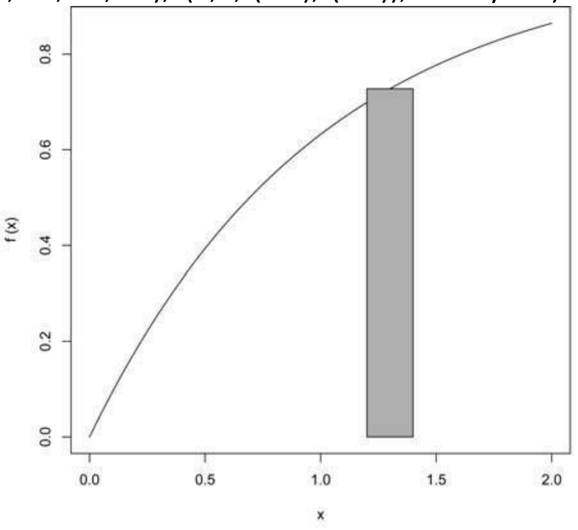


Figure 12 10: Rectangular area strip -

Smoothing Points: The lowess() and loess() Functions:

Just plotting a cloud of points, connected or not, may give you nothing but an uninformative mess. In many cases, it is better to smooth out the data by fitting a nonparametric regression estimator such as lowess().

Let’s do that for our test score data. We’ll plot the scores of exam 2 against those of exam 1:

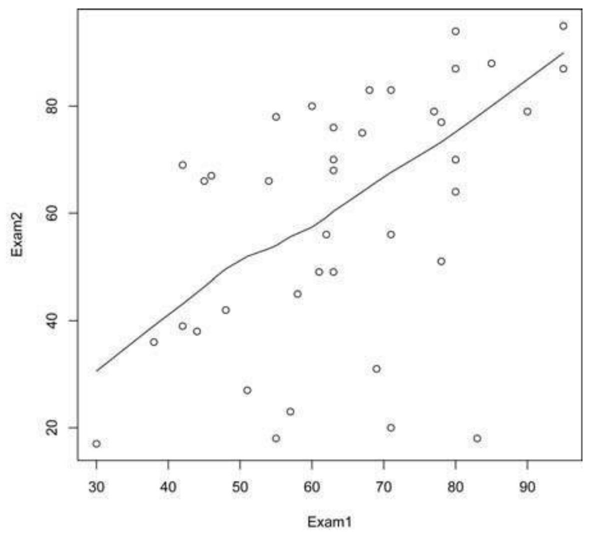
```
> plot(testscores)
> lines(lowess(testscores))
```

A newer alternative to lowess() is loess(). The two functions are similar but have different defaults and other options.

Graphing Explicit Functions

Say you want to plot the function $g(t) = (t^2 + 1)^{0.5}$ for t between 0 and 5. You could use the following R code:

```
g <- function(t) { return (t^2+1)^0.5 } # define g() x <-
seq(0,5,length=10000) # x = [0.0004, 0.0008, 0.0012,..., 5]
y <- g(x) # y = [g(0.0004), g(0.0008), g(0.0012), ..., g(5)] plot(x,y,type="l")
```



Smoothing the exam score relation

But you could avoid some work by using the curve() function, which basically uses the same method:

```
> curve((x^2+1)^0.5,0,5)
```

If you are adding this curve to an existing plot, use the add argument:

```
> curve((x^2+1)^0.5,0,5,add=T)
```

The optional argument n has the default value 101, meaning that the function will be evaluated at 101 equally spaced points in the specified range of x. Use just enough points for visual smoothness. If you find 101 is not enough, experiment with higher values of n. You can also use plot(), as follows:

```
> f <- function(x) return((x^2+1)^0.5)
> plot(f,0,5) # the argument must be a function name
```

Here, the call plot() leads to calling plot.function(), the implementation of the generic plot() function for the function class.

Again, the approach is your choice; use whichever one you prefer.

Saving Graphs to Files

The R graphics display can consist of various graphics devices. The default device is the screen. If you want to save a graph to a file, you must set up another device.

Let’s go through the basics of R graphics devices first to introduce R graphics device concepts, and then discuss a second approach that is much more direct and convenient.

R Graphics Devices Let’s open a file:

```
> pdf("d12.pdf")
```

This opens the file *d12.pdf*. We now have two devices open, as we can confirm:

```
> dev.list() X11 pdf
  2 3
```

The screen is named X11 when R runs on Linux. (It's named windows on Windows systems.) It is device number 2 here. Our PDF file is device number 3. Our active device is the PDF file:

```
> dev.cur() pdf
  3
```

All graphics output will now go to this file instead of to the screen. But what if we wish to save what's already on the screen?

Saving the Displayed Graph

One way to save the graph currently displayed on the screen is to reestablish the screen as the current device and then copy it to the PDF device, which is 3 in our example, as follows:

```
> dev.set(2)
X11
  2
> dev.copy(which=3) pdf
  3
```

But actually, it is best to set up a PDF device as shown earlier and then rerun whatever analyses led to the current screen. This is because the copy operation can result in distortions due to mismatches between screen devices and file devices.

12.3.3Closing an R Graphics Device

Note that the PDF file we create is not usable until we close it, which we do as follows:

```
> dev.set(3) pdf
  3
> dev.off()
X11
  2
```

You can also close the device by exiting R, if you're finished working with it. But in future versions of R, this behavior may not exist, so it's probably better to proactively close.

Creating Three-Dimensional Plots

R offers a number of functions to plot data in three dimensions such as persp() and wireframe(), which draw surfaces, and cloud(), which draws threedimensional scatter plots. Here, we'll look at a simple example that uses wireframe().

```
> library(lattice)
> a <- 1:10
> b <- 1:15
> eg <- expand.grid(x=a,y=b)
> eg$z <- eg$x^2 + eg$x * eg$y
> wireframe(z ~ x+y, eg)
```

First, we load the lattice library. Then the call to expand.grid() creates a data frame, consisting of two columns named x and y, in all possible combinations of the values of the two inputs. Here, a and b had 10 and 15 values, respectively, so the resulting data frame will have 150 rows. (Note that

the data frame that is input to wireframe() does not need to be created by expand.grid().)

We then added a third column, named z, as a function of the first two columns. Our call to wireframe() creates the graph. The arguments, given in regression model form, specify that z is to be graphed against x and y. Of course, z, x, and y refer to names of columns in eg. The result is shown in Figure 12-13.

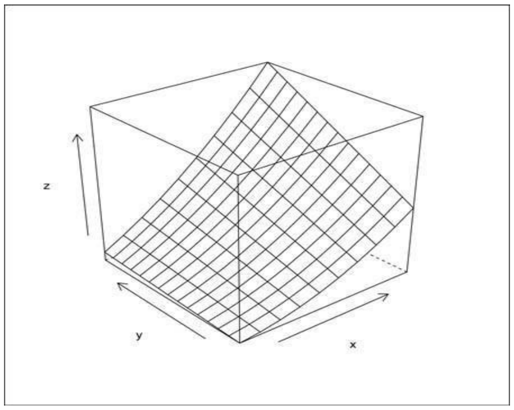


Figure 12-13: Example of using wireframe()

All the points are connected as a surface (like connecting points by lines in two dimensions). In contrast, with cloud(), the points are isolated.

For wireframe(), the (*x*,*y*) pairs must form a rectangular grid, though not necessarily be evenly spaced.

The three-dimensional plotting functions have many different options. For instance, a nice one for wireframe() is shade=T, which makes the data easier to see.

Another examples for graphs:

R - Pie Charts:

R Programming language has numerous libraries to create charts and graphs. A pie-chart is a representation of values as slices of a circle with different colors. The slices are labeled and the numbers corresponding to each slice is also represented in the chart.

In R the pie chart is created using the **pie()** function which takes positive numbers as a vector input. The additional parameters are used to control labels, color, title etc. Syntax:The basic syntax for creating a pie-chart using the R is − pie(x, labels, radius, main, col, clockwise)

Following is the description of the parameters used − **x** is a vector containing the numeric values used in the pie chart. **labels** is used to give description to the slices.

radius indicates the radius of the circle of the pie chart.(value between −1 and +1).

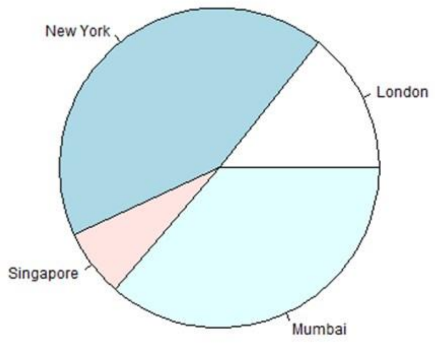
main indicates the title of the chart. **col** indicates the color palette. **clockwise** is a logical value indicating if the slices are drawn clockwise or anti clockwise.

Example

A very simple pie-chart is created using just the input vector and labels. The below script will create and save the pie chart in the current R working directory.

```
# Create data for the graph.
x <- c(21, 62, 10, 53)
labels <- c("London", "New York", "Singapore", "Mumbai") # Give the chart file a name.
png(file = "city.jpg") # Plot the chart.
pie(x,labels) # Save the file.
dev.off()
```

When we execute the above code, it produces the following result –



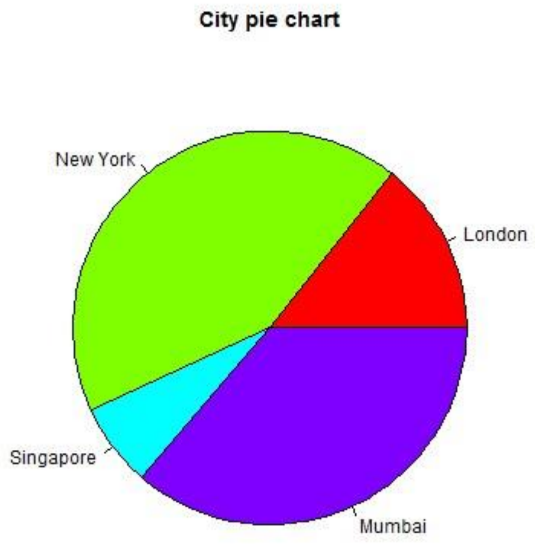
Pie Chart Title and Colors

We can expand the features of the chart by adding more parameters to the function. We will use parameter **main** to add a title to the chart and another parameter is **col** which will make use of rainbow colour pallet while drawing the chart. The length of the pallet should be same as the number of values we have for the chart. Hence we use length(x). Example

The below script will create and save the pie chart in the current R working directory.

```
# Create data for the graph.
x <- c(21, 62, 10, 53) labels <- c("London", "New York", "Singapore", "Mumbai") # Give the chart file a name.
png(file = "city_title_colours.jpg")
# Plot the chart with title and rainbow color pallet.
pie(x, labels, main = "City pie chart", col = rainbow(length(x))) # Save the file.
dev.off()
```

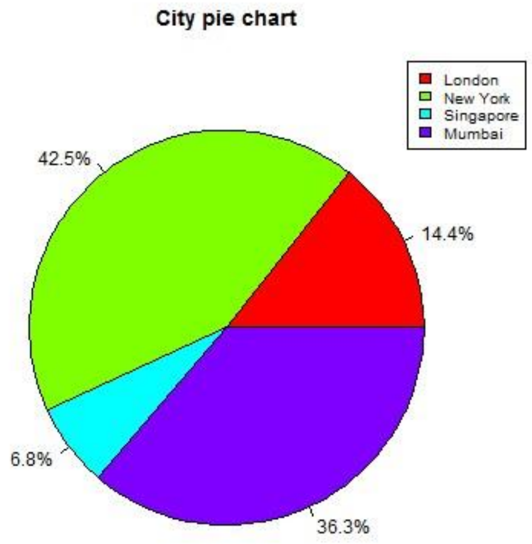
When we execute the above code, it produces the following result –



Slice Percentages and Chart Legend

We can add slice percentage and a chart legend by creating additional chart variables.

# Create data for the graph. x <- c(21, 62, 10,53)
labels <- c("London","New York","Singapore","Mumbai") piepercent<- round(100*x/sum(x), 1) # Give the chart file a name. png(file = "city_percentage_legends.jpg") # Plot the chart. pie(x, labels = piepercent, main = "City pie chart",col = rainbow(length(x))) legend("topright", c("London","New York","Singapore","Mumbai"), cex = 0.8,   fill = rainbow(length(x))) # Save the file.
dev.off()
When we execute the above code, it produces the following result –

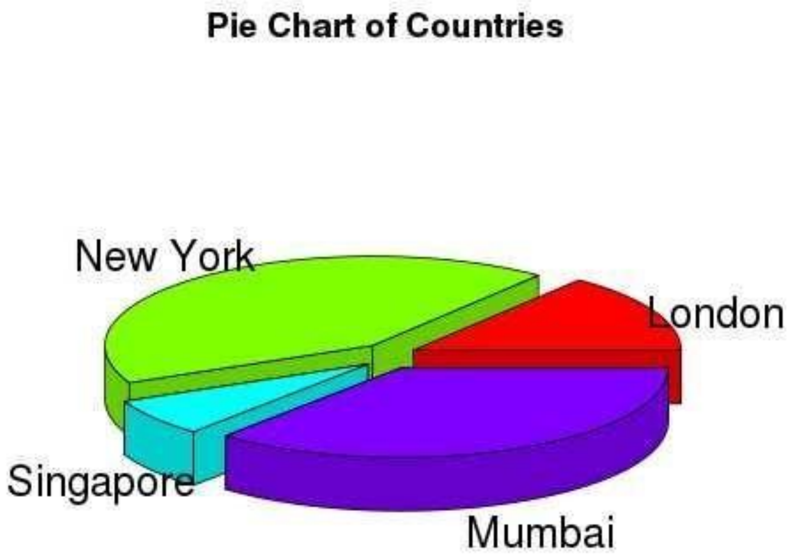


3D Pie Chart

A pie chart with 3 dimensions can be drawn using additional packages. The package **plotrix** has a function called **pie3D()** that is used for this.

# Get the library. library(plotrix)
# Create data for the graph. x <- c(21, 62, 10,53) lbl <- c("London","New York","Singapore","Mumbai") # Give the chart file a name. png(file = "3d_pie_chart.jpg") # Plot the chart.

pie3D(x,labels = lbl,explode = 0.1, main = "Pie Chart of Countries ") # Save the file.
dev.off()
When we execute the above code, it produces the following result −



R - Bar Charts

A bar chart represents data in rectangular bars with length of the bar proportional to the value of the variable. R uses the function **barplot()** to create bar charts. R can draw both vertical and horizontal bars in the bar chart. In bar chart each of the bars can be given different colors.

Syntax

The basic syntax to create a bar-chart in R is −

barplot(H, xlab, ylab, main, names.arg, col)

Following is the description of the parameters used −

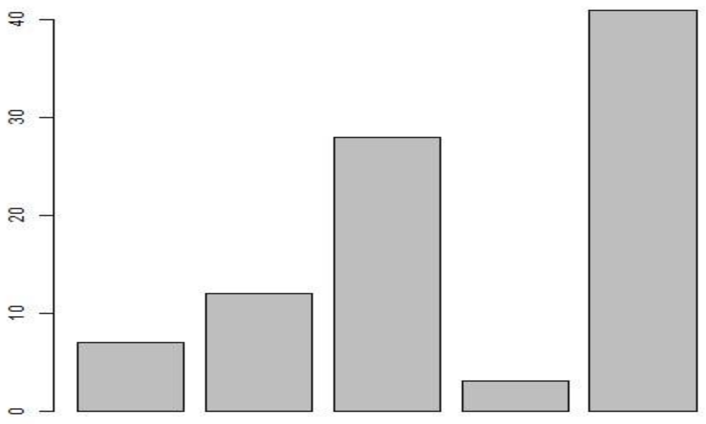
H is a vector or matrix containing numeric values used in bar chart.
xlab is the label for x axis. **ylab** is the label for y axis. **main** is the title of the bar chart.
names.arg is a vector of names appearing under each bar. **col** is used to give colors to the bars in the graph.

Example

A simple bar chart is created using just the input vector and the name of each bar.

The below script will create and save the bar chart in the current R working directory.

# Create the data for the chart.
H <- c(7,12,28,3,41) # Give the chart file a name. png(file = "barchart.png") # Plot the bar chart. barplot(H) # Save the file.
dev.off()
When we execute the above code, it produces the following result −



Bar Chart Labels, Title and Colors

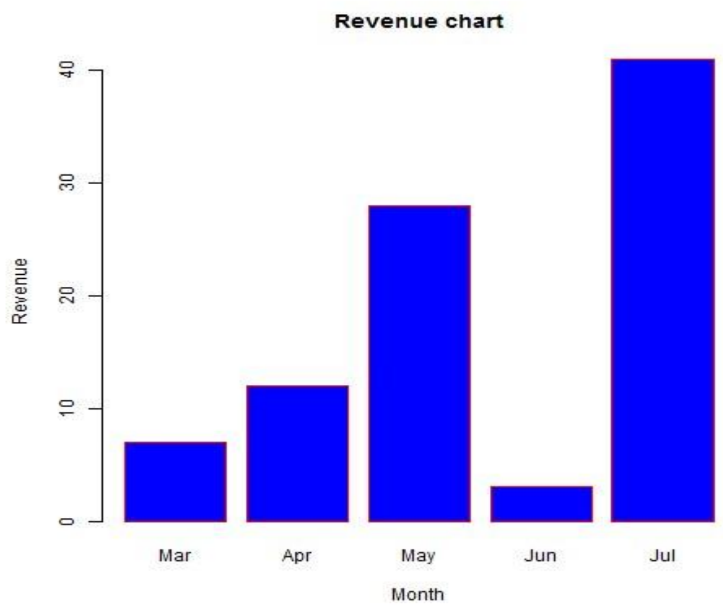
The features of the bar chart can be expanded by adding more parameters.

The **main** parameter is used to add **title**. The **col** parameter is used to add colors to the bars.

The **args.name** is a vector having same number of values as the input vector to describe the meaning of each bar.

Example

The following script will create and save the bar chart in the current R working directory. # Create the data for the chart.
H <- c(7,12,28,3,41)
M <- c("Mar","Apr","May","Jun","Jul") # Give the chart file a name.
png(file = "barchart_months_revenue.png") # Plot the bar chart.
barplot(H,names.arg = M,xlab = "Month",ylab = "Revenue",col = "blue", main = "Revenue chart",border = "red") # Save the file.
dev.off()
When we execute the above code, it produces the following result −

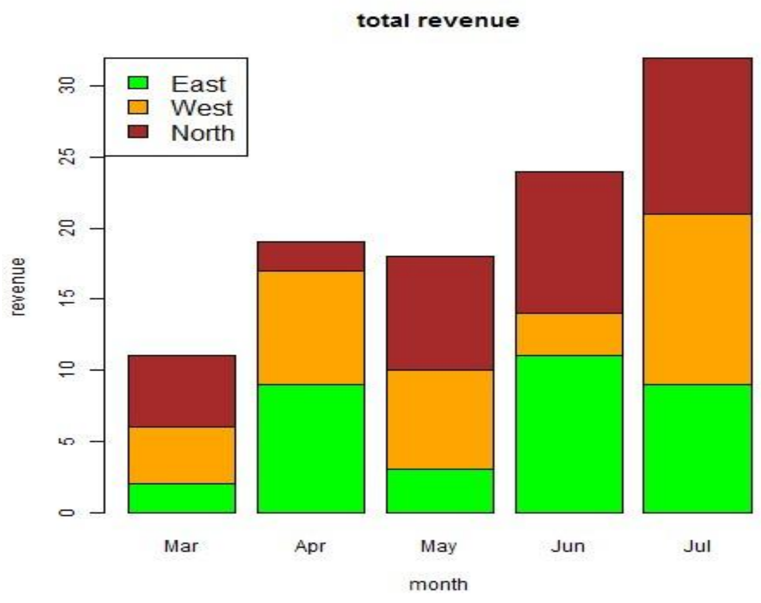


Group Bar Chart and Stacked Bar Chart

We can create bar chart with groups of bars and stacks in each bar by using a matrix as input values.

More than two variables are represented as a matrix which is used to create the group bar chart and stacked bar chart.

```
# Create the input vectors.
colors <- c("green","orange","brown") months <- c("Mar","Apr","May","Jun","Jul") regions <-
c("East","West","North") # Create the matrix of the values.
Values <- matrix(c(2,9,3,11,9,4,8,7,3,12,5,2,8,10,11),nrow = 3,ncol = 5,byrow = TRUE) #
Give the chart file a name. png(file = "barchart_stacked.png") # Create the bar chart.
barplot(Values,main = "total revenue",names.arg = months,xlab = "month",ylab =
"revenue",
  col = colors)
# Add the legend to the chart.
legend("topleft", regions, cex = 1.3, fill = colors) # Save the file.
dev.off()
```



R – Boxplots: Boxplots are a measure of how well distributed is the data in a data set. It divides the data set into three quartiles. This graph represents the minimum, maximum, median, first quartile and third quartile in the data set. It is also useful in comparing the distribution of data across data sets by drawing boxplots for each of them.

Boxplots are created in R by using the **boxplot()** function.

Syntax

The basic syntax to create a boxplot in R is −

boxplot(x, data, notch, varwidth, names, main)

Following is the description of the parameters used − **x** is a vector or a formula. **data** is the data frame. **notch** is a logical value. Set as TRUE to draw a notch.

varwidth is a logical value. Set as true to draw width of the box proportionate to the sample size. **names** are the group labels which will be printed under each boxplot. **main** is used to give a title to the graph.

Example: We use the data set "mtcars" available in the R environment to create a basic boxplot.

Let's look at the columns "mpg" and "cyl" in mtcars.

input <- mtcars[,c('mpg','cyl')] print(head(input))

When we execute above code, it produces following result − mpg cyl Mazda RX4 21.0 6

Mazda RX4 Wag 21.0 6

Datsun 710 22.8 4

Hornet 4 Drive 21.4 6

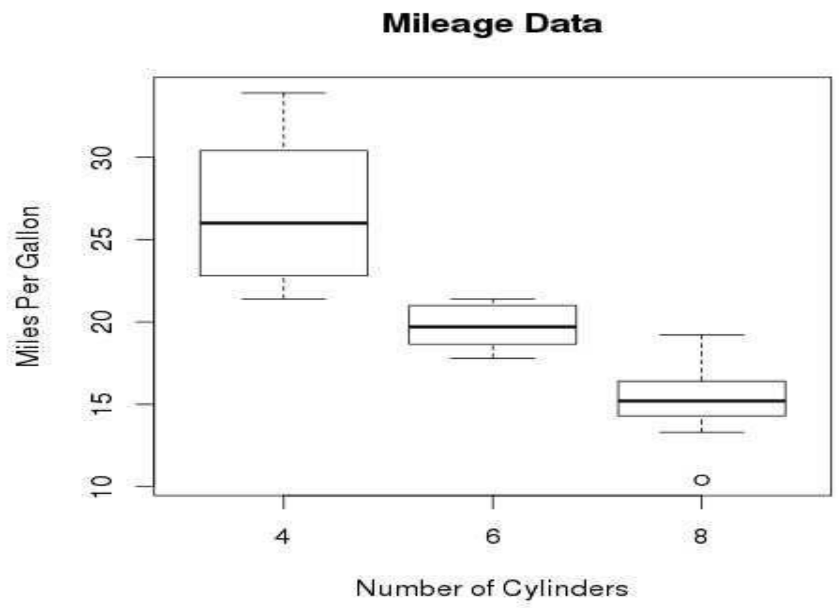
Hornet Sportabout 18.7 8

Valiant 18.1 6 **Creating the Boxplot**

The below script will create a boxplot graph for the relation between mpg (miles per gallon) and cyl (number of cylinders).

```
# Give the chart file a name.
png(file = "boxplot.png") # Plot the chart.
boxplot(mpg ~ cyl, data = mtcars, xlab = "Number of Cylinders",    ylab = "Miles Per Gallon",
main = "Mileage Data") # Save the file.
dev.off()
```

When we execute the above code, it produces the following result −



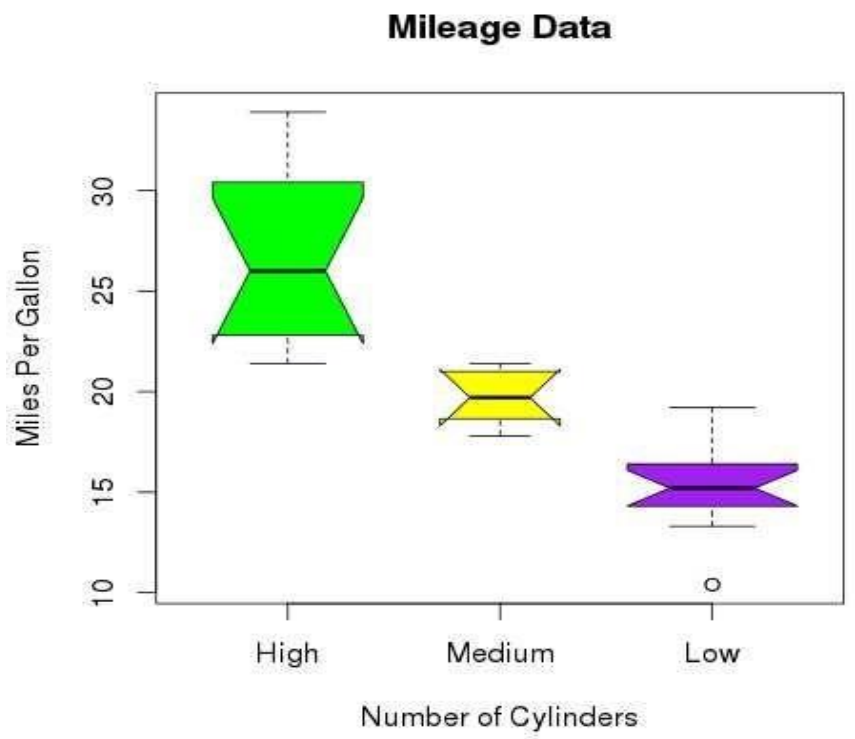
Boxplot with Notch

We can draw boxplot with notch to find out how the medians of different data groups match with each other.

The below script will create a boxplot graph with notch for each of the data group.

```
# Give the chart file a name. png(file = "boxplot_with_notch.png") # Plot the chart.
boxplot(mpg ~ cyl, data = mtcars,    xlab = "Number of Cylinders",    ylab = "Miles Per
Gallon",    main = "Mileage Data",    notch = TRUE,    varwidth = TRUE,
  col = c("green","yellow","purple"),    names = c("High","Medium","Low")
)
# Save the file.
dev.off()
```

When we execute the above code, it produces the following result −



R – Scatterplots: Scatterplots show many points plotted in the Cartesian plane. Each point represents the values of two variables. One variable is chosen in the horizontal axis and another in the vertical axis.The simple scatterplot is created using the plot() function.
Syntax: The basic syntax for creating scatterplot in R is −

plot(x, y, main, xlab, ylab, xlim, ylim, axes)

Following is the description of the parameters used − **x** is the data set whose values are the horizontal coordinates. **y** is the data set whose values are the vertical coordinates. **main** is the tile of the graph. **xlab** is the label in the horizontal axis. **ylab** is the label in the vertical axis. **xlim** is the limits of the values of x used for plotting. **ylim** is the limits of the values of y used for plotting. **axes** indicates whether both axes should be drawn on the plot.

Example: We use the data set **"mtcars"** available in the R environment to create a basic scatterplot. Let's use the columns "wt" and "mpg" in mtcars.

input <- mtcars[,c('wt','mpg')] print(head(input))

When we execute the above code, it produces the following result −

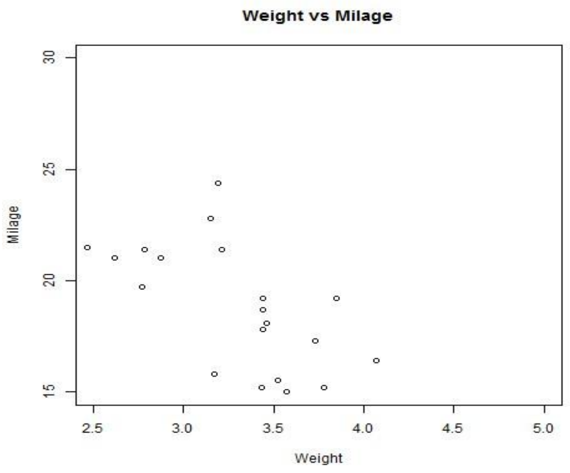
```
                      wt      mpg
Mazda RX4           2.620   21.0
Mazda RX4 Wag       2.875   21.0
Datsun 710          2.320   22.8
Hornet 4 Drive      3.215   21.4
Hornet Sportabout   3.440   18.7
Valiant             3.460   18.1
```

Creating the Scatterplot

The below script will create a scatterplot graph for the relation between wt(weight) and mpg(miles per gallon).

```
# Get the input values.
input <- mtcars[,c('wt','mpg')] # Give the chart file a name. png(file = "scatterplot.png")
# Plot the chart for cars with weight between 2.5 to 5 and mileage between 15 and 30.
plot(x = input$wt,y = input$mpg,    xlab = "Weight",    ylab = "Milage",
   xlim = c(2.5,5),
   ylim = c(15,30),
   main = "Weight vs Milage"
)
# Save the file.
dev.off()
```

When we execute the above code, it produces the following result −



Scatterplot Matrices: When we have more than two variables and we want to find the correlation between one variable versus the remaining ones we use scatterplot matrix. We use **pairs()** function to create matrices of scatterplots.

Syntax: The basic syntax for creating scatterplot matrices in R is −

```
pairs(formula, data)
```

Following is the description of the parameters used − **formula** represents the series of variables used in pairs. **data** represents the data set from which the variables will be taken.

Example: Each variable is paired up with each of the remaining variable. A scatterplot is plotted for each pair.

```
# Give the chart file a name.
png(file = "scatterplot_matrices.png")
# Plot the matrices between 4 variables giving 12 plots.
# One variable with 3 others and total 4 variables. pairs (~wt+mpg+disp+cyl, data = mtcars,
main = "Scatterplot Matrix") # Save the file.
 dev.off()
```

When the above code is executed we get the following output.